Algoritmos Híbridos para Problemas de Corte Unidimensional

Glauber Ferreira Cintra

Dissertação apresentada ao
Instituto de Matemática e Estatística da
Universidade de São Paulo para obtenção
do grau de Mestre em Matemática Aplicada

Área de Concentração: Ciência da Computação
Orientadora: Profa. Dra. Yoshiko Wakabayashi

Durante a elaboração deste trabalho o autor recebeu apoio financeiro do CNPq

—São Paulo, julho de 1998—

Algoritmos Híbridos para Problemas de Corte Unidimensional

Glauber Ferreira Cintra

Este exemplar corresponde à redação final da dissertação devidamente corrigida e defendida por Glauber Ferreira Cintra e aprovada pela comissão julgadora.

Banca examinadora:

- Profa. Dra. Yoshiko Wakabayashi (Orientadora) -IME-USP
- Prof. Dr. Flávio K. Miyazawa - IC-UNICAMP
- Prof. Dr. Nelson Maculan - COPPE-UFRJ

—São Paulo, julho de 1998—

à minha esposa Fabiana
e aos meus pais Lino e Maria

Agradecimentos

Muitas pessoas ajudaram-me durante o mestrado e aproveito este espaço para externar meu sincero agradecimento.

Agradeço aos colegas de mestrado, dentre eles o *Alexandre, Alfredo, Armando, Cesar, Fábio Kon, Fábio Viduani, Jair, Jefferson, Lee, Loparic, Marcelo, Marco Aurélio e Ricardo Ueda*, que me incentivaram e com que compartilhei bons momentos aqui no IME-USP.

Ao pessoal do Centro de Ensino de Computação do IME-USP, *Airton, Ary, Gislaine, Jeane, Rosemira, Selma* e *Serginho*, pelo companheirismo nestes anos que temos trabalhado juntos.

Aos administradores da rede do IME-USP, por mantê-la em funcionamento (*quase*) contínuo, o que permitiu realizar os testes computacionais apresentados nesta dissertação.

Aos professores que tive na graduação, em especial aos professores *Marcelino Pequeno, Plácido Pinheiro* e *Wamberto Vasconcelos*, que me instilaram o desejo de aprofundar meus conhecimentos através de uma pós-graduação. Aos professores *Imre Simon, José Augusto, Paulo Feofiloff, Routo Terada, Siang Song* e *Yoshiharu Kohayakawa*, que contribuíram para o meu aperfeiçoamento através das disciplinas que cursei durante o mestrado.

Aos professores *Carlos Ferreira* e *Horácio Yanasse* que participaram como membros da banca na minha qualificação e cujas sugestões foram valiosas para a consumação deste trabalho. Agradeço também ao professor *Gerhard Wäscher* por ter disponibilizado material bibliográfico que foi fundamental para o nosso trabalho de pesquisa.

Ao professor *Flávio Miyazawa*, pelas sugestões que me ajudaram a elaborar esta dissertação, pelo apoio, incentivo e companheirismo e pelas valiosas dicas em relação à linguagem *C*, ao *CPLEX* e ao LaTeX.

À professora *Yoshiko Wakabayashi*, por suas demonstrações de confiança e carinho, pela forma dedicada e paciente com que me orientou e por em momento algum ter tolhido minha liberdade na escolha e desenvolvimento do tema.

Aos meus pais, *Lino* e *Maria*, que plantaram as sementes que agora estou colhendo.

À minha irmã *Katia*, pelo apoio logístico que tem me dado desde que cheguei à São Paulo, pelos conselhos, carinho e incentivo para seguir a carreira acadêmica.

Agradeço em especial à *Fabiana*, minha esposa, pelo seu apoio, incentivo e amor, e pelo maravilhoso presente que me deu, nossa filha *Sarah*.

Finalmente, agradeço a *Deus*, que é a fonte de minha força e a quem credito o mérito desta conquista.

Algoritmos Híbridos para Problemas de Corte Unidimensional

Glauber Ferreira Cintra

Resumo da dissertação apresentada ao IME-USP para
obtenção do grau de Mestre em Matemática Aplicada

Resumo

Nesta dissertação apresentamos uma visão abrangente dos problemas de corte e empacotamento, analisando suas principais características, a partir das quais introduzimos a classificação proposta por Dickhoff. Discutimos brevemente as principais estratégias utilizadas na resolução destes problemas, citando algumas referências para o leitor interessado neste tópico.

Investigamos o problema de corte de estoque unidimensional, formulando-o como um problema de programação linear inteira, e propomos um algoritmo híbrido, baseado no método de geração de colunas e num algoritmo exato. Tal algoritmo exato é adequado para resolver instâncias pequenas do problema de corte unidimensional quando se conhece previamente um limitante inferior para o valor da solução inteira ótima. Mostramos ainda que o algoritmo híbrido proposto encontra uma solução inteira cujo valor objetivo difere do valor objetivo ótimo de no máximo 1, se a conjectura MIRUP (Modified Integer Round-Up Property) for verdadeira. Variações são propostas no algoritmo híbrido de modo a diminuir o tempo gasto na resolução dos problemas. Adaptamos ainda o algoritmo híbrido para o problema de corte unidimensional no qual a quantidade de itens distintos nos padrões é limitada por uma constante.

Os resultados obtidos na resolução de um expressivo número de instâncias práticas e instâncias geradas aleatoriamente são analisados, indicando um desempenho bastante satisfatório do algoritmo híbrido e suas variações.

Palavras-chave: problemas de corte e empacotamento, corte unidimensional, geração de colunas.

Orientadora: Profa. Dra. Yoshiko Wakabayashi

Hybrid Algorithms for Unidimensional Cutting Stock Problems

Glauber Ferreira Cintra

Abstract of the Master's dissertation presented at the IME-USP

Abstract

In this dissertation we present an overview of cutting and packing problems, analyzing their main features and then we introduce the tipology proposed by Dickhoff. We briefly discuss the main strategies used in the resolution of these problems, mentioning some references for the reader interested in this topic.

We investigate the unidimensional cutting stock problem, modelled as an integer linear program, and propose an hybrid algorithm that is based in the column generation method and in an exact algorithm. The exact algorithm we use is appropriate to solve small instances of the unidimensional cutting stock problem when we know previously a lower bound for the value of the optimal integer solution. We show that the proposed hybrid algorithm finds an integer solution whose objective value differs from the optimal value by at most 1, under the assumption that the MIRUP (Modified Integer Round-Up Property) conjecture is true. Variations are proposed for the hybrid algorithm in order to speed up its runtime. We adapt the hybrid algorithm for a special case of the unidimensional cutting stock problem in which the amount of distinct itens in the patterns is limited by a constant.

The results obtained in the resolution of an expressive number of real world instances as well as randomly generated instances are analyzed, indicating that the hybrid algorithm and its variations have a very satisfactory performance.

Keywords: cutting and packing problems, unidimensional cutting stock problem, column generation.

Dissertation supervisor: Profa. Dra. Yoshiko Wakabayashi

Índice

Lista de Figuras

Lista de Tabelas

Introdução

Nas últimas quatro décadas, os problemas de corte e empacotamento têm sido largamente estudados por um número crescente de pesquisadores, gerando e tendo se beneficiado de avanços significativos em diversas áreas, tais como *programação linear*, *programação dinâmica*, *teoria da complexidade computacional* e *algoritmos de aproximação*.

O interesse por estes problemas é em parte explicado por sua aparente simplicidade e grande aplicabilidade prática. São porém, problemas de natureza complexa, $\mathcal{NP}$*-difíceis* [32]. A pesquisa sobre este assunto deu origem a diversos modelos matemáticos e o desenvolvimento de variadas técnicas para lidar com a intratabilidade destes problemas.

Quanto à aplicabilidade desses problemas, especialmente nas indústrias, podemos citar os processos de corte de barras de aço e alumínio, bobinas de papel e tecido, chapas de metal e madeira, lâminas de vidro e fibra de vidro, peças de couro e carpete, blocos de isopor. Já os processos de carregamento de contêineres de navio, carrocerias de caminhões e vagões de trem são alguns exemplos práticos de problemas de empacotamento.

Pequenas melhorias nestes processos podem levar a ganhos substanciais, dependendo da escala de produção, e representar uma vantagem decisiva na competição com outras empresas do setor.

Existem ainda outras situações menos óbvias onde o problema de empacotamento ocorre. Por exemplo, determinar como arranjar os processos na memória principal de um computador de modo a diminuir a necessidade de fazer *swap* entre a memória principal e a memória secundária. Um outro exemplo seria o de sequenciar a exibição de anúncios durante os intervalos comerciais na programação de uma emissora de televisão. Enfim, existe um largo espectro de aplicações práticas para os problemas de corte e empacotamento, sendo estes relevantes para a indústria siderúrgica, metalúrgica, têxtil, de embalagens, de papel, de vidro, moveleira, para empresas de transporte rodoviário, ferroviário, marítimo, aéreo, etc.

Em sua forma mais geral, os *problemas de corte* consistem basicamente em: *dado um objeto*

com dimensões especificadas e uma lista de objetos menores do mesmo tipo (aqui chamados de itens), também com dimensões especificadas, determinar "a melhor maneira" de cortar o objeto de forma a produzir os itens da lista. Dependendo do contexto, "a melhor maneira" significa minimizar o desperdício (sobra) do objeto ou minimizar a quantidade de objetos utilizados, ou ainda maximizar o valor dos itens produzidos (neste último caso um valor está associado a cada item).

Uma definição análoga pode ser dada aos problemas de empacotamento: *dado um recipiente com dimensões especificadas e uma lista de itens, também com dimensões especificadas, determinar "a melhor maneira" de colocar estes itens dentro do recipiente.*

Na sua forma geral estes dois problemas podem ser considerados equivalentes. No entanto, nas suas diversas variantes, motivadas por aplicações práticas, eles podem aparecer com características bem diferenciadas, recebendo nomes distintos e sendo usualmente tratados com abordagens diferentes.

Organização da dissertação

São dois os objetivos principais deste trabalho: o primeiro é dar uma visão geral sobre a área, enfocando os problemas mais representativos de corte e empacotamento e os principais métodos utilizados para resolvê-los. Os dois primeiros capítulos são dedicados a este objetivo. O segundo objetivo é investigar o problema de corte unidimensional clássico e com restrição na quantidade de itens distintos nos padrões, propondo algoritmos que encontram soluções *quase-ótimas*.

Assumimos, nesta dissertação, que o leitor tenha conhecimentos básicos de complexidade computacional, programação linear, método simplex (revisado) e de técnicas de *branch-and-bound*.

No **Primeiro Capítulo** descrevemos as principais características dos problemas de corte e empacotamento e introduzimos a classificação proposta por Dickhoff para esses problemas.

A seguir, abordamos no **Segundo Capítulo** as principais estratégias de resolução encontradas na literatura. Ênfase especial é dedicada às estratégias que servem de base para os algoritmos propostos nos capítulos subsequentes da dissertação.

No **Terceiro Capítulo** passamos a abordar o problema de corte unidimensional, formulando-o como um problema de programação linear, e detalhando um algoritmo para resolver o problema baseado no método de geração de colunas. Tal algoritmo combina geração de colunas com um algoritmo exato que resolve (em tempo razoável) instâncias pequenas do problema de corte unidimensional quando é conhecido previamente um limitante inferior para o valor da solução inteira ótima. A conjectura MIRUP é discutida e mostramos que o algoritmo proposto inicialmente encontra uma solução inteira cujo valor objetivo difere do valor objetivo ótimo de no máximo

1, se esta conjectura for verdadeira. Propomos ainda uma variação no algoritmo que leva, na prática, à redução no tempo necessário para encontrar uma solução inteira.

Os resultados obtidos na resolução de um expressivo número de instâncias práticas e instâncias geradas aleatoriamente são analisados no **Quarto Capítulo**. Comparamos os resultados dos algoritmos propostos no capítulo anterior com os resultados do algoritmo *First Fit Decreasing (FFD)* e com um limitante inferior para a solução inteira ótima.

Investigamos, no **Quinto Capítulo**, o problema de corte unidimensional onde a quantidade de itens distintos nos padrões é limitada por uma constante. Mostramos como adaptar os algoritmos propostos anteriormente e analisamos as soluções encontradas para o mesmo conjunto de instâncias resolvidas anteriormente quando restringimos a quantidade de itens distintos nos padrões.

Finalmente, nas **Considerações Finais** comentamos os resultados obtidos e discutimos possíveis desdobramentos dos algoritmos propostos na dissertação, de modo a adaptá-los para a resolução de problemas de corte bi- e tridimensionais.

Capítulo 1

Classificação dos Problemas de Corte e Empacotamento

Devido à grande diversidade de situações do mundo real onde surgem problemas de corte e empacotamento, pesquisadores de várias áreas tais como engenharia, computação, matemática e economia, têm se dedicado ao estudo destes problemas. Estes estudos deram origem a diversas variantes do problema, conhecidas na literatura por diversos nomes tais como *cutting stock problem*, *bin packing problem*, *strip packing problem*, *loading problem*, *knapsack problem*, etc.

Objetivando sistematizar o estudo dos problemas de corte e empacotamento, Dyckhoff [24] propôs uma classificação fundamentada na estrutura lógica básica dos problemas de corte e empacotamento. Tal estrutura lógica tem as seguintes características:

- Existem dois grupos básicos de dados, cujos elementos definem objetos geométricos de uma ou mais dimensões. O primeiro grupo de dados contém as dimensões e as quantidades disponíveis dos *objetos grandes* (que chamaremos simplesmente de objetos), e o segundo grupo define as dimensões e as quantidades requisitadas dos *objetos pequenos (itens)*.
- O processo de corte ou de empacotamento produz combinações geométricas dos itens dentro dos objetos. Chamaremos estas combinações de *padrões*.

Usualmente os objetos e itens são considerados como tendo uma, duas, três ou mais dimensões relevantes. Nas aplicações práticas, as dimensões relevantes dos objetos e itens frequentemente representam comprimento, largura e altura. Em certos problemas práticos as dimensões podem ser peso [21, 27], tempo [82, 15], valores monetários [55, 58, 75], espaço de memória RAM [33], etc.

A seguir apresentamos as principais características dos problemas de corte e empacotamento. Assumimos, no entanto, que o leitor possui uma certa familiaridade com os problemas de corte

e empacotamento, razão pela qual deixamos de definir formalmente as variantes do problema citadas neste capítulo.

1.1 Características dos Problemas de Corte e Empacotamento

A estrutura lógica descrita anteriormente provê um esquema para uma sistematização dos problemas de corte e empacotamento. A partir desta sistematização podemos identificar características comuns em problemas que, à primeira vista, parecem não estar relacionados. Analogamente, diferenças entre problemas aparentemente similares podem ser detectadas. A classificação proposta por Dyckhoff leva em consideração características geométricas e combinatórias dos objetos, itens e processos de corte e empacotamento. A seguir descrevemos brevemente estas características.

1.1.1 Dimensionalidade

Define o número de dimensões necessárias para descrever a geometria dos padrões. Os tipos elementares são *unidimensional, bidimensional, tridimensional e multidimensional* (mais de 3 dimensões). Estes quatro tipos elementares de dimensão podem parecer bastante precisos para classificar os problemas, no entanto para certos problemas estes tipos elementares não são adequados. Por exemplo, o problema de carregamento de páletes onde a altura do pálete é restrita [23] é considerado como sendo de dimensão "2 + 1", em vez de 3. Já o problema de corte bidimensional onde somente *cortes de guilhotina* são permitidos é classificado como "1 + 1"-dimensional (um corte de guilhotina é um corte perpendicular que vai de uma aresta à aresta oposta do objeto).

1.1.2 Medidas de quantidade

Diz respeito à maneira de medir a quantidade de objetos utilizados. Tal medição pode ser *discreta (inteira)* ou *contínua (fracionária)*. No primeiro caso contamos o número de objetos utilizados. No caso contínuo, uma das dimensões dos objetos é ilimitada (contínua), portanto medimos a quantidade utilizada nesta dimensão. A combinação de problemas unidimensionais com medição contínua é frequentemente chamada de problema 1,5-dimensional [26] ou *empacotamento em faixa*, enquanto que a combinação de problemas bidimensionais com medição contínua é frequentemente chamada de problema 2,5-dimensional ou *empacotamento em altura*.

1.1.3 Formato das figuras

Outra característica importante, diretamente relacionada com a dimensionalidade, é a *figura* dos objetos e itens. A figura de um objeto ou item é definida por sua forma geométrica, seu tamanho e sua orientação.

A forma geométrica pode ser regular ou irregular. Formas regulares podem ser descritas através de um pequeno conjunto de parâmetros. A grande maioria dos problemas considerados na literatura lida com formas regulares, especialmente retângulos e paralelepípedos, no entanto figuras de formas irregulares, incluindo formas não convexas e não simétricas, são comuns em algumas aplicações industriais.

Podemos distinguir três casos no que diz respeito à orientação dos itens em relação aos objetos: (a) qualquer orientação é permitida, (b) apenas rotações de 90 graus (em uma ou mais direções) são permitidas e (c) a orientação é fixa (em todas as direções). Neste último caso dizemos que o problema é *orientado*.

1.1.4 Sortimento

Indica o número de formas e figuras distintas encontradas no problema. Na indústria de artefatos de metal, por exemplo, é comum os objetos possuírem forma retangular enquanto que os itens possuem formas variadas, incluindo retângulos, círculos, etc. Mesmo quando os objetos e os itens possuem a mesma forma, diferenças de tamanho e orientação resultam em figuras distintas. Um fator que frequentemente influencia na dificuldade de resolver um problema específico é o tamanho dos itens em relação aos objetos [41].

1.1.5 Disponibilidade

Esta característica refere-se à *quantidade* disponível de objetos e itens e à *ordem* em que podem ser utilizados. A quantidade de objetos e itens pode ser infinita ou finita. Quando lidamos com quantidades finitas, podemos ter muitos ou apenas uns poucos objetos ou itens. Tanto o problema de empacotamento quanto o problema de corte, na sua forma clássica, pressupõem uma quantidade ilimitada de objetos. Usualmente, no entanto, no problema de empacotamento têm-se poucos itens de cada figura e um grande número de figuras, ao contrário do que ocorre no problema de corte onde têm-se poucas figuras, mas um grande número de itens de cada figura. Já o problema de carregamento de páletes e o problema da mochila são caracterizados por utilizar um único objeto.

A ordem em que os objetos e os itens podem ser utilizados é outro aspecto da disponibilidade. Por exemplo, ao empacotar itens que serão utilizados em uma linha de montagem, é preciso levar

em consideração a ordem parcial existente entre os itens. Outro exemplo é o carregamento de contêineres, onde os itens têm que ser descarregados em portos diferentes.

1.1.6 Restrições dos padrões

Existem diversas restrições que podem ser impostas aos padrões em função de certas peculiaridades dos processos de produção. Destacamos quatro grupos de restrições:

– Distâncias mínimas ou máximas entre os itens ou entre os cortes são frequentemente importantes, por exemplo, no corte de vidro ou no carregamento de contêineres.

– A orientação dos itens em relação ao objeto tem que ser levada em consideração, por exemplo, no carregamento de itens frágeis em páletes.

– Em certas situações o número de vezes que um item pode aparecer no padrão ou o número de figuras distintas no padrão deve ser limitado.

– O tipo e o número de cortes permitidos também podem ser restritos. Quando os objetos têm formas retangulares, os padrões podem ser classificados como *ortogonais*, se todos os cortes são perpendiculares a algum dos lados do objeto, ou *não-ortogonais*, caso contrário. No corte de guilhotina pode haver restrição com respeito ao número de estágios de cortes, bem como no número de cortes paralelos por estágio. Cada mudança na direção dos cortes determina um novo estágio de corte.

1.1.7 Restrições de alocação

O processo de alocação dos itens dentro dos objetos, gerando os padrões, pode apresentar diversas restrições, muitas delas estreitamente relacionadas com a disponibilidade dos itens e objetos, característica discutida anteriormente. Algumas restrições comuns são:

– O tipo de alocação determina, tanto para os objetos quanto para os itens, se todos ou apenas uma parte deles deve ser utilizada. Em princípio, podemos ter quatro categorias, sendo que duas delas têm recebido destaque na literatura. O problema da mochila é um dos exemplos da categoria em que devem ser utilizados todos os objetos, mas somente um subconjunto dos itens. Já o problema clássico de corte de estoque é um exemplo da categoria onde todos os itens devem ser alocados, utilizando-se apenas uma parte dos objetos disponíveis.

– O número de estágios de alocação define se os itens devem ser cortados ou empacotados simultaneamente ou em estágios sucessivos. No segundo caso as figuras residuais de um estágio tornam-se os objetos dos estágios seguintes. Isso sugere a possibilidade de utilização de algoritmos recursivos para resolver o problema.

– A sequência de alocação pode ser restrita, seja por razões tecnológicas, seja pela existência de ordenação entre os objetos ou itens. A quantidade permitida de padrões distintos ou de padrões iguais limita o número e a frequência dos padrões, constituindo-se também em restrições de alocação.

– A alocação dos itens dentro dos objetos pode ser de natureza estática ou dinâmica. Na alocação estática os itens e objetos são alocados sem possibilidade de remanejamento e normalmente não são conhecidos os próximos itens ou objetos a serem utilizados, característica básica dos processos de corte e empacotamento ditos *on-line*. Em contrapartida, nos processos *off-line* a alocação é dinâmica, pois existe a possibilidade de realocação dos itens dentro dos objetos.

1.1.8 Objetivos

Nem sempre é possível definir exatamente se uma característica é geométrica ou combinatória. Algumas incluem os dois aspectos, outras nenhum. Os objetivos dos problemas de corte e empacotamento frequentemente têm aspectos geométricos, outras vezes têm aspectos combinatórios. Ademais, os objetivos podem estar relacionados com os objetos, com os itens, com os padrões ou com os processos de alocação. O objetivo de um problema de corte ou empacotamento pode ser definido como um critério a ser maximizado ou minimizado. Podemos citar como exemplos de objetivos:

– Minimizar a quantidade de objetos utilizados;

– Minimizar o tempo gasto no processo de alocação;

– Minimizar o desperdício nos padrões;

– Maximizar o valor dos itens produzidos. Os itens podem ter valores arbitrários ou valores relacionados ao seu tamanho.

A lista anterior, embora não exaustiva, relaciona os objetivos dos principais problemas encontrados na literatura. Também são comuns problemas onde vários objetivos tem que ser considerados (*cf.* Wäscher [83]).

1.1.9 Status da informação e variabilidade

Os dados dos problemas podem ser determinísticos, estocásticos ou incertos. Além disso, os dados podem ser valores estritos ou intervalos de valores. Por exemplo, no corte de barras de aço a serem usadas em estruturas de concreto armado é comum os objetos possuírem tamanhos levemente variados. Além disso, o peso dos objetos, que deveriam ter a mesma dimensão, também varia, provocando o que chamamos de desbitolamento. A inexatidão nos processos de medição podem também ocasionar variabilidade dos dados.

Características dos...	Características geométricas	Características combinatórias	Outras características
Objetos e itens	Dimensionalidade Formato das figuras	Medidas de quantidade Sortimento Disponibilidade	Objetivos Status da informação Variabilidade
Padrões	Dimensionalidade Restrições dos padrões (figuras; tipos de corte; distâncias; orientação...)	Restrições dos padrões (número de cortes; tipo, número e combinação das figuras)	Objetivos Status da Informação Variabilidade
Processos de alocação	-	Restrições no número de estágios, ordem ou frequência dos padrões	Objetivos Status da informação Variabilidade

Tabela 1.1: Sistematização das principais características.

1.2 Classificação de Dyckhoff

Certamente as características abordadas na seção anterior não incluem todas as possíveis propriedades dos problemas de corte e empacotamento. Além disso, várias características citadas se sobrepõem. Por exemplo, a disponibilidade e o tipo de alocação são características estreitamente relacionadas. Usando estas características, Dyckhoff propôs um esquema que permite integrar os diversos tipos de problemas de corte e empacotamento de maneira consistente e sistemática, estabelecendo uma classificação abrangente.

Tal esquema define classes de problemas combinando-se os tipos principais de quatro características básicas. As quatro características assim como seus tipos principais, denotados por seus respectivos símbolos, são:

1. Dimensionalidade
   (1) Unidimensional
   (2) Bidimensional
   (3) Tridimensional
   (N) N-dimensional ($N > 3$)
2. Tipo de alocação
   (B) Todos os objetos e uma parte dos itens
   (V) Uma parte dos objetos e todos os itens

3. Sortimento dos objetos
   (O) Um objeto
   (I) Objetos de figuras idênticas
   (D) Objetos de diferentes figuras

4. Sortimento dos itens
   (F) Poucos itens (ou figuras diferentes)
   (M) Muitos itens de muitas figuras diferentes
   (R) Muitos itens de relativamente poucas figuras distintas
   (C) Figuras congruentes (mesma forma e tamanho)

Cada tipo de problema de corte e empacotamento é definido como sendo uma quádrupla $\alpha/\beta/\gamma/\delta$, onde α é a dimensionalidade, β o tipo de alocação, γ o sortimento dos objetos e δ o sortimento dos itens. Por exemplo, a quádrupla 1/V/I/R denota a classe de problemas unidimensionais onde todos os itens, de relativamente poucas figuras distintas, devem ser alocados dentro de objetos de mesmo tamanho.

É possível obter classes mais abrangentes deixando de especificar parte dos símbolos da quádrupla, indicando que as características não especificadas podem ser de qualquer um dos tipos principais. Por exemplo, a classe 1/B/O/ inclui o problema clássico da mochila, onde o sortimento dos itens pode ser de qualquer tipo.

Na Tabela 1.2 listamos os principais problema de corte e empacotamento abordados na literatura, indicando a classe a que pertencem. Pela tabela podemos perceber que dentro de uma mesma classe podemos encontrar diversos problemas que se diferenciam por outras características além das quatro principais adotadas na classificação. Por exemplo, o problema de alocação de memória, o *bin packing* clássico, o problema da linha de montagem e o problema de alocação de tarefas em multiprocessador pertencem todos à mesma classe (1/V/I/M), mas possuem características diferentes não incluídas na classificação básica aqui definida.

Ressaltamos aqui que o *bin packing* clássico (1/V/I/M) e o *problema de corte de estoque unidimensional* (1/V/I/R) se diferenciam apenas quanto ao sortimento dos itens. Nas aplicações industriais, é mais comum a ocorrência deste último. Este é o problema que investigamos nesta dissertação, que para simplificar será referido como *problema de corte unidimensional*. Trataremos da variante em que há uma demanda associada a cada item.

Problema	Classe
Mochila clássico	1/B/O/
Mochila multidimensional	/B/O/
Carregamento de páletes	2/B/O/C
Carregamento de veículos	1/V/I/F ou 1/V/I/M
Carregamento de contêineres	3/V/I/ ou 3/B/O/
Bin packing clássico	1/V/I/M
Bin packing dual	1/B/O/M
Bin packing bidimensional	2/V/D/M
Empacotamento em faixa	2/V/O/M
Empacotamento em altura	3/V/O/M
Corte de estoque unidimensional(*)	1/V/I/R
Corte de estoque bidimensional	2/V/I/R
Corte de estoque generalizado	1///, 2/// ou 3///
Linha de montagem	1/V/I/M
Alocação de tarefas em multiprocessador	1/V/I/M
Alocação de memória	1/V/I/M
Planejamento de investimentos multiperiódicos	N/B/O/

Tabela 1.2: Problemas comuns na literatura e sua classificação.

(*) Este é o problema estudado nesta dissertação, referido simplesmente como *problema de corte unidimensional* ou *problema de corte linear*.

Capítulo 2

Estratégias de Resolução

Várias abordagens têm sido propostas para se resolver as diversas variantes do problema de corte, sendo vasta a literatura a este respeito. Não é nosso propósito aqui discorrer longamente sobre as dezenas de algoritmos propostos para as diversas variantes deste problema. Pretendemos apenas exemplificar as principais abordagens com alguns dos algoritmos encontrados na literatura. Neste capítulo, assumimos novamente que o leitor possui certa familiaridade com os problemas de corte e empacotamento, de modo que deixamos de detalhar as características dos problemas. Ademais, apresentamos apenas uma descrição breve e informal dos algoritmos.

Diversos artigos de revisão bibliográfica sobre o assunto têm sido escritos nos últimos anos. Recomendamos ao leitor interessado os artigos de Dyckhoff et al. [25, 26], Sweeny e Paternoster [77], Dowsland [23], Coffman, Garey e Johnson [16], Hinxman [44] e Golden [40]. No final deste capítulo, apresentamos uma tabela contendo diversos algoritmos encontrados na literatura para os problemas de corte e empacotamento.

Podemos dividir as abordagens em três categorias: algoritmos exatos, algoritmos de aproximação e métodos heurísticos. Algoritmos exatos conduzem à uma solução ótima. Porém os algoritmos exatos conhecidos têm complexidade de tempo exponencial, sendo portanto aplicáveis apenas a problemas de pequeno e médio porte.

Algoritmos de aproximação encontram em tempo polinomial uma solução que se aproxima da solução ótima e possuem uma *garantia de desempenho*. Tal garantia, no caso de desempenho absoluto, é uma constante α tal que, no caso de problemas de minimização, a razão entre o valor da solução encontrada pelo algoritmo e o valor de uma solução ótima, para qualquer instância do problema, é no máximo α.

Os métodos heurísticos se propõem a achar uma solução "boa" em tempo "aceitável", geralmente polinomial, mas não proporcionam uma garantia para a qualidade da solução fornecida em relação à solução ótima. Uma tendência recente é atacar o problema de corte de estoque usando algoritmos híbridos, que combinam algoritmos exatos com esquemas de aproximação e

métodos heurísticos. A seguir apresentaremos alguns exemplos das abordagens mencionadas anteriormente.

2.1 Algoritmos Exatos

Consideraremos primeiramente o problema de corte de estoque unidimensional, no qual uma demanda é associada a cada item. Este problema pertence à classe 1/V/I/R, segundo a classificação de Dyckhoff, e consiste em: dado um objeto, genericamente denominado de *barra*, de comprimento L, e uma lista de m itens, cada item i com comprimento l_i e demanda $d_i \in I\!N$ $(i = 1, \ldots, m)$, determinar o menor número de barras necessário para atender a demanda, ou seja, produzir d_i itens, cada qual de comprimento l_i.

Vejamos como formular este problema como um problema de Programação Linear Inteira (PLI). Chamamos de padrão de corte (ou simplesmente *padrão*) cada possível forma de cortar uma barra. Mais precisamente, supondo que haja m itens $1, \ldots, m$, um padrão j pode ser representado por um vetor-coluna a_j com m elementos, cujo i-ésimo elemento indica o número de vezes que o item i ocorre nesse padrão. Assim, uma vez determinados todos os padrões possíveis (chamados viáveis), digamos que sejam n, o problema agora consiste em considerar os n padrões viáveis e decidir quantas vezes cada padrão deve ser utilizado de modo a atender a demanda, minimizando o número total de barras utilizadas.

Assim, para formular este problema como um PLI, introduzimos uma variável $x = [x_1, \ldots, x_n]$ onde cada elemento x_j indica quantas vezes o padrão j é selecionado. Note que se x é uma solução viável do problema acima, então o valor $\sum_{j=1}^{n} x_j$ corresponde ao número de barras a serem cortadas, já que cada padrão corresponde a uma barra. Assim, denotando por A a matriz $m \times n$ cujas colunas são os vetores $a_1, a_2, \ldots, a_n$, e representando por d o vetor das demandas, o problema pode ser assim formulado:

$$\begin{aligned} &\min \textstyle\sum_{j=1}^{n} x_j \\ &\text{sujeito a } Ax = d \\ &\qquad x_j \geq 0 \text{ e inteiro} \qquad j = 1, \ldots, n. \end{aligned} \tag{2.1}$$

Vance *et al.* [79], propuseram um algoritmo para o caso especial deste problema (chamado de problema de corte de estoque unidimensional binário) onde a demanda de cada item é exatamente 1. O algoritmo proposto usa *geração de colunas* e *branch-and-bound* para obter soluções inteiras ótimas.

Usando o algoritmo *FIRST FIT DECREASING*, que será explicado na próxima seção, obtém-se uma solução inicial, cuja matriz correspondente, acrescida da matriz identidade de

ordem m constituirão a matriz A inicial. Resolve-se então a relaxação linear de (2.1):

$$\begin{aligned} &\min \textstyle\sum_{j=1}^{n} x_j \\ &\text{sujeito a } Ax \geq d \\ &\qquad x_j \geq 0 \qquad\qquad j = 1, \ldots, n \ . \end{aligned} \tag{2.2}$$

Se (x, y) é a solução corrente, onde $y = [y_1, \ldots, y_m]$ é a variável dual correspondente, a nova coluna é gerada resolvendo-se o seguinte problema da mochila:

$$\begin{aligned} &\max \textstyle\sum_{i=1}^{m} y_i z_i \\ &\text{sujeito a } \textstyle\sum_{i=1}^{m} z_i l_i \leq L \\ &\qquad z_i \in \{0, 1\} \qquad\qquad i = 1, \ldots, m \ . \end{aligned} \tag{2.3}$$

A solução ótima desse problema é encontrada, na maioria dos casos, usando-se o *algoritmo guloso* ou o *algoritmo Horowitz-Sahni* ou o *algoritmo Horowitz-Sahni Modificado* [45], todos polinomiais. No entanto, em alguns casos é preciso usar um otimizador inteiro de uso geral. Quando o valor da função objetivo correspondente à solução ótima de (2.3) for menor ou igual a 1 então a solução corrente de (2.2) é ótima (note que esta condição implica que y é solução ótima do problema dual). Se a solução de (2.2) for fracionária, introduzimos as restrições de *branching*:

$$\sum_{s:a_{ks}=1, a_{ls}=1} x_s \geq 1 \qquad \text{e} \qquad \sum_{s:a_{ks}=1, a_{ls}=1} x_s \leq 0,$$

onde k e l são tais que

$$0 < \sum_{s:a_{ks}=1, a_{ls}=1} x_s < 1,$$

e retomamos o processo de geração de colunas. Em [79] é demonstrada a convergência do método, que conduz a uma solução inteira ótima.

Consideraremos agora o problema de corte de estoque bidimensional, sem demanda associada a cada item, no qual todos os cortes devem ser cortes de guilhotina. Tal problema é frequentemente chamado de problema de corte de guilhotina bidimensional. Herz [43] propôs um algoritmo recursivo para este problema. Basicamente, este algoritmo calcula uns pontos de discretização da seguinte forma:

Sejam C e L o comprimento e a largura da placa, respectivamente, c_i e l_i o comprimento e a largura do item i $(i = 1, \ldots, m)$. Sejam p e q os vetores-coluna de elementos c_i e l_i,

respectivamente. Denote por z um vetor-linha de elementos não negativos tal que $zp \leq C$ e $zq \leq L$. É possível mostrar que existe um número finito de vetores z que satisfazem estas restrições. Sejam P e Q os conjuntos finitos de valores de zp e zq, respectivamente. Os elementos de P e Q são chamados de *pontos de discretização*.

Um padrão é dito *canônico* se todos os cortes são feitos nos pontos de discretização. Ademais, se $\wp$ é um padrão que corresponde a uma solução ótima, então a seguinte propriedade recursiva é válida: os padrões correspondentes às subplacas obtidas através de um corte de guilhotina em $\wp$ também são ótimos. Usando então os pontos de discretização o algoritmo calcula recursivamente as soluções parciais ótimas até obter uma solução final ótima. Usando recursividade e programação dinâmica, Beasley [8] resolveu o mesmo problema obtendo resultados satisfatórios mesmo para problemas de médio porte.

2.2 Algoritmos de Aproximação

A principal característica dos algoritmos de aproximação é ser eficiente (polinomial) e ter garantia de desempenho. Essa garantia é uma medida da proximidade da solução encontrada pelo algoritmo em relação à solução ótima do problema.

Seja $\mathcal{X}$ um algoritmo para uma classe de problemas de minimização P. Seja $\mathcal{X}(I)$ o valor da solução encontrada pelo algoritmo $\mathcal{X}$ para uma instância $I \in P$. Seja $OPT(I)$ o valor da solução ótima de I. Sejam α e β constantes. Se

$$\mathcal{X}(I) \leq \alpha \cdot OPT(I) + \beta \quad \text{para todo} \quad I \in P$$

dizemos que α é um limite de desempenho *assintótico* para o algoritmo $\mathcal{X}$. Se $\beta = 0$, dizemos que α é um limite de desempenho *absoluto* para o algoritmo $\mathcal{X}$.

Considere o problema de empacotamento (corte) unidimensional e uma instância I deste problema que consiste de uma lista de m itens e barras de comprimento L. O mais simples dentre os algoritmos de aproximação que resolvem este problema é o *NEXT FIT (NF)*. Consideremos que as barras L estão indexadas. Iniciamos então o processo empacotando o item 1 na barra L_1. Suponha que o item i é o próximo item a ser empacotado, e L_j seja a barra não vazia com o maior índice. Se o comprimento do item i for menor que a parte vazia de L_j então empacotamos o item i em L_j, caso contrário, empacotamos o item i na barra L_{j+1}. Repetimos este procedimento até que todos os itens tenham sido empacotados. Claramente este algoritmo é bem rápido (linear), e é fácil mostrar que $NF(I) \leq 2 \cdot OPT(I)$. Neste caso, dizemos que 2 é um *limite de desempenho absoluto* deste algoritmo. Sabe-se ainda que este limite não pode ser melhorado, em outras palavras, é justo.

É possível obter um algoritmo com melhor desempenho com uma variação simples: ao empacotar o item i procuramos a barra não vazia de menor índice onde caiba o item i. Caso não

exista tal barra passamos a utilizar uma nova barra. Este algoritmo é chamado de *FIRST FIT (FF)*. Pode-se mostrar que $FF(I) \leq \frac{17}{10} \cdot OPT(I) + 2$, sendo que o limite $\frac{17}{10}$ também é justo [48, 31]. Neste caso, dizemos que o *limite de desempenho assintótico* deste algoritmo é $\frac{17}{10}$.

Outra modificação simples, que consiste em primeiramente ordenar os elementos da lista de itens em ordem decrescente de comprimento, melhora consideravelmente o desempenho. Essa é a estratégia do algoritmo *FIRST FIT DECREASING (FFD)*. Pode-se mostrar (não tão facilmente) que $FFD(I) \leq \frac{11}{9} \cdot OPT(I) + 4$, e que este limite é justo [48, 3, 46]. Vários outros algoritmos de aproximação para este problema foram propostos na literatura [48, 46, 47, 51], destacando-se o *MODIFIED FIRST FIT DECREASING (MFFD)*. Neste caso sabe-se que $MFFD(I) \leq \frac{71}{60} \cdot OPT(I) + 4$.

Existem ainda algoritmos que fornecem um esquema de aproximação em que o limite de desempenho assintótico pode chegar tão próximo de 1 quanto se queira. Um exemplo é o algoritmo linear FL, descoberto por Fernandez de la Vega e Lueker [29], para o qual $FL(I) \leq (1+\epsilon) \cdot OPT(I) + (\frac{1}{\epsilon})^2$, $\epsilon > 0$. Karmakar e Karp [49] propuseram um algoritmo (KK) tal que $KK(I) \leq OPT(I) + \mathcal{O}(\frac{\log^2 OPT(I)}{OPT(I)})$. Estes algoritmos têm significado principalmente teórico, devido ao fato de que as constantes aditivas envolvidas são demasiadamente grandes para aplicações práticas.

Existem ainda diversos algoritmos de aproximação que têm sido propostos para o problema de empacotamento em faixa (bidimensional) [5, 17] e em altura (tridimensional) [60, 61, 62].

2.3 Métodos Heurísticos

Os primeiros métodos heurísticos utilizados para resolver o problema de corte unidimensional (com demanda) foram propostos por Gilmore e Gomory [37, 38, 39]. As estratégias propostas por estes autores consistem basicamente em formular este problema como um PLI (como mencionamos na Seção 2.1), resolver a relaxação linear de (2.1) e arredondar para cima a solução x obtida (isto é, arredondar cada x_i para $\lceil x_i \rceil$).

A grande dificuldade de se implementar estas estratégias está no tratamento do número (em geral) excessivamente grande de colunas da matriz A. Para se desvencilhar desta dificuldade, Gilmore e Gomory propuseram um método de geração de colunas, onde a partir de uma solução viável x (a solução inicial pode ser facilmente obtida tomando-se a matriz identidade de ordem m como a matriz A), uma nova coluna é gerada visando obter uma solução melhor. Assim, se (x, y) é a solução corrente, onde $y = [y_1, \ldots, y_m]$ é a variável dual correspondente, a nova coluna é gerada resolvendo-se o seguinte problema da mochila:

$$\begin{aligned}
&\max \textstyle\sum_{i=1}^{m} y_i z_i \\
&\text{sujeito a } \textstyle\sum_{i=1}^{m} z_i l_i \leq L \\
&\qquad z_i \geq 0 \text{ e inteiro} \qquad\qquad i = 1, \ldots, m \ .
\end{aligned}$$

A solução ótima $z = [z_1, \ldots, z_m]$ constitui a nova coluna. Este processo de geração de colunas termina quando o valor da solução ótima do problema da mochila for menor ou igual a 1. Esta condição é suficiente para que a solução corrente x seja a solução ótima da relaxação linear de (2.1). Estes autores propuseram dois métodos para resolver o problema da mochila, um deles usando programação dinâmica e outro usando recursividade.

Arredondar para cima a solução da relaxação linear de (2.1) contorna o problema de encontrar uma solução inteira, no entanto a otimalidade da solução não é garantida. Os métodos propostos por Gilmore e Gomory têm sua importância por terem sido os precursores da maior parte dos algoritmos para o problema de corte de estoque que usam programação linear.

Diversas outras heurísticas têm sido propostas na tentativa de se resolver o problema de corte de estoque. Tais heurísticas basicamente consistem em estabelecer certas restrições que devem ser observadas ao gerar padrões de corte. Estas restrições têm como objetivo diminuir o espaço de busca de uma solução, restringindo o número de padrões viáveis.

Algumas destas restrições são puramente arbitrárias, enquanto que outras estão associadas a características práticas dos processos de corte e empacotamento no mundo real. Focalizaremos as idéias propostas por Christofides e Whitlock [12] e por Wang [81] com o objetivo de resolver o problema de corte de guilhotina bidimensional sem demanda associada a cada item.

Christofides e Whitlock propuseram associar a cada item um inteiro b_i $(i = 1, \ldots, m)$, onde b_i é o número máximo de vezes que o item i pode ser produzido em um padrão qualquer. Este problema é conhecido como problema de corte de guilhotina bidimensional *restrito*.

Levando em consideração esta restrição, são calculados uns pontos de discretização usando programação dinâmica e depois a solução "ótima" é encontrada percorrendo um árvore de busca onde o caminho da raiz até uma folha representa um padrão. Ao percorrer a árvore são calculados limites inferiores, que são usados para otimizar a busca na árvore. Este método se mostrou efetivo para a resolução de problemas de médio porte. Entretanto a solução encontrada não necessariamente é a solução ótima do problema irrestrito.

Em [81], Wang também abordou o problema de corte de guilhotina bidimensional restrito. Ele propôs estabelecer a priori um valor $0 \leq \beta \leq 1$, tal que a área desperdiçada em qualquer solução parcial do problema fosse no máximo βCL, onde C e L representam o comprimento e a largura da placa, respectivamente. A solução é encontrada percorrendo uma árvore de busca onde a cada nó da árvore é verificado se o desperdício ultrapassou o limite máximo permitido.

Em caso afirmativo a solução parcial é descartada e um novo caminho da árvore é explorado.

Oliveira e Ferreira [65] propuseram uma pequena modificação no algoritmo proposto por Wang, resultando num ganho significativo de desempenho. A idéia consiste em rejeitar tão logo quanto possível as soluções parciais que com certeza levarão a soluções cujo desperdício total ultrapassará o limite permitido. Para isso, a cada nó da árvore o desperdício da solução parcial é somado ao desperdício mínimo das áreas ainda não cortadas. Tal desperdício pode ser estimado resolvendo através de programação dinâmica o problema irrestrito nas subplacas remanescentes.

Nesta heurística a escolha de β é crucial. Se escolhermos β muito pequeno corremos o risco não encontrar solução alguma. Por outro lado, se β for muito grande podemos não obter a desejada redução no tempo de execução do algoritmo.

Um outro método heurístico que pode ser aplicado a diversas variantes do problema de corte de estoque é conhecido como *Repeated Pattern Exhaustion (RPE) Technique* [66]. Esta técnica consiste em gerar um padrão com o menor desperdício possível e usar tal padrão o maior número de vezes, o que é determinado pela demanda e a frequência com que cada item aparece no padrão. As demandas são então atualizadas e o processo é repetido até que a demanda de todos os item tenha sido atendida. Uma das dificuldades desta técnica é a de gerar um padrão com o menor desperdício possível (que depende da variante do problema de corte de estoque que estamos abordando). Para o problema de corte de guilhotina bidimensional, o algoritmo exato proposto por Herz (que comentamos na Seção 2.1) é capaz de encontrar um padrão com o menor desperdício possível.

Em 1990, Stadtler [74] propôs um algoritmo para resolver o problema de corte de estoque unidimensional. Tal algoritmo consiste basicamente em resolver a relaxação linear de (2.1), fixar as variáveis inteiras e arredondar para cima a variável cuja parte fracionária seja mais próxima de 1. O processo é repetido para o problema residual até que todas as variáveis sejam inteiras. Note que a cada iteração, pelo menos uma variável torna-se inteira, portanto o processo leva no máximo m iterações. Os valores de certas variáveis são reduzidos de uma unidade de modo a evitar produção em excesso. O problema residual resultante é então resolvido usando-se o algoritmo FFD.

O algoritmo proposto por Stadtler foi modificado por Wäscher e Gau [86], resultando numa sensível melhoria na qualidade da solução. A partir de uma solução da relaxação linear de (2.1), as variáveis fracionárias são arredondadas para baixo. Enquanto alguma das variáveis arredondadas for maior que zero o processo é repetido para o problema residual. Se ainda houver demanda não atendida, o último problema residual é então resolvido usando-se o algoritmo MTP [59].

Na Tabela 2.1 listamos um expressivo número de algoritmos encontrados na literatura para os problemas de corte e empacotamento, alguns dos quais foram abordados brevemente neste

capítulo. Indicamos os autores que propuseram ou abordaram os algoritmos e listamos também as referências bibliográficas onde o leitor poderá obter detalhes dos algoritmos e as classes de problemas para as quais o algoritmo é indicado, seguindo a classificação de Dyckhoff. Embora a relação de algoritmos contida na Tabela 2.1 não seja exaustiva, acreditamos ser útil ao leitor interessado em aprofundar-se nas estratégias de resolução propostas para os problemas de corte e empacotamento.

Autores (algoritmo)	Ano	Referências	Classes
Algoritmos Exatos			
Herz	1972	[43]	2/B/O/M
Diegel (TRIMOPT)	1988	[22]	1/V/I/R
Martello e Toth (MTP)	1990	[59, 72]	1/V/I/M
Ashford e Daniel	1992	[2]	1/V/I/R
Vance *et al.*	1994	[79]	1/V/I/M
Arenales e Morábito	1995	[1]	2/V/I/R
Scheithauer e Terno	1995	[68]	1/V/I/R
Vanderbeck	1996	[80]	1/V/I/M
Scholl, Klein e Jürgens (BISON)	1996	[71]	1/V/I/M
Vance	1998	[78]	1/V/I/R
Algoritmos de Aproximação			
Johnson, Csirik e Simchi-Levi (FFD)	1973	[46, 19, 73]	1/V/I/M
Johnson *et al.* (NF e FF)	1974	[48, 31]	1/V/I/M
Coffman *et al.* (FFDH, NFDH e SF)	1980	[17]	2/V/O/M
Baker, Coffman e Rivest (BLDW)	1980	[5, 16]	2/V/O/M
Fernandez de la Vega e Lueker	1981	[29]	1/V/I/M
Baker, Brown e Katseff (UD)	1981	[4]	2/V/O/M
Karmakar e Karp	1982	[49]	1/V/I/M
Chung, Garey e Johnson (HFF)	1982	[13]	2/V/I/M
Coffman *et al.* (MFFD e BFB)	1984	[16, 73]	1/V/I/M
Lee e Lee (H_m)	1985	[52]	1/V/I/M
Li e Cheng (FF^2 e FFLSr, s)	1990	[53, 54]	2/V/I/M e 3/V/I/M
Schiermeyer e Steinberg (M,S)	1994	[70, 76]	2/V/O/M
Miyazawa e Wakabayashi ($\mathcal{A}_k$)	1994	[61, 62]	3/V/O/M
Csirik e Wöginger (Shelf($H_{m,p}$))	1997	[20]	2/V/O/M
Miyazawa	1997	[60]	2/V// e 3/V//
Métodos Heurísticos			
Gilmore e Gomory	1961	[37, 38, 39]	/V/I/R
Pierce (RPE-Technique)	1964	[66]	/V/I
Christofides e Whitlock	1977	[12]	2/B/I/R
Heicken e König	1980	[42]	/V/I/R
Wang	1983	[81]	2/B/I/R
Stadtler	1990	[74]	1/V/I/R
Oliveira e Ferreira	1990	[65]	2/B/I/R
Morábito e Arenales	1995	[63]	2/V/I/R
Wäscher e Gau	1996	[86, 35]	1/V/I/R
Falkenauer	1996	[28]	1/V/I/M
Bortfeldt e Gehring	1997	[10]	1/V/I/R

Tabela 2.1: Alguns algoritmos encontrados na literatura e sua aplicação.

Um Algoritmo Híbrido para o Problema de Corte Unidimensional

Neste capítulo abordamos o problema de corte unidimensional, também conhecido como problema de corte linear. Lembramos que este problema pertence à classe 1/V/I/R, segundo a classificação de Dyckhoff, e consiste em: dado um objeto, genericamente denominado de *barra*, de comprimento L, e uma lista de m itens, cada item i com comprimento l_i e demanda $d_i \in I\!N$ $(i = 1, \ldots, m)$, determinar o menor número de barras necessário para atender a demanda, ou seja, produzir d_i itens, cada qual de comprimento l_i. Obviamente também estamos interessados em determinar como as barras devem ser cortadas. Como já mencionamos, cada possível forma de cortar uma barra é chamada de *padrão de corte* (ou simplesmente *padrão*).

Na Figura 3.1 mostramos uma instância do problema de corte unidimensional onde a barra tem comprimento $L = 30$ e os itens têm comprimentos $l_1 = 5$, $l_2 = 12$, $l_3 = 7$ e demandas $d_1 = 4$, $d_2 = 2$, $d_3 = 2$. Indicamos também uma solução inteira para esta instância. Note que esta solução é ótima, pois não é possível atender a demanda usando apenas uma barra. A região pontilhada representa a parte desperdiçada nas barras.

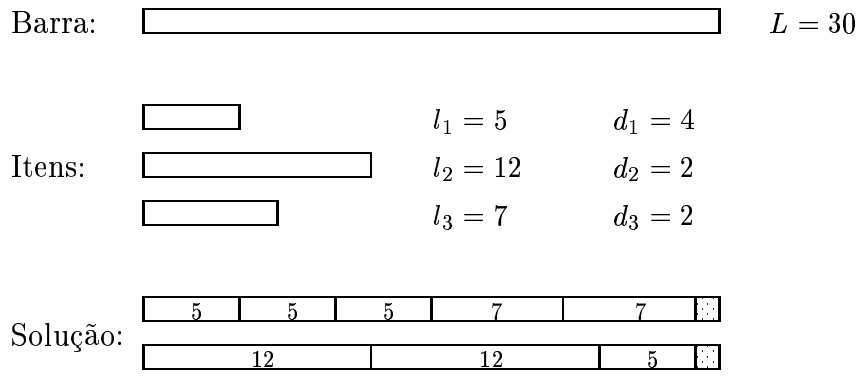

Figura 3.1: Instância e solução ótima de um problema de corte unidimensional.

Para tornar este capítulo auto-contido, apesar de já termos introduzido estas idéias no capítulo anterior, mencionamos novamente como este problema pode ser formulado como um problema de Programação Linear Inteira (PLI). Para fazer isso, representamos cada padrão j por um vetor-coluna a_j, cujo i-ésimo elemento indica o número de vezes que o item i ocorre nesse padrão. Assim, supondo que haja m itens, a_j é um vetor com m elementos.

No caso da Figura 3.1, a solução indicada na primeira barra corresponde a um padrão j onde $a_j = [3, 0, 2]$. Esse vetor a_j indica que o item 1 ocorre 3 vezes, o item 2 não ocorre nenhuma vez, e o item 3 ocorre 2 vezes no padrão j. Dizemos que um padrão é *viável* se $\sum_{i=1}^{m} (a_j)_i l_i \leq L$. O problema agora consiste em considerar os padrões viáveis e decidir quantas vezes cada padrão deve ser utilizado de modo a atender a demanda, minimizando o número total de barras utilizadas.

Para isso, introduzimos uma variável x cujos elementos são inteiros $x_j\,(j = 1, \ldots, n)$, que indicam quantas vezes o padrão j é selecionado. Note que se x é uma solução viável do problema acima, então o valor $\sum_{j=1}^{n} x_j$ corresponde ao número de barras a serem cortadas, já que cada padrão corresponde a uma barra. Assim, denotando por A a matriz $m \times n$ cujas colunas são os vetores $a_1, \ldots, a_n$, e representando por d o vetor das demandas, o problema pode ser assim formulado:

$$\begin{aligned} &\min \textstyle\sum_{j=1}^{n} x_j \\ &\text{sujeito a } Ax = d \\ &\qquad x_j \geq 0 \text{ e inteiro} \qquad j = 1, \ldots, n. \end{aligned} \tag{3.1}$$

A formulação acima traz consigo duas dificuldades em termos computacionais. A primeira é determinar a matriz A (que pode ter um número exponencial de colunas); a segunda é resolver um problema de programação linear inteira (que é sabido ser $\mathcal{NP}$-difícil). Para lidar com estas duas dificuldades, Gilmore e Gomory propuseram o método de geração de colunas [37, 38], delineado na seção seguinte.

3.1 Geração de Colunas

A idéia do método de geração de colunas consiste, como o próprio nome sugere, em ir gerando gradativamente as colunas da matriz A (dos padrões viáveis). Iniciamos com a matriz A correspondendo a uma solução básica viável. Com essa nova matriz A, que chamaremos de *base*, resolvemos a relaxação linear de (3.1):

$$\begin{aligned} &\min \textstyle\sum_{j=1}^{n} x_j \\ &\text{sujeito a } Ax = d \\ &\qquad x_j \geq 0 \qquad j = 1, \ldots, n. \end{aligned} \tag{3.2}$$

O método de geração de colunas consiste basicamente em, a partir de uma solução básica viável inicial, representada pela matriz A, usar o algoritmo *simplex revisado* [14], onde a cada iteração uma nova coluna é gerada resolvendo-se um problema da mochila. Este algoritmo é conhecido como *simplex revisado com geração de colunas*, que chamaremos simplesmente de SimplexGC. As soluções encontradas por este método convergem para uma solução ótima, não necessariamente inteira. Para contornar este problema, Gilmore e Gomory [37] propuseram arredondar para cima o valor das variáveis, após achar a solução ótima, obtendo assim uma solução inteira. No entanto, este procedimento pode acarretar a produção de itens em quantidade superior à demanda, e conduz a uma solução inteira eventualmente longe da ótima.

Para resolver (3.2), primeiramente precisamos determinar a matriz A. Felizmente, é fácil obter uma matriz A que permita calcular trivialmente uma solução básica viável para o problema de corte linear. Definimos, no passo 1, a matriz A (diagonal), de dimensão $m \times m$, fazendo $a_{ij} = \lfloor \frac{L}{l_i} \rfloor$ se $i = j$; e $a_{ij} = 0$, caso contrário ($i = 1, \ldots, m; j = 1, \ldots, m$). No passo 2, calculamos a solução x inicial fazendo $x_i = \frac{d_i}{a_{ii}}$ ($i = 1, \ldots, m$). Isto pode ser justificado pelo seguinte fato, fácil de ser provado.

Fato 3.1.1. *$x_i = \frac{d_i}{a_{ii}}$ ($i = 1, \ldots, m$) é solução básica viável de (3.2).*

Uma descrição mais formal e completa dos passos desse algoritmo é dada mais adiante. Cada iteração do método consiste em tentar gerar uma nova coluna que deve entrar na base. Se tal coluna não existir estamos então na solução ótima. Caso contrário, encontramos uma coluna que deve sair, substituindo-a pela nova coluna gerada. Para encontrar uma nova coluna que deve entrar na base precisamos primeiramente, no passo 3, resolver $yA = c$, onde c é o vetor-linha com entradas $c_i = 1$ ($i = 1, \ldots, m$). Usando a variável y, geramos, no passo 4, uma nova coluna resolvendo o seguinte problema da mochila:

$$\begin{aligned} &\max \textstyle\sum_{i=1}^{m} y_i z_i \\ &\text{sujeito a } \textstyle\sum_{i=1}^{m} l_i z_i \leq L \\ &\qquad z_i \geq 0 \text{ e inteiro} \qquad\qquad i = 1, \ldots, m\ . \end{aligned} \tag{3.3}$$

Para resolver este problema utilizamos o algoritmo MOCHILA que será descrito mais adiante. Note que se z é uma solução viável do problema (3.3), então z corresponde a um padrão viável.

Daqui por diante chamaremos de *valor* de uma solução o valor da função objetivo associado à essa solução. No passo 5, verificamos se o valor da solução ótima de (3.3) é menor ou igual a 1. Em caso afirmativo, a solução x corrente é uma solução ótima de (3.2). Nesse caso o algoritmo retorna x e pára. Caso contrário, o vetor z deve entrar na base. Para encontrar uma coluna que deve sair da base primeiramente calculamos, no passo 6, o vetor coluna b, tal que $Ab = z$. No passo seguinte calculamos $t = min\{\frac{x_i}{b_i} \mid b_i > 0 \ (i = 1, \ldots, m)\}$. Deve sair da base uma coluna a_s tal que:

$$s = min\{i \mid \frac{x_i}{b_i} = t \ (i = 1, \ldots, m)\}. \tag{3.4}$$

Calculamos s e, no passo 8, substituímos a_s por z, obtendo a nova base, e calculamos a nova solução corrente x fazendo $x_i = x_i - b_i t$, se $i \neq s$; e $x_i = t$, caso contrário $(i = 1, \ldots, m)$. No passo 9 a iteração é finalizada retornando ao passo 3. Na próxima subseção discutiremos um método para resolver (3.3).

3.1.1 Um Algoritmo para o Problema da Mochila

Apresentamos aqui um algoritmo bastante simples, mas satisfatório para os nossos propósitos, que resolve o problema da mochila (3.3). Nesse problema, dizemos que o coeficiente y_i é o *custo reduzido* do item i, e a razão $\frac{y_i}{l_i}$ é o *custo relativo* do item i. A *eficiência* da variável z_i é igual ao custo relativo do item i. Sem perda de generalidade, podemos assumir que as variáveis estão indexadas em ordem não crescente de eficiência, ou seja:

$$\frac{y_1}{l_1} \geq \ldots \geq \frac{y_m}{l_m}. \tag{3.5}$$

Toda solução ótima de (3.3) deve satisfazer

$$L - \sum_{i=1}^{m} l_i z_i < l_m, \tag{3.6}$$

pois se isso não ocorresse seria possível incrementar z_m de uma unidade. Chamamos as soluções viáveis que satisfazem (3.6) de *soluções sensíveis*. O algoritmo, que chamaremos de *MOCHILA* (descrito adiante), consiste basicamente em enumerar as soluções sensíveis, começando por aquelas que parecem ser mais promissoras, buscando entre elas uma solução ótima.

No passo 1 ordenamos decrescentemente os itens por seu custo relativo. A seguir calculamos, no passo 2, uma solução inicial fazendo $z_j = \lfloor (L - \sum_{i=1}^{j-1} l_i z_i)/l_j \rfloor$, para $j = 1, \ldots, m$. Note que esta solução inicial é composta do maior número possível dos itens de maior custo relativo. Usaremos z^* para denotar a melhor solução encontrada pelo algoritmo até o momento e $M = \sum_{i=1}^{m} y_i z_i^*$ para denotar o valor da solução z^*. Naturalmente, a esta altura do algoritmo $z^* = z$.

A cada iteração do algoritmo fazemos, no passo 3, $k = max\{i \mid z_i > 0 \ (i = m-1, \ldots, 1)\}$. Se não existe tal k então podemos parar, pois z^* é a solução ótima. Caso contrário, decrementamos z_k de uma unidade e calculamos $z_j = \lfloor (L - \sum_{i=1}^{j-1} l_i z_i)/l_j \rfloor$, para $j = k+1, \ldots, m$. No passo 4, comparamos a solução corrente z com a melhor solução encontrada: se $\sum_{i=1}^{m} y_i z_i > M$ então fazemos $z^* = z$. No passo 5 voltamos ao passo 3, recomeçando a iteração. Claramente o algoritmo converge para uma solução inteira ótima.

Podemos aperfeiçoar o algoritmo de modo a evitar que soluções claramente inferiores à melhor solução já encontrada sejam analisadas. Tal procedimento limita sobremaneira o espaço de busca, tornando o algoritmo sensivelmente mais rápido. Ao considerar uma possível nova solução $\bar{z}$, fazemos $\bar{z}_i = z_i$ para $i = 1, \ldots, k-1$ e $\bar{z}_k = z_k - 1$. A idéia é determinar *a priori*, quaisquer que sejam os valores de $\bar{z}_{k+1}, \ldots, \bar{z}_m$, se $\bar{z}$ tem chance de ser melhor que z^*. Por (3.5), podemos afirmar que cada variável $\bar{z}_{k+1}, \ldots, \bar{z}_m$ tem eficiência de no máximo $\frac{y_{k+1}}{l_{k+1}}$, portanto $\sum_{i=k+1}^{m} y_i \bar{z}_i \leq \frac{y_{k+1}}{l_{k+1}} \sum_{i=k+1}^{m} l_i \bar{z}_i$. Dessa forma, a restrição $\sum_{i=1}^{m} l_i \bar{z}_i \leq L$ implica em

$$\sum_{i=1}^{m} y_i \bar{z}_i \leq \sum_{i=1}^{k} y_i \bar{z}_i \ + \ \frac{y_{k+1}}{l_{k+1}} (L - \sum_{i=1}^{k} l_i \bar{z}_i). \tag{3.7}$$

A não ser que o lado direito de (3.7) seja estritamente maior que M, nenhuma possível solução $\bar{z}$ tem chance de melhorar z^*. Portanto, para evitar que soluções claramente inferiores sejam investigadas, basta fazer uma leve modificação no critério de escolha de k, alterando o passo 3.1 do algoritmo:

$$k = max\{i \mid z_i > 0 \ e \ \sum_{j=1}^{i} y_j \bar{z}_j \ + \ \frac{y_{i+1}}{l_{i+1}} (L - \sum_{j=1}^{i} l_j \bar{z}_j) > M \ \ (i = m-1, \ldots, 1)\}.$$

A condição suficiente para que a nova coluna possa entrar na base é $\sum_{i=1}^{m} y_i z_i > 1$. Dessa forma, não precisamos resolver o problema da mochila até a otimalidade: basta encontrar uma solução cujo valor seja maior que 1. Esta alteração permite resolver mais rapidamente o problema da mochila. Em nossa implementação, no entanto, escolhemos encontrar uma solução ótima do problema da mochila. De uma forma geral, este procedimento faz o método de geração de colunas convergir mais rapidamente, ou seja, é necessário gerar um número menor de colunas, o que compensa o tempo extra necessário para resolver o problema da mochila até a otimalidade.

Nos testes, não foi verificada uma diferença significativa no tempo total gasto pelos algoritmos híbridos (que veremos mais adiante) para resolver o problema de corte linear quando resolvemos o problema da mochila até a otimalidade ou não. Todavia, a maior motivação para resolver o problema da mochila até a otimalidade foi solucionar o problema da ciclagem, inerente ao *Simplex*, usando uma regra bem simples, explicada na próxima subseção.

Algoritmo MOCHILA

Entrada: $(L, l_1, \ldots, l_m, y_1, \ldots, y_m)$.
Saída: $z_1, \ldots, z_m \in I\!N$ tais que $\sum_{i=1}^{m} l_i z_i \leq L$ e $\sum_{i=1}^{m} y_i z_i$ é máximo.

1 Ordene os itens em ordem decrescente de custo relativo $(\frac{y_i}{l_i})$.
2 Faça $z_j = \lfloor (L - \sum_{i=1}^{j-1} l_i z_i)/l_j \rfloor$ para $j = 1, \ldots, m$, $z^* = z$ e $M = \sum_{i=1}^{m} y_i z_i^*$.
3 Faça $k = max\{i \mid z_i > 0 \ e \ \sum_{j=1}^{i} y_j \bar{z}_j \ + \ \frac{y_{i+1}}{l_{i+1}}(L - \sum_{j=1}^{i} l_j \bar{z}_j) > M \ (i = m-1, \ldots, 1)\}$.
3.1 Se não existe tal k então devolva z^* e pare.
3.2 Caso contrário faça $z_k = z_k - 1$ e $z_j = \lfloor (L - \sum_{i=1}^{j-1} l_i z_i)/l_j \rfloor$, para $j = k+1, \ldots, m$.
4 Se $M \leq \sum_{i=1}^{m} y_i z_i$ faça $z^* = z$ e $M = \sum_{i=1}^{m} y_i z_i^*$.
5 Retorne ao passo 3.

3.1.2 A Regra de Bland

Em 1972, Klee e Minty [50] exibiram uma família de problemas de programação linear, criada artificialmente a partir de deformações do cubo n-dimensional, para a qual o algoritmo simplex, com regra de pivotação de Dantzig-Wolfe, não converge. De fato, a não convergência do algoritmo simplex, que está associada ao fenômeno da *degenerescência*, já havia sido constatada anteriormente por Hoffman e Beale [7]. Para garantir a convergência do método de geração de colunas, introduzimos a regra descoberta em 1977 por Robert Bland [9], que evita ciclagem.

Tal regra consiste em: (1) dentre todas as colunas que podem entrar na base escolher a de *menor índice*; e (2) dentre todas as colunas que podem sair da base escolher a de *menor índice*. Para implementar a regra de Bland precisamos resolver o problema da mochila até a otimalidade e no caso de haver várias soluções ótimas para o problema da mochila, escolhemos a primeira delas. Este procedimento já está implícito no algoritmo MOCHILA, descrito anteriormente. De forma análoga, existindo várias colunas que podem sair da base, escolhemos a de menor índice, regra que já está expressa em (3.4). Apesar da regra de Bland ser extremamente simples, e fácil de implementar, sua prova não é trivial, pelo que deixamos de transcrevê-la neste texto. Apresentamos a seguir o algoritmo SimplexGC.

Algoritmo SimplexGC

Entrada: $(L, l_1, \ldots, l_m, d_1, \ldots, d_m)$.
Saída: Uma solução ótima de: $\min \sum_{j=1}^{n} x_j$
sujeito a $Ax = d$
$x_j \geq 0 \qquad j = 1, \ldots, n,$
onde cada coluna j de A é tal que $\sum_{i=1}^{m} a_{ij} l_i \leq L$.

1 Para $i = 1$ até m faça.
1.1 Para $j = 1$ até m se $i = j$ então faça $a_{ij} = \lfloor \frac{L}{l_i} \rfloor$; caso contrário, faça $a_{ij} = 0$.
2 Faça $x_i = \frac{d_i}{a_{ii}}$ $(i = 1, \ldots, m)$.
3 Resolva $yA = c$.
4 Execute o algoritmo MOCHILA com parâmetros $L, l_1, \ldots, l_m, y_1, \ldots, y_m$.
5 Se $\sum_{i=1}^{m} y_i z_i \leq 1$, retorne x e pare.
6 Caso contrário, resolva $Ab = z$.
7 Calcule $t = min\{\frac{x_i}{b_i} \mid b_i > 0 \ (i = 1, \ldots, m)\}$ e $s = min\{i \mid \frac{x_i}{b_i} = t \ (i = 1, \ldots, m)\}$.
8 Para $i = 1$ até m faça $a_{is} = z_i$ $(i = 1, \ldots, m)$.
8.1 Se $i = s$ então faça $x_i = t$; caso contrário, faça $x_i = x_i - b_i t$.
9 Retorne ao passo 3.

Nesta seção delineamos como resolver a relaxação linear de (3.1) usando o método de geração de colunas, obtendo assim uma solução ótima, possivelmente fracionária. Na seção seguinte descrevemos como obter uma solução inteira sem que haja excesso de produção e que se aproxime bem mais da solução inteira ótima do que se apenas arredondarmos a solução fracionária para cima.

3.2 Obtendo uma Solução Inteira

Aplicando o método de geração de colunas obtemos uma solução ótima x para a relaxação linear de (3.1). Chamemos de v_{rl} o valor desta solução. O método proposto para encontrar uma solução inteira de (3.1) consiste em arredondar x para baixo obtendo $\bar{x}$, ou seja, $\bar{x}_j = \lfloor x_j \rfloor$ $(j = 1, \ldots, m)$. Seja $\bar{v}$ o valor da solução $\bar{x}$. Claramente $\bar{v} \leq v_{rl}$. Se $\bar{v} = v_{rl}$, então x é uma solução inteira ótima, caso contrário, uma parte da demanda não foi atendida, pois $\bar{x}_j < x_j$ para algum $j \in \{1, \ldots, m\}$ e portanto $\sum_{j=1}^{m} a_{ij} \bar{x}_j < d_i$ para tal j. Temos assim um problema residual onde cada item i possui demanda *inteira* $d'_i = d_i - \sum_{j=1}^{m} a_{ij} \bar{x}_j$ $(i = 1, \ldots, m)$.

Após calcular $\bar{x}$ e d', se $\bar{v} > 0$ então aplicamos recursivamente o algoritmo ao problema residual; caso contrário, resolvemos o problema residual utilizando um algoritmo que forneça

uma solução inteira, como o *First Fit Decreasing (FFD)*, o *First Fit Decreasing especializado (FFDe)* e o *FFDWB*, abordados nas próximas subseções.

3.2.1 O Algoritmo FFD

Escolhemos utilizar o FFD pois este algoritmo é bem simples, e além disso, apresenta um desempenho bastante satisfatório. Em 1973, Johnson [46] provou que o algoritmo FFD tem limite de desempenho assintótico $\frac{11}{9}$. Mais precisamente, denotando por $FFD(I)$ o valor da solução encontrada pelo algoritmo FFD e por $OPT(I)$ o valor de uma solução ótima para uma instância I, vale o seguinte resultado.

Teorema 3.2.1. *Para toda instância I, $FFD(I) \leq \frac{11}{9}OPT(I) + 4$.*

Em 1991 Yue [87] mostrou que a constante aditiva do teorema acima pode ser trocada por 1. Ademais, o FFD apresenta desempenho empírico no caso médio de 1,02 (*cf.* Bramel *et al.* [11]) e pode ser implementado de forma a ter complexidade de tempo $\mathcal{O}(n \log n)$. Descrevemos a seguir o algoritmo FFD, adotando as seguintes notações: $\wp_i$ representa o padrão de corte i e $c(\wp_i)$ é uma função que retorna o comprimento da barra não utilizado pelo padrão $\wp_i$.

A entrada do algoritmo consiste do comprimento L de uma barra e de uma lista S de n itens de comprimentos $l_1, \ldots, l_n$. Neste caso, estamos supondo que a entrada correspondente ao problema residual, que consiste de m itens de comprimento l_i $(i = 1, \ldots, m)$ e demanda d_i, é transformada numa entrada para o algoritmo FFD criando-se uma lista S onde cada item de comprimento l_i aparece d_i vezes.

Algoritmo FFD

Entrada: $(L, l_1, \ldots, l_n)$.
Saída: Uma solução inteira $\wp_1, \ldots, \wp_k$ que atende a demanda.

1 Ordene $l_1, \ldots, l_n$ em ordem decrescente e faça $k = 1$ e $\wp_k = \emptyset$.
2 Para $i = 1$ até n procure $j = min\{h \mid c(\wp_h) \geq l_i \ (h = 1, \ldots, k)\}$
  2.1 Se existir tal j então empacote o item i no padrão $\wp_j$, ou seja, faça $\wp_j = \wp_j \cup \{i\}$.
  2.2 Caso contrário, faça $k = k + 1$ e empacote o item i em $\wp_k$ fazendo $\wp_k = \{i\}$.
3 Retorne $\wp_1, \ldots, \wp_k$ e pare.

Primeiramente, ordenamos os itens em ordem decrescente de comprimento. Iniciamos então o processo empacotando o item 1 no padrão $\wp_1$ e fazendo $k = 1$. Suponha que o item i é o próximo item a ser empacotado. Procuramos dentre os padrões $\wp_1, \ldots, \wp_k$ aquele de menor índice onde é possível empacotar o item i. Caso não exista tal padrão, iniciamos a geração de

um novo padrão, ou seja, fazemos $k = k+1$ e empacotamos o item i no padrão $\wp_k$. Este processo é repetido até que todos os itens tenham sido empacotados, após o que retornamos a solução $\wp_1, \ldots, \wp_k$.

3.2.2 O Algoritmo FFD especializado (FFDe)

Apresentaremos a seguir uma variação do algoritmo FFD, que chamaremos de *FFD especializado*, denotado por FFDe, especializado para o problema que queremos resolver. Tal versão é especialmente apropriada por requerer menor tempo de execução e por permitir que a implementação do algoritmo seja feita utilizando-se estruturas de dados mais simples.

Seja S uma lista de m itens de comprimentos $l_1, \ldots, l_m$ e demandas $d_1, \ldots, d_m$. Ordene, no passo 1, os elementos de S em ordem decrescente de comprimento e faça $k = 1$. No passo seguinte, repetimos o procedimento descrito no próximo parágrafo até que todos os itens sejam empacotados, ou seja, até obtermos $d_i = 0$ $(i = 1, \ldots, m)$, depois do que o algoritmo retorna $\wp_1, \ldots, \wp_k$ e $r_1, \ldots, r_k$.

Procuramos o item de menor índice que caiba em $\wp_k$. Se existir tal item, empacotamos este item no padrão $\wp_k$ o maior número de vezes possível, limitado à demanda do item, e atualizamos a demanda. Caso contrário, determinamos r_k, que é o número de vezes que o padrão $\wp_k$ deve ser utilizado, fazendo $r_k = min(\lfloor \frac{d_i}{f_i(\wp_k)} \rfloor, i \in \wp_k)$, onde $f_i(\wp_k)$ é uma função que retorna a frequência do item i em $\wp_k$, atualizamos as demandas fazendo $d_i = d_i - f_i(\wp_k)r_k$ para todo item i contido em $\wp_k$ e fazemos $k = k + 1$.

Na implementação deste algoritmo, uma preocupação pode ser a quantidade de memória necessária para armazenar a solução. Podemos, no entanto, afirmar que o número de padrões distintos necessários para cortar uma lista S que possui m itens de comprimentos distintos é linear em relação a m, como mostra o seguinte resultado.

Teorema 3.2.2. *Seja S uma lista que possui m itens de comprimentos distintos. O algoritmo FFDe encontra uma solução constituída de no máximo $2m$ padrões distintos.*

Prova. Por indução em m. Claramente o teorema vale para $m = 1$. Seja $c = \lfloor \frac{L}{l_1} \rfloor$. Se $c \geq d_1$ a solução encontrada pelo FFDe é constituída de apenas um padrão onde o item 1 aparece d_1 vezes. Caso contrário, a solução encontrada pelo FFDe utilizará $\lfloor \frac{d_1}{c} \rfloor$ vezes o padrão $\wp_1$, no qual o item 1 aparece c vezes. Seja $r = d_1 - \lfloor \frac{d_1}{c} \rfloor \cdot c$. Se $r > 0$ então será usado mais um padrão, no qual o item 1 aparecerá r vezes.

Considere uma lista com $m \geq 2$ itens. Retirando desta lista o item de menor comprimento, temos uma lista com $m - 1$ itens, e pela hipótese de indução, o FFDe aplicado a essa lista encontra uma solução com no máximo $2m - 2$ padrões distintos. A demanda do item de menor comprimento pode, eventualmente, ser atendida utilizando-se o espaço restante nos padrões já

gerados. Se isso não for possível, esta demanda pode ser atendida usando-se no máximo mais 2 padrões, como vimos no parágrafo anterior. □

Algoritmo FFD*e*

Entrada: $(L, l_1, \ldots, l_m, d_1, \ldots, d_m)$.
Saída: Uma solução inteira que atende a demanda.

1 Ordene $l_1, \ldots, l_m$ em ordem decrescente e faça $k = 1$ e $\wp_k = \emptyset$.
2 Repita até que $d_i = 0$ $(i = 1, \ldots, m)$.
  2.1 Procure $j = min\{h \mid l_h \leq c(\wp_k)\ (h = 1, \ldots, m)\}$.
  2.2 Se existir tal j então faça $f = min(d_j, \lfloor \frac{c(\wp_k)}{l_j} \rfloor)$ e empacote f vezes o item j em $\wp_k$, fazendo f vezes $\wp_j = \wp_j \cup \{i\}$.
  2.3 Caso contrário, faça $r_k = min\{\lfloor \frac{d_i}{f_i(\wp_k)} \rfloor,\ i \in \wp_k\}$.
    2.3.1 Para todo $i \in \wp_k$ faça $d_i = d_i - f_i(\wp_k) r_k$. Faça $k = k + 1$ e $\wp_k = \emptyset$.
3 Retorne $\wp_1, \ldots, \wp_k$ e $r_1, \ldots, r_k$ e pare.

Note que, dado um item i, o algoritmo FFD procura imediatamente determinar em qual padrão este item deve ser cortado. Já no algoritmo FFD*e*, dado um padrão k, procuramos qual o próximo item que deve ser empacotado nesse padrão. Chamamos o algoritmo FFD de *item-orientado* e o algoritmo FFD*e* de *padrão-orientado*. Esta característica, ser um algoritmo padrão-orientado, permite adaptar, sem maiores dificuldades, o FFD*e* obtendo o algoritmo FFDWB, que será detalhado na próxima subseção. Apesar deste enfoque diferenciado entre o FFD e o FFD*e*, vale o seguinte resultado.

Teorema 3.2.3. *Dada uma lista S de itens, a solução do algoritmo FFD é igual à solução do algoritmo FFD*e*.*

Prova. Por indução em n, o número de itens da lista S. Para $n = 1$, claramente a solução encontrada pelo FFD é igual à solução encontrada pelo FFD*e*. Considere agora uma lista S com $n \geq 2$ itens ordenados, sem perda de generalidade, em ordem decrescente. Seja $\wp_1, \ldots, \wp_k$ a solução encontrada pelo FFD aplicado à lista S e $\wp_j$ o padrão onde o item n foi empacotado.

Seja S' a lista obtida de S após retirar o item n (que possui o menor comprimento). Claramente a solução encontrada pelo FFD aplicado à lista S' é $\wp_1, \ldots, \wp_j - \{n\}, \ldots, \wp_k$. Pela hipótese de indução, essa solução é igual à solução encontrada pelo FFD*e* aplicado à S'. Note que o item n cabe em $\wp_j$ e não cabe nos padrões de menor índice.

Uma análise cuidadosa nos permite concluir que ao aplicar o algoritmo FFD*e* à lista S, o item n também será empacotado em $\wp_j$ pois ao construir os padrões $\wp_1, \ldots, \wp_{j-1}$, o item n não será

escolhido (por ser de maior índice). Portanto a solução obtida pelo algoritmo FFD*e* aplicado a S também será $\wp_1, \ldots, \wp_k$. □

O seguinte corolário é resultado direto dos teoremas 3.2.2 e 3.2.3.

Corolário 3.2.4. *Seja S uma lista que possui m itens de comprimentos distintos. O algoritmo FFD encontra uma solução constituída de no máximo 2m padrões distintos.*

Apesar de ser empiricamente bom no caso médio, Simchi-Levi [73] mostrou que o limite de desempenho absoluto do FFD é 1,5 no pior caso; e que este limite é justo a não ser que $\mathcal{P} = \mathcal{NP}$. Foi também observado que o FFD encontra dificuldade, no que diz respeito à qualidade da solução, em instâncias pequenas ou onde os itens têm comprimento de aproximadamente $\frac{1}{3}L, \frac{1}{4}L, \frac{1}{5}L, \ldots$, onde L é o tamanho da barra (*cf.* Schwerin e Wäscher [72]) ou ainda nas instâncias que Falkenauer [28] chamou de *triplets*. Apresentamos a seguir o algoritmo FFDWB, baseado no algoritmo FFD*e*, que encontra uma solução inteira *ótima*.

3.2.3 O Algoritmo FFDWB

Na Seção 3.1 vimos que o método de geração de colunas fornece uma solução ótima para a relaxação linear de (3.1) cujo valor chamaremos de v_{rl}. Podemos então usar $v_{ip} = \lceil v_{rl} \rceil$ como um limite inferior para o valor de qualquer solução inteira ótima. Como veremos na Seção 3.4, este limitante é justo na grande maioria dos problemas.

A idéia básica do algoritmo *First Fit Decreasing with Backtracking (FFDWB)*, que propomos, é aplicar um procedimento semelhante ao algoritmo FFD*e* para gerar um padrão de cada vez. O desperdício deste padrão é então comparado com o desperdício da solução cujo valor é v_{ip}, da forma que explicitaremos mais à frente. Se o padrão gerado não for promissor executamos um retrocesso, diminuindo a frequência dos itens dentro do padrão, do último item que foi cortado até o primeiro, e aplicamos recursivamente o procedimento semelhante ao FFD*e* na parte da barra não utilizada pelo padrão. Na solução encontrada pelo FFDWB, cada padrão é utilizado uma vez, de modo que a cada padrão corresponde uma barra.

Para facilitar a compreensão do algoritmo FFDWB, detalhamos primeiramente o procedimento PACKING. Este procedimento recebe como parâmetros de entrada um valor D_k, que representa o comprimento que pode ser desperdiçado numa solução que utiliza C barras depois de gerados os padrões $\wp_1, \ldots, \wp_{k-1}$; um valor k, representando o índice do padrão corrente; e ainda os valores f e i, onde f representa quantas vezes o item de comprimento l_i deve ser empacotado no padrão corrente. No procedimento assumimos como constantes globais o valor C, que representa a quantidade máxima de barras que podem ser utilizadas na solução, os comprimentos $l_1, \ldots, l_m$ dos itens e suas demandas $d_1, \ldots, d_m$.

No passo 1, empacotamos f vezes o item i em $\wp_k$ e atualizamos a demanda do item i fazendo $d_i = d_i - f$. Procuramos então, no passo 2, o item j de menor índice cuja demanda é maior do que zero. Se não existir tal j significa que a demanda de todos os itens foi atendida. Nesse caso a solução $\wp_1, \ldots, \wp_k$ é retornada e o procedimento pára. Caso contrário, buscamos determinar, no passo 3, o item i' de menor índice que possui demanda maior que zero e que ainda cabe em $\wp_k$.

Se existir tal item i', calculamos, no passo 3.1, $f = min(d_{i'}, \lfloor \frac{c(\wp_k)}{l_{i'}} \rfloor)$. Desviamos então o fluxo de execução para o passo 5 e buscamos uma solução fazendo chamadas recursivas ao procedimento PACKING, com f' variando de f até 0, tendo como parâmetros D_k, k, f' e i'. Essas chamadas recursivas podem levar a uma solução, caso em que o procedimento pára. Caso contrário, após cada chamada recursiva infrutífera atualizamos a demanda fazendo $d_{i'} = d_{i'} + f'$ e removemos de $\wp_k$ as f' ocorrências do item i'.

Se não existir tal item i', significa que nenhum item pode ser empacotado no espaço restante em $\wp_k$. Neste caso algumas condições devem ser satisfeitas antes que o padrão $\wp_k$ seja aceito. É preciso verificar se ainda é possível utilizar outra barra, ou seja, se $k < C$ e se o desperdício em $\wp_k$ é tal que, possivelmente, vai conduzir a uma solução que utilize mais do que C barras. Se a desigualdade

$$c(\wp_k) \leq \lfloor \frac{D_k}{C-k+1} \rfloor \tag{3.8}$$

não for satisfeita, rejeitamos o padrão. Aqui abrimos um parênteses para explicar a restrição (3.8) e mostrar que ela não nos impede de encontrar uma solução. Para isto utilizaremos o seguinte lema, fácil de ser verificado.

Lema 3.2.5. *Se existir uma solução que utilize C barras, pelo menos um padrão contido nesta solução apresenta desperdício de no máximo $\lfloor \frac{D}{C} \rfloor$, onde $D = CL - \sum_{i=1}^{m} l_i d_i$.*

O parâmetro D_k representa o quanto ainda pode ser desperdiçado e $C - k + 1$ o número de padrões que ainda podem ser gerados, visto que já geramos $k - 1$ padrões. Pelo Lema 3.2.5, se existir uma solução (parcial) que utilize $C - k + 1$ padrões, existirá também um padrão cujo desperdício não excede $\lfloor \frac{D_k}{C-k+1} \rfloor$. O procedimento escolhe construir uma solução utilizando os padrões que satisfazem (3.8).

Podemos impor mais uma restrição antes de aceitar um padrão. No procedimento PACKING, ao terminar de gerar o padrão $\wp_k$, o desperdício máximo do padrão $\wp_{k+1}$ é dado por $\lfloor \frac{D_k - c(\wp_k)}{C-k} \rfloor$. Dessa forma, resolvemos um problema da mochila onde os itens têm comprimento l_i, custo reduzido também l_i e demanda d_i $(i = 1, \ldots, m)$, de modo a descobrir se existe um padrão cujo desperdício seja no máximo $\lfloor \frac{D_k - c(\wp_k)}{C-k} \rfloor$. Caso não exista tal padrão, rejeitamos $\wp_k$.

No algoritmo FFDWB, cada padrão aceito é utilizado *uma vez*, portanto a frequência de cada item i contido no padrão não pode exceder d_i. Note ainda que o custo reduzido de cada item é um

número inteiro. Essas duas peculiaridades ensejam modificações no algoritmo MOCHILA, proposto na Subseção 3.1.1, modificações essas que limitam o espaço de busca, permitindo resolver o problema da mochila mais rapidamente. No passo 2 e no passo 3.2 do algoritmo MOCHILA, calculamos z_j fazendo $z_j = min(d_j, \lfloor (L - \sum_{i=1}^{j-1} l_i z_i)/l_j \rfloor)$. Com isto fazemos com que a frequência de cada item no padrão não exceda sua demanda. Além disso, como o custo reduzido y_i de cada item i é inteiro, podemos substituir a inequação $\sum_{j=1}^{i} y_j \bar{z}_j + \frac{y_{i+1}}{l_{i+1}}(L - \sum_{j=1}^{i} l_j \bar{z}_j) > M$, no passo 3, pela inequação mais forte $\sum_{j=1}^{i} y_j \bar{z}_j + \frac{y_{i+1}}{l_{i+1}}(L - \sum_{j=1}^{i} l_j \bar{z}_j) \geq M + 1$. A seguir descrevemos o algoritmo *MOCHILA especializado* (denotado por MOCHILAe).

Algoritmo MOCHILAe

Entrada: $(L, l_1, \ldots, l_m, y_1, \ldots, y_m, d_1, \ldots, d_m)$.
Saída: $z_1, \ldots, z_m \in \mathbb{N}$ tais que $\sum_{i=1}^{m} l_i z_i \leq L$ e $\sum_{i=1}^{m} y_i z_i$ é máximo.

1 Ordene os itens em ordem decrescente de custo relativo $(\frac{y_i}{l_i})$.
2 Faça $z_j = min(d_j, \lfloor (L - \sum_{i=1}^{j-1} l_i z_i)/l_j \rfloor)$ para $j = 1, \ldots, m$, $z^* = z$ e $M = \sum_{i=1}^{m} y_i z_i^*$.
3 Faça $k = max\{i \mid z_i > 0 \ e \ \sum_{j=1}^{i} y_j \bar{z}_j + \frac{y_{i+1}}{l_{i+1}}(L - \sum_{j=1}^{i} l_j \bar{z}_j) \geq M + 1 \ (i = m-1, \ldots, 1)\}$.
3.1 Se não existe tal k então devolva z^* e pare.
3.2 Caso contrário, faça $z_k = z_k - 1$ e $z_j = min(d_j, \lfloor (L - \sum_{i=1}^{j-1} l_i z_i)/l_j \rfloor)$ para $j = k+1, \ldots, m$.
4 Se $M \leq \sum_{i=1}^{m} y_i z_i$ faça $z^* = z$ e $M = \sum_{i=1}^{m} y_i z_i^*$.
5 Retorne ao passo 3.

Voltamos a descrever o procedimento PACKING. Seja $t = L - \sum_{i=1}^{m} l_i z_i$ o comprimento desperdiçado na solução encontrada pelo algoritmo MOCHILAe. Qualquer padrão $\wp_k$ gerado no procedimento PACKING tem que possuir desperdício $c(\wp_k)$ tal que

$$t \leq \lfloor \tfrac{D_k - c(\wp_k)}{C-k} \rfloor. \tag{3.9}$$

No passo 3.2 executamos o algoritmo MOCHILAe com os parâmetros $L, l_1, \ldots, l_m, l_1, \ldots, l_m, d_1, \ldots, d_m$. No passo seguinte fazemos $t = L - \sum_{i=1}^{m} l_i z_i$. Podemos então verificar, no passo 3.4, se o padrão $\wp_k$ deve ser aceito ou rejeitado. Se $k < C$ e $c(\wp_k) \leq \lfloor \frac{D_k}{C-k+1} \rfloor$ e $t \leq \lfloor \frac{D_k - c(\wp_k)}{C-k} \rfloor$ o padrão $\wp_k$ deve ser aceito. Neste caso é feita uma chamada recursiva ao procedimento PACKING com os parâmetros $D_k - c(\wp_k), k+1, min(d_j, \lfloor \frac{L}{l_j} \rfloor)$ e j. Note que ao incrementarmos k, iniciamos a geração de um novo padrão, dessa forma o item j (que é o de menor índice com demanda maior que zero) deve ser empacotado $min(d_j, \lfloor \frac{L}{l_j} \rfloor)$ vezes neste novo padrão.

Caso a recursão contida no passo 3.4 não leve a uma solução, a chamada ao procedimento foi infrutífera. Devemos portanto executar um retrocesso. Para isso, no passo 4, fazemos $f = -1$ de forma que o laço contido no passo 5 não seja executado.

Procedimento PACKING(D_k, k, f, i)

1 Empacote f vezes o item i em $\wp_k$, fazendo f vezes $\wp_k = \wp_k \cup \{i\}$, e faça $d_i = d_i - f$.
2 Procure $j = min\{h \mid d_h > 0 \ (h = 1, \ldots, m)\}$. Se não existir tal j retorne $\wp_1, \ldots, \wp_k$ e pare.
3 Procure $i' = min\{h \mid d_h > 0 \ e \ l_h \leq c(\wp_k) \ (h = i+1, \ldots, m)\}$.
  3.1 Se existir i' então faça $f = min(d_{i'}, \lfloor \frac{c(\wp_k)}{l_{i'}} \rfloor)$ e vá para o passo 5.
  3.2 Execute o algoritmo MOCHILAe com parâmetros $L, l_1, \ldots, l_m, l_1, \ldots, l_m, d_1, \ldots, d_m$.
  3.3 Faça $t = L - \sum_{i=1}^{m} l_i z_i$.
  3.4 Se $k < C$ e $c(\wp_k) \leq \lfloor \frac{D_k}{C-k+1} \rfloor$ e $t \leq \lfloor \frac{D_k - c(\wp_k)}{C-k} \rfloor$ então execute
    PACKING$(D_k - c(\wp_k), k+1, min(d_j, \lfloor \frac{L}{l_j} \rfloor), j)$.
4 Faça $f = -1$. {Para que o laço no passo seguinte não seja efetuado}
5 Para $f' = f$ até 0 execute PACKING(D_k, k, f', i'), faça $d_{i'} = d_{i'} + f'$ e remova de $\wp_k$ as f' ocorrências do item i', fazendo f vezes $\wp_k = \wp_k - \{i'\}$.

O procedimento PACKING com parâmetros D_k, k, f e i, encontra, se existir, uma solução que utiliza no máximo $C - k + 1$ barras onde a primeira barra tem comprimento $c(\wp_k)$ e as demais barras têm comprimento L. Uma vez compreendido o procedimento PACKING fica fácil entender o algoritmo FFDWB, que descrevemos a seguir.

Inicialmente colocamos os itens em ordem crescente usando como chave de ordenação, para cada item i, o número $min(d_i, \lfloor \frac{L}{l_i} \rfloor)$. Tal número representa o número máximo de vezes que o item i pode aparecer num padrão. Nos testes realizados constatamos ser este critério de ordenação usualmente mais vantajoso do que utilizar, por exemplo, ordem decrescente de comprimento, o que é feito no algoritmo FFDe.

Não está bem claro para nós por que esta heurística apresenta resultados melhores do que colocar os itens em ordem decrescente de comprimento, mas parece-nos que a explicação deste fenômeno está associada à idéia de limitar o número de ramificações nos primeiros níveis da árvore de busca. O algoritmo FFDWB consegue podar a árvore com maior frequência nos níveis de maior profundidade, portanto parece ser boa idéia construir a árvore de modo que os primeiros níveis possuam poucas ramificações e os níveis posteriores, que eventualmente não serão percorridos, possuam mais ramificações.

No passo 2 do algoritmo é calculado o desperdício D de uma solução que utiliza C barras fazendo-se $D = CL - \sum_{i=1}^{m} l_i d_i$, e é determinado f, que indica o número máximo de vezes que o item 1 pode aparecer no primeiro padrão. No passo seguinte o algoritmo busca uma solução fazendo chamadas ao procedimento PACKING com f' variando de f até 0, tendo como parâmetros $D, 1, f'$ e 1. Se uma solução for encontrada, o algoritmo pára no passo 2 do procedimento PACKING; caso contrário, no passo 4, o algoritmo retorna 0, significando que não existe solução que utilize no máximo C barras.

Basicamente, o algoritmo FFDWB enumera todos os padrões viáveis, buscando uma coleção de padrões cuja cardinalidade seja no máximo C e que atenda toda a demanda. Podemos dizer que o algoritmo executa uma busca em profundidade numa árvore, onde cada nó da árvore representa um padrão, e a solução encontrada é um ramo da árvore, desde a raiz até uma folha. Esta árvore tem altura no máximo C. Utilizando as desigualdades (3.8) e (3.9), podamos a árvore, diminuindo bastante o espaço de busca. Note que, ao percorrer um ramo da árvore com profundidade k, temos que $D_1 \leq \ldots \leq D_k$ (não necessariamente $c(\wp_1) \leq \ldots \leq c(\wp_k)$). Isto significa que o FFDWB prefere gerar inicialmente os padrões que apresentam pequeno desperdício; com isso, os primeiros níveis da árvore possuem poucas ramificações, quando comparados com os níveis posteriores.

Algoritmo FFDWB

Entrada: $(L, l_1, \ldots, l_m, d_1, \ldots, d_m, C)$.
Saída: Uma solução inteira de valor no máximo C, se existir; caso contrário, o valor 0.

1 Coloque os itens em ordem crescente de $min(d_i, \lfloor \frac{L}{l_i} \rfloor)$ $(i = 1, \ldots, m)$.
2 Faça $D = CL - \sum_{i=1}^{m} l_i d_i$ e $f = min(d_1, \lfloor \frac{L}{l_1} \rfloor)$.
3 Para $f' = f$ até 0 execute PACKING$(D, 1, f', 1)$.
4 Retorne 0 e pare.

O seguinte teorema garante a corretude do algoritmo FFDWB.

Teorema 3.2.6. *O algoritmo FFDWB encontra, se existir, um empacotamento que utiliza no máximo C barras.*

Prova. Seja $\wp_1, \ldots, \wp_C$ um empacotamento que utiliza C barras, ordenadas, sem perda de generalidade, em ordem crescente de desperdício. Seja $D_k = CL - \sum_{i=1}^{m} l_i d_i - \sum_{i=1}^{k-1} c(\wp_i)$. Claramente $c(\wp_k) \leq \lfloor \frac{D_k}{C-k+1} \rfloor$ e $c(\wp_j) \leq \lfloor \frac{D_k - c(\wp_k)}{C-k} \rfloor$ para algum j $(k < j \leq C)$, de outra forma a demanda não poderia ter sido totalmente atendida pelas C barras. Portanto nenhum dos padrões $\wp_1, \ldots, \wp_C$ seria descartado no passo 3.4 do procedimento PACKING. Resta apenas verificar se o procedimento PACKING seria capaz de gerar estes padrões.

Note que cada item, digamos i, aparece em $\wp_1$ no máximo $min(d_i, \lfloor \frac{L - \sum_{j=1}^{i-1} l_j f_j(\wp_1)}{l_i} \rfloor)$ vezes. Conforme o passo 3 do algoritmo FFDWB, a frequência do item 1 irá variar de $min(d_1, \lfloor \frac{L}{l_1} \rfloor)$ até $f_i(\wp_1)$. Conforme o passo 3.1 do procedimento PACKING, a frequência do item 2 irá variar de $min(d_2, \lfloor \frac{L - l_1 f_1(\wp_1)}{l_1} \rfloor)$ até $f_2(\wp_1)$, e assim sucessivamente até que o algoritmo tenha gerado $\wp_1$. De forma análoga construímos $\wp_2, \ldots, \wp_C$, completando a demonstração □

A Figura 3.2 mostra a árvore percorrida pelo algoritmo FFDWB ao resolver a instância

caracterizada por $L = 20$, $l = \{10, 8, 7, 6, 5, 4\}$, $d = \{3, 1, 1, 4, 1, 1\}$ e $C = 4$. Cada nó da árvore representa um padrão. Observe que os primeiros dois ramos da árvore (olhando de cima para baixo) foram podados no segundo nível, pois os dois primeiros padrões gerados neste nível apresentam desperdício de 2 e 3, enquanto que o passo 3.4 do procedimento PACKING impõe que o segundo padrão possua desperdício zero ($c(\wp_2) \leq \lfloor \frac{D_2}{C-2+1} \rfloor = 0$). A solução encontrada pelo FFDWB é representada pelo terceiro ramo da árvore, e utiliza 4 barras. É interessante notar que o algoritmo FFD aplicado a esta mesma instância encontra uma solução que utiliza 5 barras.

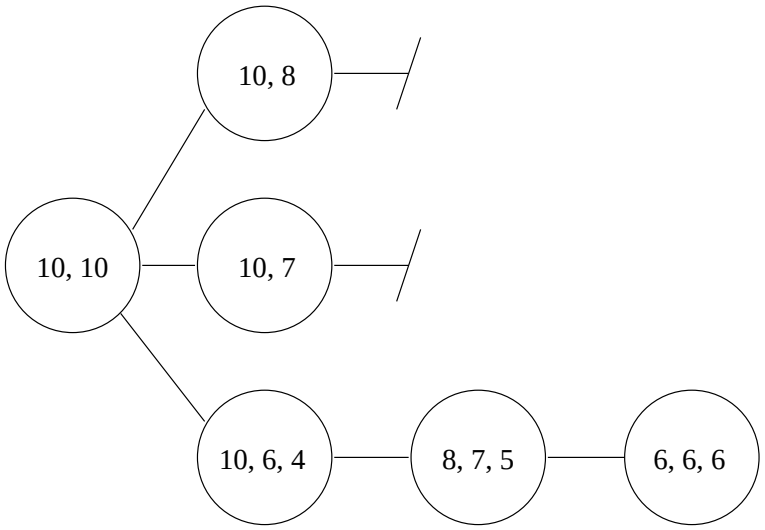

Figura 3.2: Árvore percorrida pelo algoritmo FFDWB ao resolver a instância caracterizada por $L = 20$, $l = \{10, 8, 7, 6, 5, 4\}$, $d = \{3, 1, 1, 4, 1, 1\}$ e $C = 4$.

3.3 O Algoritmo HÍBRIDO

Estamos agora em condições de descrever o algoritmo híbrido que desenvolvemos, doravante referido como HÍBRIDO. No passo 1, inicializamos com o valor zero a variável v_{rl}, que representa o valor da solução ótima do problema relaxado, e a variável v_{ip} que representa o valor da solução inteira corrente. Aplicamos, no passo 2, o algoritmo SimplexGC para resolver a relaxação linear do problema. Na primeira iteração guardamos o valor da solução ótima x fazendo $v_{rl} = \sum_{i=1}^{m} x_i$. Nas demais iterações $\sum_{i=1}^{m} x_i$ nunca vai exceder v_{rl}, portanto v_{rl} não será mais modificado durante o algoritmo.

No passo 3 truncamos as variáveis fracionárias fazendo $x_i^* = \lfloor x_i \rfloor$ $(i = 1, \ldots, m)$, e acrescentamos o valor desta solução parcial inteira a v_{ip} fazendo $v_{ip} = v_{ip} + \sum_{i=1}^{m} x_i^*$. A seguir verificamos, no passo 4, se alguma parte da demanda foi atendida pelas variáveis x^*. Em caso

afirmativo temos que $x_i^* > 0$ para algum $i = 1, ..., m$. Neste caso, retornamos, no passo 4.1, os padrões dados pelas colunas da matriz A com suas respectivas frequências, $x_1^*, \ldots, x_m^*$. Nos passos 4.2, 4.3 e 4.4, calculamos o problema residual. Primeiramente atualizamos a demanda fazendo $d_i = d_i - a_{ij}x_j^*$ $(i = 1, \ldots, m)$ e calculamos m', que representa a quantidade de itens cuja demanda ainda não foi totalmente atendida. Se $m' = 0$ então toda a demanda foi atendida portanto o algoritmo pára. Caso contrário, o problema residual deve ser resolvido. Para isso eliminamos os itens cuja demanda foi atendida, fazemos $m = m'$ e retornamos ao passo 2.

Se o teste executado no passo 4 falhar, ou seja, $x_i^* = 0$ $(i = 1, \ldots, m)$, então a solução obtida pelo método de geração de colunas, arredondada para baixo, não foi capaz de atender nenhuma parte da demanda. Neste caso, abandonamos o método de geração de colunas e utilizamos o algoritmo FFDWB para resolver este último problema residual.

No passo 5 fazemos $C = \lceil v_{rl} \rceil - v_{ip}$ e executamos o algoritmo FFDWB com os parâmetros $L, l_1, \ldots, l_m, d_1, \ldots, d_m, C$. Se o FFDWB não retornar 0, uma solução inteira utilizando C barras foi encontrada; neste caso retornamos, no passo 6, esta solução, dada por $\wp_1, \ldots, \wp_C$. Caso contrário, fazemos, no passo 7, uma nova chamada ao FFDWB com os parâmetros $L, l_1, \ldots, l_m$, $d_1, \ldots, d_m, C + 1$. Se o FFDWB não retornar 0, uma solução inteira utilizando $C + 1$ barras foi encontrada; neste caso retornamos, no passo 8, esta solução, dada por $\wp_1, \ldots, \wp_{C+1}$. Caso contrário, executamos o algoritmo FFDe com parâmetros $L, l_1, \ldots, l_m, d_1, \ldots, d_m$, retornamos a solução encontrada e paramos.

Algoritmo HÍBRIDO

Entrada: $(L, l_1, \ldots, l_m, d_1, \ldots, d_m)$.
Saída: Uma solução de: $\min \sum_{j=1}^{n} x_j$
sujeito a $Ax = d$
$x_j \geq 0$ e inteiro $\quad j = 1, \ldots, n$,
onde cada coluna j de A é tal que $\sum_{i=1}^{m} a_{ij} l_i \leq L$.

1 Faça $v_{rl} = 0$ e $v_{ip} = 0$.
2 Execute o algoritmo SimplexGC com parâmetros $L, l_1, \ldots, l_m, d_1, \ldots, d_m$. Se $\sum_{i=1}^{m} x_i > v_{rl}$ então faça $v_{rl} = \sum_{i=1}^{m} x_i$.
3 Para $i = 1$ até m faça $x_i^* = \lfloor x_i \rfloor$. Faça $v_{ip} = v_{ip} + \sum_{i=1}^{m} x_i^*$
4 Se $x_i^* > 0$ para algum $i = 1, ..., m$ então
  4.1 Retorne A e $x_1^*, \ldots, x_m^*$.
  4.2 Para $i = 1$ até m faça
    4.2.1 Para $j = 1$ até m faça $d_i = d_i - a_{ij} x_j^*$.
  4.3 Faça $m' = 0$.
  4.4 Para $i = 1$ até m faça
    4.3.1 Se $d_i > 0$ faça $m' = m' + 1$, $l_{m'} = l_i$ e $d_{m'} = d_i$.
  4.5 Se $m' = 0$ então pare.
  4.6 Faça $m = m'$ e retorne ao passo 2.
5 Faça $C = \lceil v_{rl} \rceil - v_{ip}$. Execute o FFDWB com parâmetros $L, l_1, \ldots, l_m, d_1, \ldots, d_m, C$.
6 Se o algoritmo FFDWB não retornou 0 então retorne $\wp_1, \ldots, \wp_C$ e pare.
7 Caso contrário, execute o algoritmo FFDWB com parâmetros $L, l_1, \ldots, l_m, d_1, \ldots, d_m, C+1$.
8 Se o algoritmo FFDWB não retornou 0 então retorne $\wp_1, \ldots, \wp_{C+1}$ e pare.
9 Caso contrário, execute o algoritmo FFDe com parâmetros $L, l_1, \ldots, l_m, d_1, \ldots, d_m$, retorne a solução encontrada e pare.

Podemos esperar que o algoritmo FFDWB resolva o último problema residual em tempo razoável, na maioria dos casos. Note que $C \leq m$. Ademais a demanda de cada item i está limitada por $m\lfloor \frac{L}{l_i} \rfloor$ $(i = 1, \ldots, m)$. Se pudermos estabelecer que $\frac{L}{l_i} \leq \alpha$ $(i = 1, \ldots, m)$, sendo α uma constante, podemos afirmar que o número máximo de ramificações de qualquer nó da árvore percorrida pelo FFDWB é constante.

Note que executamos o algoritmo FFDWB para tentar achar uma solução que utilize apenas C ou $C + 1$ barras. Se o FFDWB não puder encontrar tais soluções, usamos o algoritmo FFDe para encontrar uma solução inteira para o problema residual. O motivo pelo qual não utilizamos o FFDWB para procurar soluções com $C + 2, C + 3, \ldots$ barras é explicado na próxima seção.

3.4 A Conjectura MIRUP

Seja I uma instância de um problema de programação linear inteira P, no qual queremos minimizar a função objetivo, $v(I)$ o valor de uma solução inteira ótima de I e $v_{rl}(I)$ o valor de uma solução ótima da relaxação linear do problema. Baum e Trotter [6] definiram a *Integer Round-Up Property (IRUP)* da seguinte forma:

Definição 3.4.1. *Um problema de programação linear inteira P (de minimização) possui a integer round-up property (IRUP) se $v(I) = \lceil v_{rl}(I) \rceil$ para toda instância $I \in P$.*

Também dizemos que a IRUP é válida (ou se verifica) para uma instância I de um PLI se $v(I) = \lceil v_{rl}(I) \rceil$. Baum e Trotter [6] estabeleceram que esta propriedade é válida para certas classes de matrizes que surgem no contexto da teoria dos polimatróides. Em 1985, Marcotte [56] mostrou que várias classes de instâncias do problema de corte unidimensional possuem a IRUP. Até então conjecturava-se que a IRUP era válida para qualquer instância do problema de corte unidimensional. Por outro lado, Dyer, Frieze e McDiarmid [57] provaram o seguinte resultado.

Teorema 3.4.1. *É $\mathcal{NP}$-difícil determinar se uma instância qualquer do problema de corte unidimensional possui ou não a integer round-up property.*

No ano seguinte, no entanto, Marcotte [57] apresentou uma instância para a qual a IRUP não vale. Para esta instância $v(I) = \lceil v_{rl}(I) \rceil + 1$. Tal instância foi construída artificialmente a partir do problema da 4-PARTIÇÃO e apresentava coeficientes da ordem de 10^7, o que levou Marcotte a conjecturar que qualquer instância que não tivesse a IRUP deveria possuir coeficientes da ordem de 10^7, sendo portanto uma instância dissociada de problemas práticos.

Fieldhouse [30], no entanto, apresentou uma instância bastante simples para a qual a IRUP não vale:

$L = 30$, $l = \{15, 10, 6\}$ e $d = \{21, 32, 54\}$.

Para esta instância $v_{rl}(I) = 31,9666\ldots$ e $v(I) = 33$. Scheithauer e Terno [67], Gau [34], Schwerin e Wäscher [72] também exibiram instâncias com coeficientes pequenos, cuja diferença entre $v(I)$ e $v_{rl}(I)$ é maior que 1. Em nossos testes computacionais, também encontramos diversas instâncias cujo *gap* é maior que 1. Em [72] Schwerin e Wäscher apresentaram uma instância com 200 itens cujo *gap* entre o valor de uma solução inteira ótima e o valor de uma solução ótima da relaxação linear é 1,14435. Este é o maior *gap* conhecido até o momento. Estes resultados levaram Scheithauer e Terno [67] a definir a *Modified Integer Round-Up Property (MIRUP)*.

Definição 3.4.2. *Um problema de programação linear inteira P (de minimização) possui a modified integer round-up property (MIRUP) se $v(I) \leq \lceil v_{rl}(I) \rceil + 1$ para toda instância $I \in P$.*

Analogamente, dizemos que a MIRUP é válida (ou se verifica) para uma instância I de um

PLI se $v(I) \leq \lceil v_{rl}(I) \rceil + 1$. Em [67], Scheithuaer e Terno provaram que todos os problemas de corte unidimensional (i.e., seus correspondentes PLI) com $m \leq 5$ possuem a MIRUP. Em experimentos computacionais realizados por diversos autores [69, 85], onde foram determinadas soluções inteiras ótimas, verificou-se que todas as instâncias testadas (incluindo as geradas aleatoriamente e aquelas mencionadas na literatura) apresentavam a MIRUP. Não se conhece até o momento nenhuma instância do problema de corte unidimensional que não possua a MIRUP. Em 1995, Scheithauer e Terno [69] conjecturaram o que chamaremos de

Conjectura MIRUP: *O problema de corte unidimensional (3.1) possui a modified integer round-up property.*

Tal conjectura explica porque no algoritmo HÍBRIDO não usamos o FFDWB para buscar soluções com $C + 2, C + 3, \ldots$ barras. Se, no passo 8 do algoritmo HÍBRIDO, o FFDWB não encontrar uma solução que utiliza $C + 1$ barras, então o problema residual correspondente constitui uma instância que não possui a MIRUP, o que é bastante improvável.

3.4.1 Garantia de Desempenho para o Algoritmo HÍBRIDO

Sob a hipótese de que a conjectura MIRUP é verdadeira, temos o seguinte resultado, que estima a qualidade da solução encontrada pelo algoritmo híbrido.

Teorema 3.4.2. *Se a conjectura MIRUP for verdadeira, a diferença entre o valor da solução encontrada pelo algoritmo HÍBRIDO e o valor de uma solução inteira ótima é no máximo 1.*

Prova. O valor $\lceil v_{rl} \rceil$ é, claramente, um limite inferior para o valor de uma solução inteira ótima. Assumindo ser verdadeira a conjectura MIRUP, o algoritmo jamais chegará ao passo 9, pois o problema residual a ser resolvido no passo 8 tem solução cujo valor não excede C, portanto, pela conjectura MIRUP, possui solução inteira cujo valor é no máximo $C + 1$, solução essa que será retornada pelo algoritmo FFDWB. Se o algoritmo parar no passo 4.5 ou 6 a solução inteira encontrada terá valor $\lceil v_{rl} \rceil$. Se o algoritmo parar no passo 8 a solução inteira terá valor $\lceil v_{rl} \rceil + 1$. □

Podemos dizer então que o algoritmo HÍBRIDO é um algoritmo *quase-exato*, se verdadeira a conjectura MIRUP. Dada uma instância I, chamemos de $H(I)$ o valor da solução encontrada pelo algoritmo HÍBRIDO e $OPT(I)$ o valor de uma solução inteira ótima. O Teorema 3.4.2 implica que o limite de desempenho assintótico do algoritmo HÍBRIDO é 1, se verdadeira conjectura MIRUP. Mais precisamente, vale o seguinte resultado.

Corolário 3.4.3. *Se a conjectura MIRUP for verdadeira então $H(I) \leq OPT(I) + 1$ para toda instância I do problema de corte unidimensional.*

3.5 Modificando o Algoritmo HÍBRIDO

Introduzimos nesta seção uma mudança na regra de arredondamento utilizada no passo 3 do algoritmo HÍBRIDO. A idéia básica por trás desta modificação é diminuir o tamanho do problema residual. Como veremos no capítulo seguinte, esta modificação leva a um ganho *médio* considerável no tempo requerido para resolver o problema, sem que haja uma diminuição sensível na qualidade da solução. Chamaremos este algoritmo de *HÍBRIDOm*.

A modificação consiste em primeiramente arredondar para baixo as variáveis cuja parte fracionária é menor ou igual a $\frac{1}{2}$, ou seja, fazemos $x_i^* = \lfloor x_i \rfloor$ se $x_i - \lfloor x_i \rfloor \leq \frac{1}{2}$ $(i = 1, \ldots, m)$. Depois arredondamos as variáveis fracionárias, uma de cada vez, para cima, se isto não ocasionar produção de itens em excesso, ou para baixo, caso contrário. Implementamos tal procedimento calculando para cada variável $x_k \notin I\!N$ $(k = 1, \ldots, m)$ os valores $d_i' = \sum_{j=1}^{k-1} a_{ij} x_j^* + a_{ik} \lceil x_k \rceil + \sum_{j=k+1}^{m} a_{ij} x_j^*$ $(i = 1, \ldots, m)$. Se $d_i' \leq d_i$ $(i = 1, \ldots, m)$ então fazemos $x_k^* = \lceil x_k \rceil$, caso contrário, fazemos $x_k^* = \lfloor x_k \rfloor$.

Ao introduzirmos esta modificação, o Teorema 3.4.2 já não é mais válido. Ainda assim, podemos estabelecer um limite para o valor $H'(I)$ da solução encontrada pelo algoritmo HÍBRIDOm para uma instância I. Tomando por base o limite de desempenho absoluto do FFD no pior caso, que Simchi-Levi [73] provou ser 1,5, podemos afirmar que vale o seguinte resultado.

Fato 3.5.1. *$H'(I) \leq OPT(I) + \frac{m}{2}$ para toda instância I do problema de corte unidimensional.*

Nos testes realizados, que apresentaremos no capítulo seguinte, verificamos que as soluções encontradas pelo algoritmo HÍBRIDO e pelo algoritmo HÍBRIDOm são praticamente equivalentes, em termos de qualidade.

Algoritmo HÍBRIDOm

Entrada: $(L, l_1, \ldots, l_m, d_1, \ldots, d_m)$.
Saída: Uma solução de: $\min \sum_{j=1}^{n} x_j$
sujeito a $Ax = d$
$x_j \geq 0$ e inteiro $\quad j = 1, \ldots, n,$
onde cada coluna j de A é tal que $\sum_{i=1}^{m} a_{ij} l_i \leq L$.

1 Faça $v_{rl} = 0$ e $v_{ip} = 0$.
2 Execute o algoritmo SimplexGC com parâmetros $L, l_1, \ldots, l_m, d_1, \ldots, d_m$.
3 Se $\sum_{i=1}^{m} x_i > v_{rl}$ então faça $v_{rl} = \sum_{i=1}^{m} x_i$.
  3.1 Para $i = 1$ até m se $x_i - \lfloor x_i \rfloor \leq \frac{1}{2}$ faça $x_i^* = \lfloor x_i \rfloor$.
  3.2 Para $k = 1$ até m faça
    3.2.1 Para $i = 1$ até m faça $d_i' = \sum_{j=1}^{k-1} a_{ij} x_j^* + a_{ik} \lceil x_k \rceil + \sum_{j=k+1}^{m} a_{ij} x_j^*$.
    3.2.2 Se $d_i' \leq d_i$ para todo $i = 1, \ldots, m$ então faça $x_k^* = \lceil x_k \rceil$.
    3.2.3 Caso contrário, faça $x_k^* = \lfloor x_k \rfloor$.
  3.3 Faça $v_{ip} = v_{ip} + \sum_{i=1}^{m} x_i^*$
4 Se $x_i^* > 0$ para algum $i = 1, ..., m$ então
  4.1 Retorne A e $x_1^*, \ldots, x_m^*$.
  4.2 Para $i = 1$ até m faça
    4.2.1 Para $j = 1$ até m faça $d_i = d_i - a_{ij} x_j^*$.
  4.3 Faça $m' = 0$.
  4.4 Para $i = 1$ até m faça
    4.3.1 Se $d_i > 0$ faça $m' = m' + 1$, $l_{m'} = l_i$ e $d_{m'} = d_i$.
  4.5 Se $m' = 0$ então pare.
  4.6 Faça $m = m'$ e retorne ao passo 2.
5 Faça $C = \lceil v_{rl} \rceil - v_{ip}$. Execute o FFDWB com parâmetros $L, l_1, \ldots, l_m, d_1, \ldots, d_m, C$.
6 Se o algoritmo FFDWB não retornou 0 então retorne $\wp_1, \ldots, \wp_C$ e pare.
7 Caso contrário, execute o algoritmo FFDWB com parâmetros $L, l_1, \ldots, l_m, d_1, \ldots, d_m, C+1$.
8 Se o algoritmo FFDWB não retornou 0 então retorne $\wp_1, \ldots, \wp_{C+1}$ e pare.
9 Caso contrário, execute o algoritmo FFDe com parâmetros $L, l_1, \ldots, l_m, d_1, \ldots, d_m$, retorne a solução encontrada e pare.

Capítulo 4

Análise dos Resultados

Avaliamos os algoritmos híbridos propostos no capítulo anterior, resolvendo 6000 instâncias geradas aleatoriamente e mais cerca de duas centenas de instâncias práticas tiradas das plantas de ferragem de obras de uma construtora. O algoritmo HÍBRIDO e o algoritmo HÍBRIDOm foram implementados na linguagem C e os testes executados numa estação Sun Sparc 1000, com dois processadores, clock de 50 mhz e 704 MB de memória principal. Utilizamos o CPLEX 2.0 [18] para resolver os sistemas de equações lineares que aparecem nos passos 3 e 6 do algoritmo SimplexGC.

4.1 Resultados dos Testes Aleatórios

Dividimos os testes com instâncias aleatórias em dois grupos: 4000 instâncias geradas utilizando o método delineado por Wäscher e Gau em [86], e mais 2000 instâncias geradas aleatoriamente usando parâmetros que julgamos mais próximos dos problemas reais. Na próxima subseção abordamos o primeiro grupo de instâncias, que chamaremos de *instâncias de Wäscher e Gau*, e na subseção seguinte abordamos o segundo grupo de instâncias, que chamaremos de *instâncias estratificadas*.

4.1.1 Reproduzindo os Testes de Wäscher e Gau

Para gerar várias classes de instâncias aleatórias, variamos três parâmetros: o tamanho do problema, o intervalo de tamanho dos itens e o valor total da demanda.

O tamanho de um problema de corte unidimensional é expresso pelo valor m, que significa o número de itens. Nos testes realizados, m recebeu os valores 10, 20, 30, 40 e 50. De acordo com Wäscher e Gau [84], todas as instâncias práticas encontradas na literatura possuem tamanho menor que 50, o que justifica limitar o tamanho das instâncias aleatórias a este valor. Utilizamos

$L = 10000$ unidades de comprimento.

Os comprimentos dos itens foram modelados como variáveis aleatórias inteiras uniformemente distribuídas dentro do intervalo $[1, \frac{\beta L}{100}]$. O tamanho dos itens em relação ao tamanho L da barra afeta significativamente a dificuldade inerente ao problema e consequentemente a performance dos algoritmos propostos para o problema. Variamos o intervalo assumindo para β os valores 25, 50, 75 e 100.

Os valores d_i das demandas também foram tratadas como variáveis aleatórias inteiras. Elas foram geradas em duas etapas. Primeiramente calculamos a demanda total T fazendo $T = m\bar{d}$ ($\bar{d} \in I\!N$). Depois distribuímos a demanda total T pelos m itens. Podemos chamar $\bar{d}$ de *fator de multiplicação*. Variando este fator podemos mudar o caráter das instâncias, obtendo instâncias típicas do *bin-packing problem*, ao fixar $\bar{d}$ em valores pequenos, ou instâncias típicas do problema de corte de estoque, fixando $\bar{d}$ em valores grandes. Os valores que utilizamos para $\bar{d}$ foram 10 e 50.

Para distribuir a demanda total T pelos m itens, geramos os números aleatórios $R_1, \ldots, R_m$, uniformemente distribuídos dentro do intervalo [0,1], e então calculamos a demanda de cada item fazendo $d_i = max(1, \lfloor \frac{R_i}{R_1+\ldots+R_m} \rfloor T)$ para $i = 1, \ldots, m-1$ e $d_m = max(1, T - \sum_{i=1}^{m-1} d_i)$.

Combinando os diferentes valores destes três parâmetros definimos 40 classes de instâncias, classes estas caracterizadas pela tripla $(m, \beta, \bar{d})$. Para cada classe, 100 instâncias do problema foram geradas, perfazendo um total de 4000 instâncias. As instâncias foram geradas usando o algoritmo *CUTGEN1*, descrito em [36], e que implementa os procedimentos descritos nos três últimos parágrafos. Tal algoritmo recebe como entrada os parâmetros m, β, $\bar{d}$ e um valor usado para inicializar a semente da função que gera números pseudo-aleatórios. De forma a obter as mesmas instâncias utilizadas por Wäscher e Gau, inicializamos a semente usando o número obtido pela concatenação de m, β (com 3 dígitos) e $\bar{d}$. Por exemplo, ao gerar as instâncias da classe caracterizada por $m = 10$, $\beta = 75$ e $\bar{d} = 10$, o número utilizado para inicializar a semente foi 1007510.

Apesar de o algoritmo FFDWB ter capacidade de achar uma solução inteira ótima, se ela existir, os testes computacionais mostraram que em algumas instâncias o tempo de computação requerido pelo FFDWB era inaceitável. Por isso, em nossa implementação, limitamos em 250000 o número de nós que podem ser percorridos pelo FFDWB. Tal procedimento também foi adotado por Wäscher e Gau [86] ao restringir em 25000 o número de retrocessos efetuados pelo algoritmo MTP devido a Martello e Toth [59].

Na Tabela 4.1 apresentamos os resultados da aplicação do algoritmo HÍBRIDO às 4000 instâncias aleatórias geradas conforme descrevemos anteriormente. Nesta tabela, agrupamos as instâncias de classes que diferem apenas no parâmetro $\bar{d}$, para poder comparar com os resultados obtidos por Wäscher e Gau. Assim, cada linha da tabela representa os resultados obtidos na

resolução de 200 instâncias.

A coluna *IRUP* representa o número de instâncias para as quais foi encontrada solução inteira que satisfaz a IRUP. A coluna *MIRUP* representa o número de instâncias para as quais foi encontrada solução inteira que satisfaz a MIRUP e não satisfaz a IRUP. A coluna *Colunas geradas* indica o número médio de colunas geradas pela aplicação do algoritmo SimplexGC em cada instância. A coluna *Tempo* informa o tempo médio requerido para resolver cada instância.

Comparamos a solução encontrada pelo algoritmo HÍBRIDO com a solução obtida pelo algoritmo FFD. Na coluna *Pior que FFD* indicamos o número de instâncias para as quais a solução encontrada pelo algoritmo HÍBRIDO foi pior que a solução encontrada pelo FFD. Informamos na coluna *Ganho sobre FFD* o ganho médio percentual da solução encontrada pelo algoritmo HÍBRIDO em relação à solução encontrada pelo FFD.

Finalmente, na coluna *Desperdício* indicamos o desperdício médio da solução encontrada pelo algoritmo HÍBRIDO. Consideramos como desperdício a parte não aproveitada das barras, independentemente da possibilidade de encontrar solução melhor, ou seja, o desperdício é dado pela razão entre o valor da solução inteira encontrada v_{ip} e $\frac{\sum_{i=1}^{m} d_i l_i}{L}$. Dessa forma, analisando a segunda linha da tabela, verificamos que ao resolver as 200 instâncias de tamanho m=10, cujos itens têm comprimento variando entre 1 e 7500 unidades de comprimento, desperdiçamos, em média, 15,084%, embora todas as soluções encontradas sejam ótimas. O propósito da coluna *Desperdício* é dar ao leitor uma idéia a respeito do desperdício inerente a essas classes de instâncias.

Pela Tabela 4.1 percebemos que a MIRUP foi verificada em todas as 4000 instâncias, sendo que em apenas 12 delas a IRUP não foi verificada. Isto significa que em no máximo 12 instâncias o algoritmo HÍBRIDO deixou de encontrar uma solução ótima. No entanto, a solução encontrada para estas 12 instâncias difere da solução ótima de no máximo 1, o que está de acordo com o Teorema 3.4.2.

Analisando a coluna *Tempo*, verificamos que o algoritmo HÍBRIDO encontra mais dificuldade para resolver instâncias onde a maior parte dos itens não apresenta comprimento muito pequeno nem muito grande em relação à barra. Estas instâncias estão caracterizadas por $\beta = 50$, ou seja, são as instâncias onde os itens têm no máximo a metade do comprimento da barra. Por outro lado, analisando a coluna *Ganho sobre o FFD*, percebemos que é justamente para $\beta = 50$ que o algoritmo HÍBRIDO apresenta maior vantagem em relação ao FFD. É interessante notar que, mesmo para a classe onde $m = 50$ e $\beta = 50$, o tempo médio necessário para resolver cada problema foi em torno de 20 segundos, o que, em termos práticos, é satisfatório.

Uma outra informação importante (embora não expressa na Tabela 4.1) é que a maior parte do tempo requerido para encontrar uma solução foi gasto na geração de colunas e não pelo algoritmo FFDWB. Em média, o tempo requerido pelo FFDWB foi inferior a 0,1 segundos.

m	Tamanho dos itens	IRUP	MIRUP	Colunas geradas	Tempo (seg)	Pior que FFD	Ganho sobre FFD	Desperdício
10	0%-100%	200	0	7,21	0,04	0	0,345%	13,734%
	0%-75%	200	0	11,92	0,07	0	0,569%	15,084%
	0%-50%	200	0	23,77	0,14	0	2,727%	3,091%
	0%-25%	200	0	41,02	0,34	0	1,590%	1,388%
20	0%-100%	199	1	26,01	0,17	0	0,273%	9,873%
	0%-75%	199	1	47,37	0,32	0	0,641%	7,450%
	0%-50%	200	0	108,98	0,97	0	2,322%	0,695%
	0%-25%	200	0	105,60	1,24	0	0,864%	0,665%
30	0%-100%	199	1	56,70	0,45	0	0,225%	10,046%
	0%-75%	200	0	112,53	1,02	0	0,545%	6,376%
	0%-50%	198	2	293,29	3,87	0	1,887%	0,362%
	0%-25%	200	0	187,85	2,71	0	0,551%	0,451%
40	0%-100%	200	0	99,31	0,85	0	0,200%	9,901%
	0%-75%	198	2	226,31	2,66	0	0,633%	4,301%
	0%-50%	200	0	585,84	9,50	0	1,456%	0,206%
	0%-25%	200	0	289,55	4,14	0	0,435%	0,348%
50	0%-100%	200	0	159,15	1,55	0	0,227%	7,175%
	0%-75%	197	3	375,12	4,78	0	0,582%	4,399%
	0%-50%	198	2	969,48	20,28	0	1,157%	0,157%
	0%-25%	200	0	397,06	17,80	0	0,363%	0,275%

Tabela 4.1: Algoritmo HÍBRIDO aplicado às instâncias de Wäscher e Gau.

Em todas as classes de instâncias o algoritmo HÍBRIDO mostrou ser superior ao FFD. Percebemos ainda que o desperdício diminui à medida que o tamanho do problema cresce, o que explica, em parte, por que o ganho sobre o FFD, de uma forma geral, também diminui quando o tamanho do problema aumenta.

Na Tabela 4.2 estão os resultados obtidos ao resolver as mesmas 4000 instâncias usando o algoritmo HÍBRIDOm. Os resultados assemelham-se bastante aos resultados obtidos pelo algoritmo HÍBRIDO, com apenas duas diferenças significativas. Observamos que o tempo gasto pelo algoritmo HÍBRIDOm foi, de uma forma geral, menor do que o tempo gasto pelo algoritmo HÍBRIDO, especialmente nas classes com $m \geq 30$. Verificamos que esta redução no tempo requerido para resolver os problemas foi da ordem de 20%. O tempo médio requerido pelo algoritmo FFDWB também foi reduzido; o FFDWB gastou, em média, menos de 0,01 segundos para cada instância.

Em contrapartida a qualidade da solução foi um pouco inferior, em comparação com o algoritmo HÍBRIDO. Em no máximo 29 instâncias o algoritmo HÍBRIDOm não encontrou uma solução ótima, achando porém uma solução cujo valor diferia da ótima de apenas 1. Além disso, em 1 instância a solução encontrada pelo algoritmo HÍBRIDOm foi pior que a solução encontrada pelo algoritmo FFD (para esta instância o FFD achou uma solução inteira ótima).

m	Tamanho dos itens	IRUP	MIRUP	Colunas geradas	Tempo (seg)	Pior que FFD	Ganho sobre FFD	Desperdício
10	0%-100%	200	0	7,22	0,04	0	0,345%	13,734%
	0%-75%	199	1	11,80	0,05	0	0,565%	15,089%
	0%-50%	199	1	22,98	0,13	0	2,721%	3,098%
	0%-25%	200	0	39,01	0,32	0	1,590%	1,388%
20	0%-100%	199	1	25,48	0,17	0	0,273%	9,873%
	0%-75%	198	2	45,86	0,24	0	0,639%	7,452%
	0%-50%	199	1	100,22	0,70	0	2,319%	0,698%
	0%-25%	200	0	101,55	1,05	0	0,864%	0,665%
30	0%-100%	198	2	54,94	0,43	0	0,224%	10,047%
	0%-75%	200	0	107,19	0,69	0	0,545%	6,376%
	0%-50%	198	2	255,22	2,67	0	1,887%	0,362%
	0%-25%	200	0	181,93	2,39	0	0,551%	0,451%
40	0%-100%	196	4	97,40	0,94	0	0,197%	9,905%
	0%-75%	196	4	205,59	1,79	0	0,631%	4,304%
	0%-50%	198	2	485,94	6,63	0	1,453%	0,210%
	0%-25%	200	0	277,71	4,12	0	0,435%	0,348%
50	0%-100%	198	2	157,41	1,94	0	0,226%	7,176%
	0%-75%	195	5	340,38	3,52	1	0,580%	4,401%
	0%-50%	198	2	828,75	16,40	0	1,157%	0,157%
	0%-25%	200	0	386,86	16,83	0	0,363%	0,275%

Tabela 4.2: Algoritmo HÍBRIDOm aplicado às instâncias de Wäscher e Gau.

No entanto, o algoritmo HÍBRIDOm verificou a IRUP em duas instâncias para as quais o algoritmo HÍBRIDO não verificou esta propriedade. Isto significa que eventualmente o algoritmo HÍBRIDOm pode encontrar uma solução superior à encontrada pelo algoritmo HÍBRIDO. Portanto estes dois algoritmos podem ser aplicados de forma complementar. Primeiramente tentamos resolver a instância com o algoritmo HÍBRIDOm, que é usualmente mais rápido. Se este algoritmo não verificar a IRUP, aplicamos então o algoritmo HÍBRIDO na tentativa de obter

uma solução melhor.

Nas figuras que se seguem, comparamos o tempo, o número de colunas geradas, o ganho em relação ao FFD e o desperdício obtido ao aplicar os algoritmos híbridos anteriormente propostos às instâncias de Wäscher e Gau. As figuras 4.1 e 4.2 mostram, respectivamente, o tempo médio requerido e o número médio de colunas geradas pelos algoritmos híbridos em função do tamanho das instâncias.

Percebemos que em todas as classes de instâncias, o algoritmo HÍBRIDOm é, em média, um pouco mais rápido que o algoritmo HÍBRIDO. Como poderíamos esperar, o número médio de colunas geradas também é um pouco menor no algoritmo HÍBRIDOm. Como mencionamos anteriormente, as classes nas quais é necessário gerar um maior número de colunas, e que consequentemente requerem mais tempo para sua resolução, são aquelas em que $\beta = 50$.

As figuras 4.3 e 4.4 mostram o ganho médio em relação ao FFD e o desperdício médio em função de m, respectivamente. Observe que as diferenças entre o ganho médio em relação ao FFD e o desperdício médio verificados nos algoritmos híbridos são praticamente imperceptíveis (pelo menos visualmente, na forma de gráfico). No entanto, como verificamos pelas tabelas 4.1 e 4.2, o ganho médio em relação ao FFD é levemente superior no algoritmo HÍBRIDO. Consequentemente, o desperdício médio no algoritmo HÍBRIDO é levemente inferior.

Percebemos também que o ganho médio em relação ao FFD e o desperdício médio tendem a diminuir e se estabilizar à medida que m cresce. É importante notar que o desperdício médio nas classes onde $\beta \leq 50$ tende a se estabilizar próximo de zero.

A Tabela 4.3 apresenta uma comparação entre os resultados obtidos pelos algoritmos híbridos aqui propostos e diversos métodos encontrados na literatura, quando aplicadas às instâncias de Wäscher e Gau. Adotamos para estes métodos os nomes sugeridos por Wäscher e Gau [86] e indicamos as referências onde o leitor pode obter detalhes deste métodos. Dentre todos estes métodos, o algoritmo HÍBRIDO foi o que encontrou soluções ótimas para o maior número de instâncias.

Baseados nestes testes, não encontramos diferenças significativas entre o algoritmo HÍBRIDO e o algoritmo HÍBRIDOm. No entanto, acreditamos que as instâncias utilizadas estão, em certo sentido, distantes das instâncias práticas pois os comprimentos dos itens são tomados aleatoriamente dentro do intervalo $[1, \frac{\beta L}{100}]$. Portanto, nestas instâncias aleatórias, frequentemente temos itens bem pequenos, menores que 1% da barra. De certa forma isto facilita encontrar soluções de boa qualidade em tempo reduzido. Na próxima subseção apresentamos os resultados destes dois algoritmos aplicados a instâncias onde não existem itens de comprimento menor que $\frac{L}{100}$.

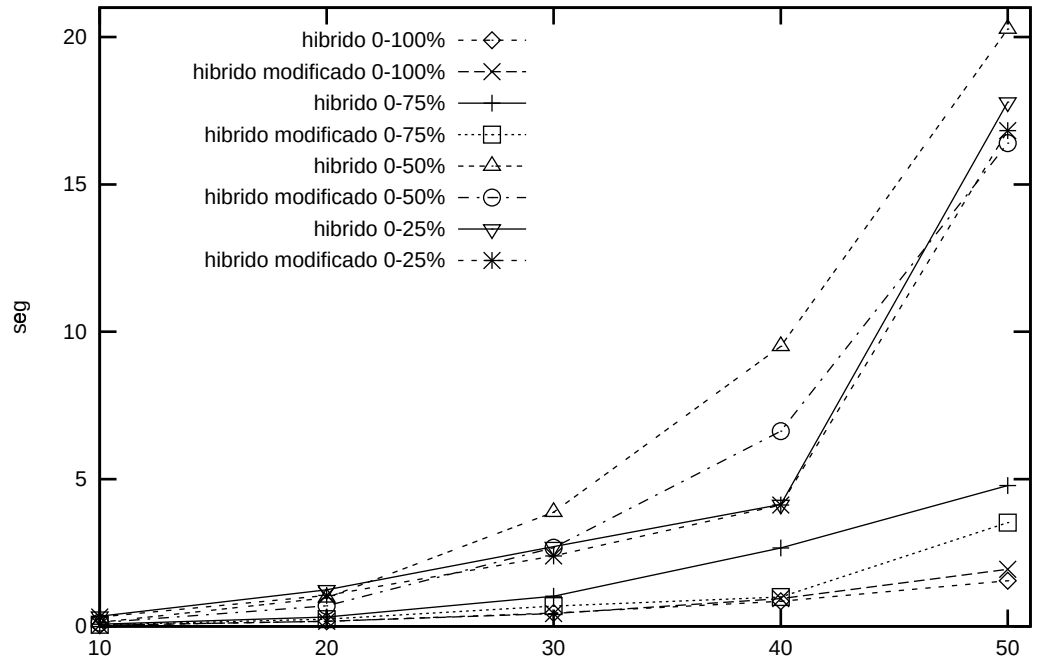

Figura 4.1: Tempo médio requerido para resolver as instâncias de Wäscher e Gau.

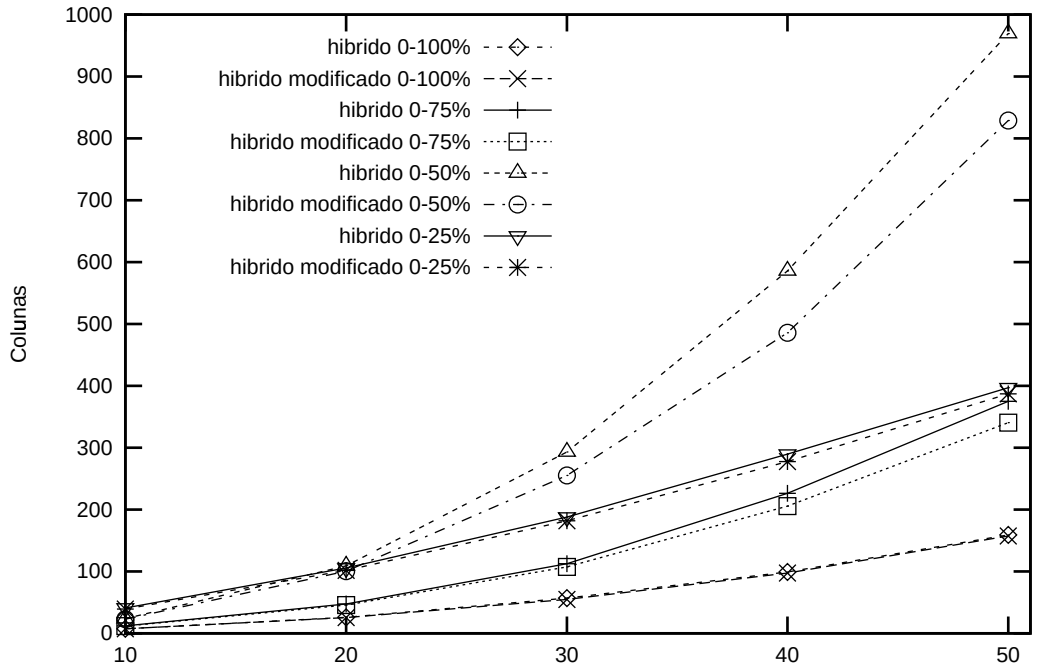

Figura 4.2: Número médio de colunas geradas ao resolver as instâncias de Wäscher e Gau.

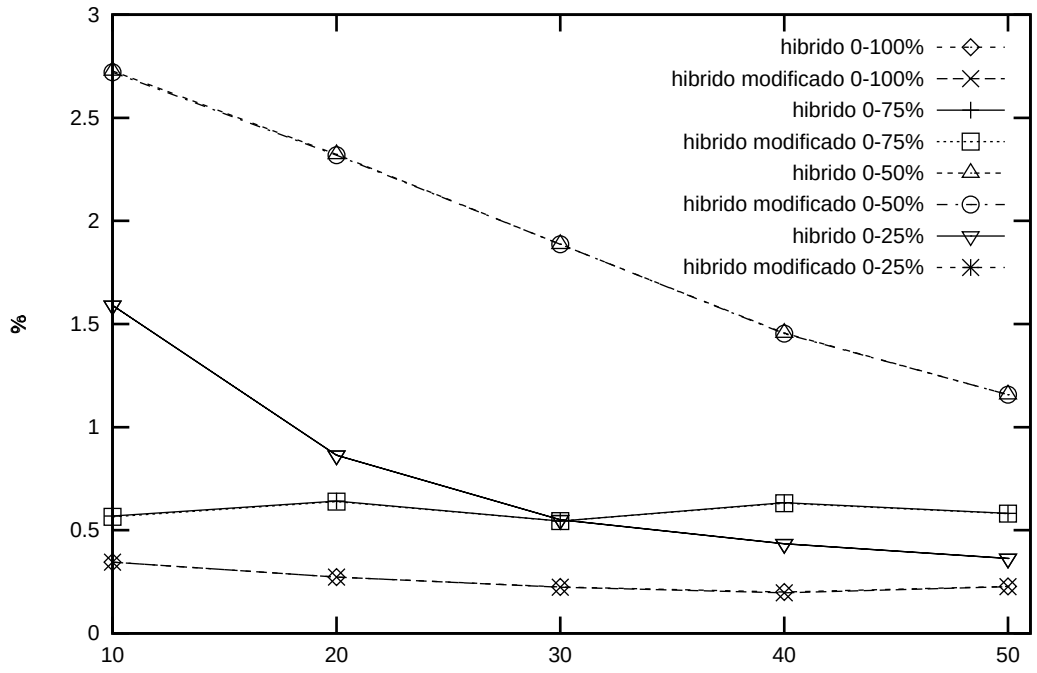

Figura 4.3: Ganho médio em relação ao FFD obtido ao resolver as instâncias de Wäscher e Gau.

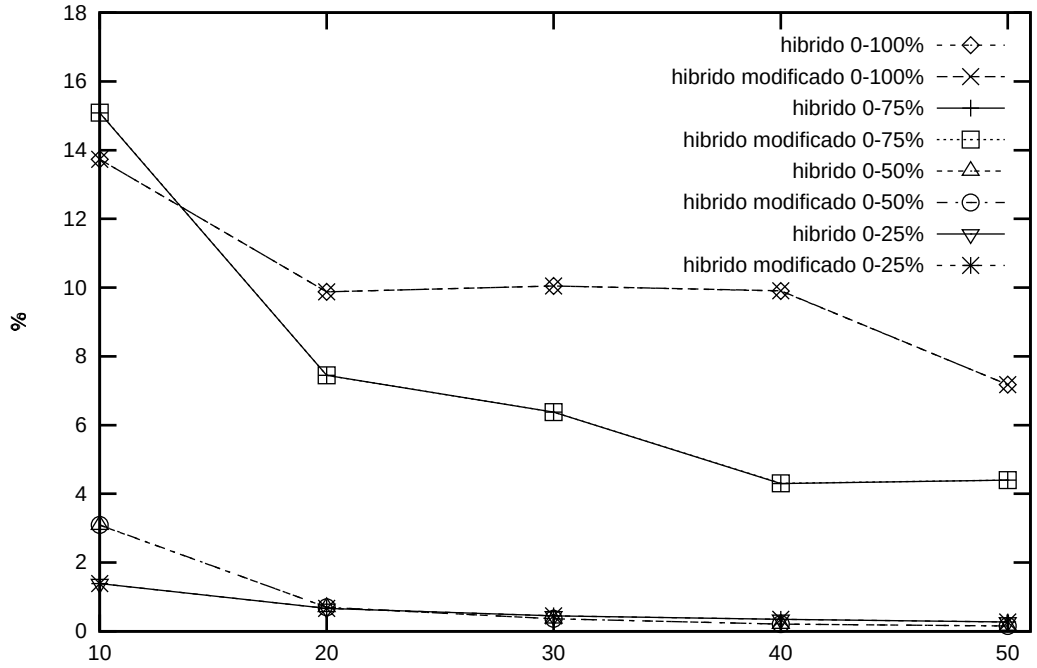

Figura 4.4: Desperdício médio obtido ao resolver as instâncias de Wäscher e Gau.

Algoritmo	$m = 10$	$m = 20$	$m = 30$	$m = 40$	$m = 50$	Total
Total de instâncias	800	800	800	800	800	4000
HÍBRIDO	800	798	797	798	795	3988
HÍBRIDOm	798	796	796	790	791	3971
RSUC [34]	797	792	790	779	763	3921
ROPT [86]	796	788	782	770	738	3874
CSTAOPT [74]	758	754	731	732	734	3709
RFFD [86]	779	758	743	726	695	3701
CSTAFFD [86]	752	746	715	716	701	3630
FFD	469	378	359	321	318	1845
BOPT [86]	428	321	262	251	210	1472
BRURED [64]	325	193	129	94	80	821
BRUSUC [74]	257	142	74	53	52	578
BRUSIM [86]	93	47	23	15	10	188

Tabela 4.3: Soluções inteiras ótimas obtidas por diversos algoritmos aplicados às instâncias de Wäscher e Gau.

4.1.2 Mais Testes Aleatórios

Para gerar várias classes de instâncias aleatórias, variamos o tamanho do problema e o intervalo de tamanho dos itens. Diferentemente dos testes realizados por Wäscher e Gau, adotamos para m os valores 10, 20, 30, 50 e 100, de forma a analisar o comportamento dos algoritmos propostos para instâncias relativamente grandes.

Utilizamos $L = 10000$ unidades de comprimento. Os comprimentos dos itens foram modelados como variáveis aleatórias inteiras uniformemente distribuídas dentro do intervalo $[\frac{\alpha L}{100}, \frac{\beta L}{100}]$. Dessa forma o intervalo é definido pelo par (α, β). Escolhemos como valores de (α, β) os pares (10,70), (20,50), (5,20) e (1,5). Estes quatro pares definem instâncias cujos itens têm comprimentos variados, médios, pequenos e muito pequenos, respectivamente, em relação ao comprimento L. Por este motivo resolvemos chamá-las de *instâncias estratificadas*.

Nossa experiência computacional demonstrou ser quase sempre trivial resolver instâncias nas quais a maioria dos itens são grandes, digamos maiores que $\frac{L}{3}$. Ademais, instâncias com esta característica não são típicas no mundo real. Dessa forma, escolhemos não executar testes com classes de instâncias onde os itens têm apenas tamanhos grandes. Os valores d_i das demandas também são variáveis aleatórias inteiras uniformemente distribuídas dentro do intervalo [1,10000].

Combinando os diferentes valores destes dois parâmetros definimos 20 classes de instâncias, classes estas caracterizadas pelo par $(m, (\alpha, \beta))$. Para cada classe, 100 instâncias do problema foram geradas, perfazendo um total de 2000 instâncias. Utilizamos como gerador de números pseudo-aleatórios a função *drand48*, que faz parte da biblioteca *stdlib.h* da linguagem C. Tal função gera números pseudo-aleatórios de precisão dupla em ponto flutuante uniformemente distribuídos dentro do intervalo [0,1]. Inicializamos a semente da função *drand48* com o valor 1000.

A Tabela 4.4 apresenta os resultados obtidos pelo algoritmo HÍBRIDO ao resolver as 2000 instâncias geradas conforme acabamos de descrever. Observamos que o algoritmo HÍBRIDO resolveu com relativa facilidade todas as classes de instâncias, exceto a classe caracterizada por (100,(20,50)). Como já tínhamos observado na subseção anterior, as instâncias nas quais os itens têm tamanho intermediário parecem ter uma complexidade intrínseca maior.

A propriedade IRUP foi verificada em 1939 instâncias, enquanto que a MIRUP só não foi verificada em 13 das 61 instâncias restantes. Todas essas 13 instâncias pertencem à classe caracterizada por (100,(20,50)) e uma solução melhor não foi encontrada porque, ao tentar resolver o problema residual, o algoritmo FFDWB alcançou o limite de 250000 nós percorridos. Mesmo assim, das 13 instâncias restantes, 8 apresentaram solução com valor $v_{ip} = \lceil v_{rl}(I) \rceil + 2$, 4 tiveram solução com valor $v_{ip} = \lceil v_{rl}(I) \rceil + 3$, e 1 instância apresentou solução com valor $v_{ip} = \lceil v_{rl}(I) \rceil + 4$.

Exceto para a classe (100,(20,50)), o tempo médio gasto para resolver os problemas foi bastante satisfatório. No entanto, como veremos mais adiante, as classes de instâncias com $m = 100$ podem ser resolvidas mais rapidamente e com melhor qualidade nas soluções encontradas usando-se o algoritmo HÍBRIDOm. Como citamos anteriormente, o FFDWB pode requerer tempo de execução exponencial, apresentando desempenho satisfatório apenas para instâncias pequenas.

Ao resolver as classes de instâncias com $m \leq 50$, o FFDWB gastou, em média, menos de 0,7 segundos. Por outro lado, para as classes com $m = 100$, o tempo médio requerido pelo algoritmo FFDWB foi de aproximadamente 18 segundos, sendo que na classe caracterizada por (100,(20,50)) o FFDWB gastou, em média, aproximadamente 70 segundos. Mesmo nesta classe, a maior parte do tempo foi gasta na geração de colunas.

Comparando as soluções do algoritmo HÍBRIDO com as soluções do algoritmo FFD, notamos um fato bastante interessante: quando os itens têm comprimento médio ou grande, digamos maior que $\frac{L}{5}$, o ganho em relação ao FFD aumenta à medida que m cresce. Observamos ainda que ao resolver instâncias com itens pequenos ($\beta \leq 20$), o desperdício médio é praticamente nulo.

m	Tamanho dos itens	IRUP	MIRUP	Colunas geradas	Tempo (seg)	Pior que FFD	Ganho sobre FFD	Desperdício
10	10%-70%	100	0	11,55	0,06	0	0,845%	11,258%
	20%-50%	100	0	10,60	0,05	0	4,642%	5,631%
	5%-20%	100	0	38,84	0,36	0	2,353%	0,057%
	1%-5%	100	0	16,14	0,14	0	0,500%	0,034%
20	10%-70%	100	0	40,83	0,23	0	1,082%	7,198%
	20%-50%	100	0	49,21	0,36	0	5,867%	3,392%
	5%-20%	100	0	104,16	1,22	0	1,856%	0,004%
	1%-5%	100	0	27,85	0,25	0	0,347%	0,017%
30	10%-70%	100	0	95,94	0,70	0	1,242%	3,992%
	20%-50%	98	2	132,06	2,41	0	6,379%	2,247%
	5%-20%	100	0	159,19	3,44	0	1,587%	0,003%
	1%-5%	100	0	38,76	0,49	0	0,284%	0,012%
50	10%-70%	100	0	282,58	3,14	0	1,378%	2,227%
	20%-50%	85	15	487,07	16,37	0	6,660%	1,362%
	5%-20%	100	0	296,33	6,02	0	1,378%	0,002%
	1%-5%	100	0	66,08	2,14	0	0,237%	0,007%
100	10%-70%	95	5	1513,48	32,52	0	1,471%	0,731%
	20%-50%	38	49	2437,21	159,32	0	6,854%	0,762%
	5%-20%	100	0	783,34	28,86	0	1,232%	0,001%
	1%-5%	100	0	132,65	43,56	0	0,203%	0,004%

Tabela 4.4: Algoritmo HÍBRIDO aplicado às instâncias estratificadas.

Passamos agora a analisar os resultados obtidos pelo algoritmo HÍBRIDOm com relação às mesmas 2000 instâncias. A Tabela 4.5 mostra que em apenas 17 instâncias a IRUP não foi verificada. Isto significa que o algoritmo HÍBRIDO encontrou uma solução ótima para, pelo menos, 1983 instâncias. Vemos também que nas 17 instâncias remanescentes a MIRUP foi verificada.

Vale destacar que o algoritmo HÍBRIDO verificou a IRUP para 5 das instâncias em que o algoritmo HÍBRIDOm não verificou esta propriedade. Dessa forma, em apenas 12 instâncias a IRUP não foi verificada por nenhum dos dois algoritmos. Isto reforça o que já havíamos dito anteriormente, quando sugerimos que estes dois algoritmos podiam ser usadas de forma complementar.

Observamos também que o tempo gasto pelo algoritmo HÍBRIDOm foi significativamente

menor que o tempo gasto pelo algoritmo HÍBRIDO. Tal redução foi especialmente significativa para as classes com ($\alpha = 20$, $\beta = 50$). Isto pode ser explicado pelo fato de o algoritmo HÍBRIDOm diminuir o tamanho do problema residual a ser resolvido pelo algoritmo FFDWB. O tempo médio gasto pelo algoritmo FFDWB para resolver cada instância, incluindo as classes com $m = 100$, foi inferior a 0,02 segundos.

m	Tamanho dos itens	IRUP	MIRUP	Colunas geradas	Tempo (seg)	Pior que FFD	Ganho sobre FFD	Desperdício
10	10%-70%	100	0	11,37	0,05	0	0,845%	11,258%
	20%-50%	99	1	9,95	0,04	0	4,642%	5,632%
	5%-20%	100	0	35,92	0,34	0	2,353%	0,057%
	1%-5%	100	0	16,14	0,12	0	0,500%	0,034%
20	10%-70%	100	0	39,52	0,23	0	1,082%	7,198%
	20%-50%	100	0	38,07	0,26	0	5,867%	3,392%
	5%-20%	100	0	85,17	1,72	0	1,856%	0,004%
	1%-5%	100	0	27,83	0,26	0	0,347%	0,017%
30	10%-70%	100	0	88,22	0,71	0	1,242%	3,992%
	20%-50%	98	2	89,17	0,86	0	6,379%	2,247%
	5%-20%	100	0	130,15	2,11	0	1,587%	0,003%
	1%-5%	100	0	39,12	0,53	0	0,284%	0,012%
50	10%-70%	97	3	236,99	2,99	0	1,378%	2,227%
	20%-50%	98	2	239,92	4,15	0	6,661%	1,361%
	5%-20%	100	0	235,94	5,12	0	1,378%	0,002%
	1%-5%	100	0	64,78	2,13	0	0,237%	0,007%
100	10%-70%	97	3	1045,08	26,17	0	1,471%	0,731%
	20%-50%	94	6	972,48	51,74	0	6,854%	0,761%
	5%-20%	100	0	565,91	21,89	0	1,232%	0,001%
	1%-5%	100	0	126,02	42,67	0	0,203%	0,004%

Tabela 4.5: Algoritmo HÍBRIDOm aplicado às instâncias estratificadas.

Verificamos que nas classes de instâncias abordadas nesta subseção, o algoritmo HÍBRIDOm apresentou melhor qualidade nas soluções e um considerável ganho no tempo requerido para encontrar as soluções. Considerando estas instâncias como mais representativas das instâncias do mundo real, isso nos permite indicar o algoritmo HÍBRIDOm como mais adequado, em especial para instâncias de maior porte.

Finalizando nossa análise dos testes baseados em instâncias aleatórias, podemos concluir que

os algoritmos híbridos propostos apresentam tempo de execução satisfatório, pelo menos para $m \leq 100$, e encontram uma solução inteira ótima na grande maioria dos casos. Mesmo quando isto não ocorre, o valor da solução encontrada difere muito pouco, geralmente de uma unidade, do valor de uma solução ótima.

Nas figuras que se seguem, comparamos o tempo, o número de colunas geradas, o ganho em relação ao FFD e o desperdício obtido ao aplicar os algoritmos híbridos anteriormente propostos às instâncias estratificadas. As figuras 4.5 e 4.6 mostram, respectivamente, o tempo médio requerido e o número médio de colunas geradas pelos algoritmos híbridos em função do tamanho das instâncias.

Percebemos que em todas as classes de instâncias, o algoritmo HÍBRIDOm é, em média, mais rápido que o algoritmo HÍBRIDO. Como poderíamos esperar, o número médio de colunas geradas também é um pouco menor no algoritmo HÍBRIDOm.

As figuras 4.7 e 4.8 mostram o ganho médio em relação ao algoritmo FFD e o desperdício médio em função de m, respectivamente. Observe que as diferenças entre o ganho médio em relação ao FFD e o desperdício médio verificados nos algoritmos híbridos são praticamente imperceptíveis (pelo menos visualmente, na forma de gráfico). No entanto, como verificamos pelas tabelas 4.4 e 4.5, o ganho médio em relação ao FFD é levemente superior no algoritmo HÍBRIDOm. Consequentemente, o desperdício médio no algoritmo HÍBRIDOm é levemente inferior.

Surpreendentemente, o ganho médio em relação ao FFD cresce nas classes de instâncias onde ($\alpha = 10$, $\beta = 70$) e ($\alpha = 20$, $\beta = 50$), à medida que o tamanho do problema aumenta. Nas outras classes testadas, o ganho médio em relação ao FFD tende a diminuir e se estabilizar. Percebemos também que o desperdício médio tende a diminuir e se estabilizar à medida que m cresce. Vale notar que o desperdício médio nas classes onde $\beta \leq 20$ tende a se estabilizar muito próximo de zero.

Comparando com os resultados obtidos por Wäscher e Gau [86] ao utilizar diversas heurísticas para resolver 4000 instâncias abordadas na subseção anterior, observamos que a qualidade das soluções encontradas pelos algoritmos híbridos é superior, visto que encontramos mais frequentemente uma solução ótima (veja Tabela 4.3). É pertinente observar que os resultados obtidos por Wäscher e Gau em 1996 eram os melhores até então.

Os resultados dos testes com essas 6000 instâncias aleatórias vêm fortalecer a conjectura MIRUP. Além das instâncias aqui abordadas, resolvemos também *alguns milhões* de instâncias geradas aleatoriamente, com m variando entre 10 e 20, com o objetivo de encontrar um contra-exemplo para a conjectura MIRUP. Não encontramos uma instância sequer com *gap* entre v_{ip} e v_{rl} maior que 1,14435.

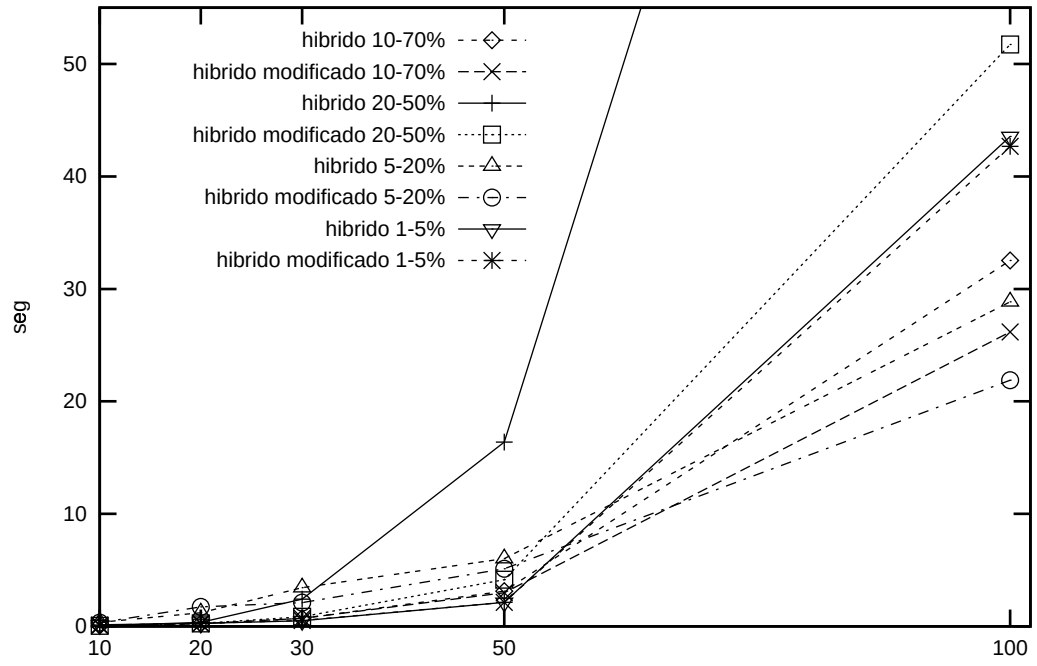

Figura 4.5: Tempo médio requerido para resolver as instâncias estratificadas.

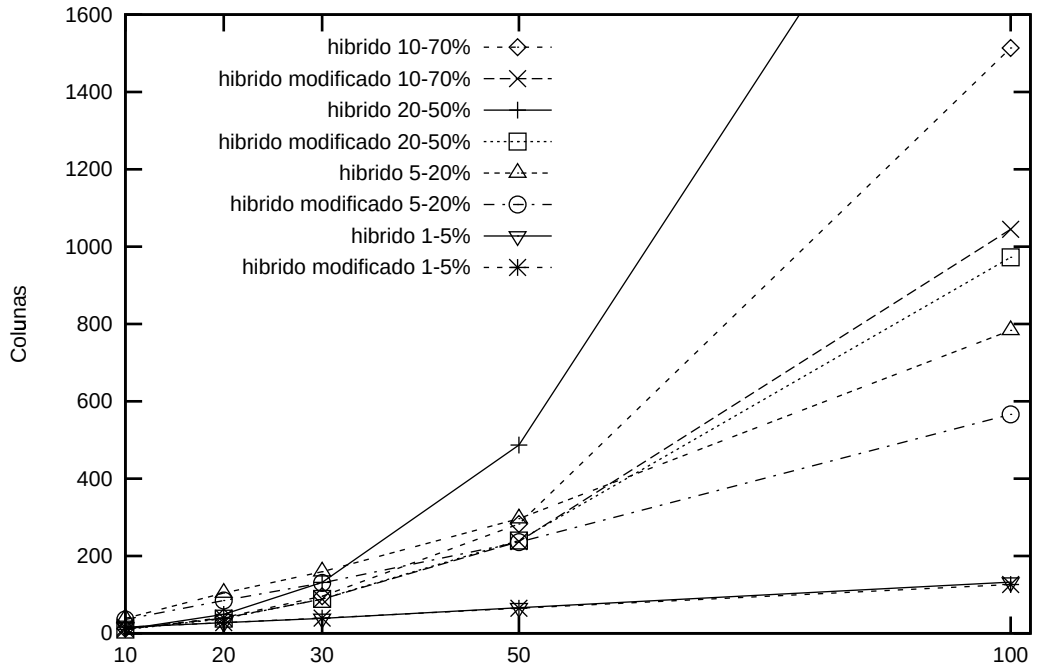

Figura 4.6: Número médio de colunas geradas ao resolver as instâncias estratificadas.

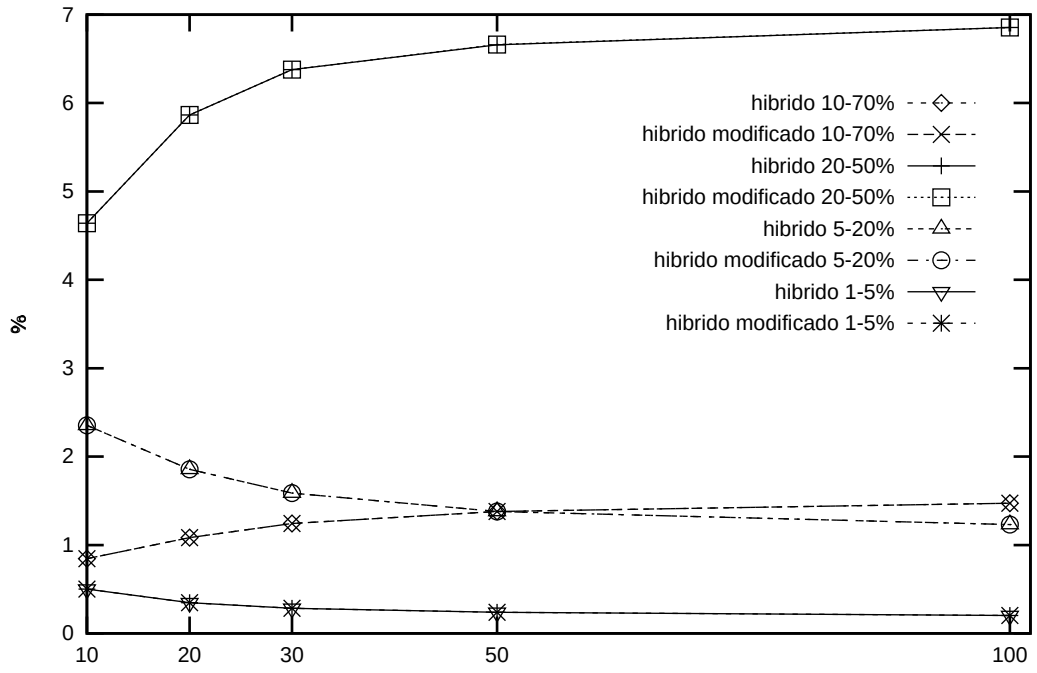

Figura 4.7: Ganho médio em relação ao FFD obtido ao resolver as instâncias estratificadas.

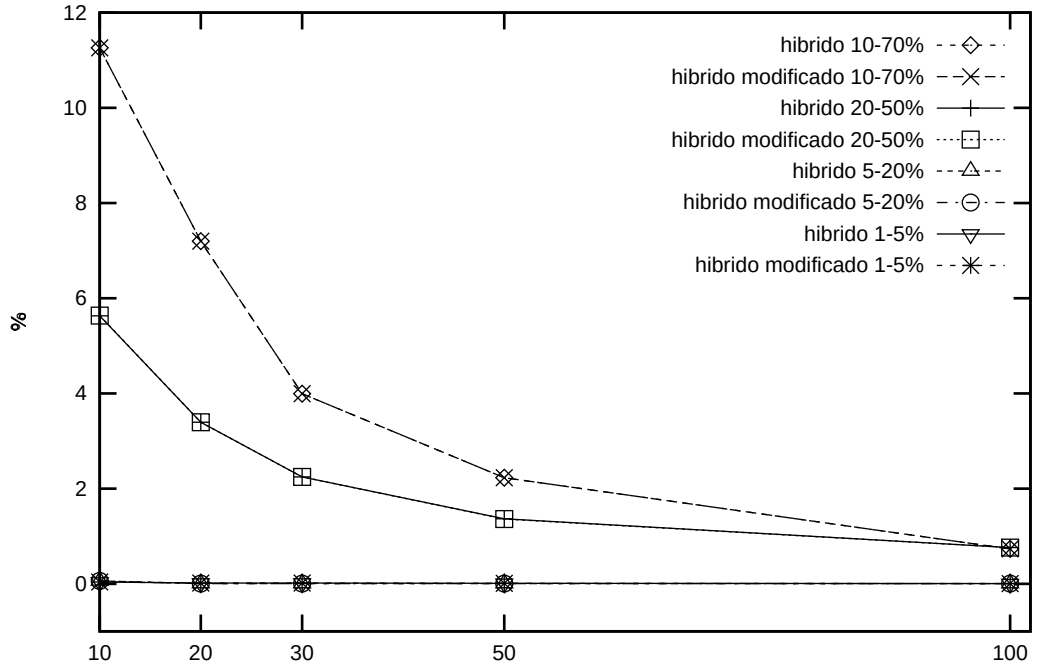

Figura 4.8: Desperdício médio obtido ao resolver as instâncias estratificadas.

4.2 Resultados dos Testes Práticos

Apresentamos nesta seção os resultados obtidos ao aplicar os algoritmos híbridos propostos a cerca de duas centenas de instâncias reais tiradas das plantas de ferragem de duas obras de uma construtora com atuação em várias cidades do país. Tratam-se de obras de construção de edifícios gêmeos de 9 andares, construídos lado a lado. As instâncias consideradas são relativas ao problema de determinar como cortar o aço a ser utilizado nas estruturas de concreto armado de modo a minimizar o desperdício de aço.

Nestas obras foi utilizado aço CA-60B com 5mm de bitola, e aço CA-50A com bitolas 6,3mm, 8mm, 10mm, 12,5mm, 16mm e 20mm. Dessa forma houve necessidade de utilizar 7 tipos de vergalhões de aço, originando 7 categorias de problemas. Obtivemos 209 instâncias, que chamaremos de *instâncias práticas*, agrupando a demanda de cada um destes 7 tipos de vergalhões em cada uma das 42 plantas de ferragem analisadas. A Tabela 4.6 mostra os resultados obtidos ao resolver estas instâncias usando o algoritmo HÍBRIDOm.

Bitola	Nº de problemas	m médio	Tamanho dos itens	IRUP	Colunas geradas	Tempo (seg)	Ganho sobre FFD	Desperdício
5.0	13	7	2%-50%	13	14,15	0,23	0,945%	0,324%
6.3	43	11	2%-98%	43	22,00	0,30	1,868%	1,438%
8.0	40	12	2%-90%	40	28,27	0,35	1,128%	6,281%
10.0	35	20	4%-92%	35	70,29	0,94	0,825%	7,843%
12.5	36	14	6%-92%	36	34,47	0,53	2,207%	5,467%
16.0	26	13	16%-75%	25	24,92	0,35	1,247%	20,675%
20.0	16	8	20%-74%	16	8,81	0,12	0,894%	12,023%

Tabela 4.6: Resultados obtidos pelo algoritmo HÍBRIDOm aplicado às instâncias práticas.

Analisando a Tabela 4.6, percebemos que o algoritmo HÍBRIDOm encontrou, dentro de um tempo bastante curto, soluções ótimas para todas as instâncias, menos uma. A IRUP foi verificada em 208 instâncias e na instância restante verificou-se a MIRUP. Resolvendo esta última instância com o algoritmo HÍBRIDO verificamos que a IRUP também é válida para ela. Além disso, o algoritmo HÍBRIDOm apresentou um ganho em relação ao FFD bastante satisfatório. O tempo médio gasto pelo algoritmo FFDWB ao resolver estas 209 instâncias foi inferior a 0,01 segundos.

Por outro lado, nas categorias onde os itens têm tamanho médio ou grande (bitolas 16mm e 20mm), apesar de termos encontrado soluções ótimas para todas as instâncias, o desperdício médio foi bastante alto (acima de 10%). Observando a primeira linha da Tabela 4.4, percebemos

que já poderíamos esperar por isto. Este resultado é explicado, em parte, pelo fato de que as instâncias nestas categorias apresentam poucos itens (em média, $m \leq 13$).

Com o intuito de diminuir o desperdício, experimentamos agrupar todas as instâncias de cada categoria gerando apenas 7 instâncias, que chamaremos de *instâncias práticas agrupadas*. Com isto obtivemos instâncias maiores, esperando obter soluções com menor desperdício. Eventualmente este procedimento pode não ser aplicável na prática, pois exigiria cortar de uma só vez todo o aço a ser utilizado na obra. Os resultados obtidos pelo algoritmo HÍBRIDOm ao resolver estas 7 instâncias são apresentados na Tabela 4.7.

Bitola	m	Tamanho dos itens	IRUP	Colunas geradas	Tempo (seg)	Ganho sobre FFD	Desperdício
5.0	48	2%-50%	Sim	118	42	0,607%	0,020%
6.3	209	2%-98%	Sim	622	40	0,474%	0,017%
8.0	264	2%-90%	Sim	2706	200	1,278%	0,009%
10.0	365	4%-92%	Sim	10758	1601	1,365%	0,061%
12.5	301	6%-92%	Sim	7954	484	1,481%	0,268%
16.0	218	16%-75%	Sim	1215	169	1,673%	4,995%
20.0	105	20%-74%	Sim	573	32	0,489%	2,507%

Tabela 4.7: Resultados obtidos pelo algoritmo HÍBRIDOm aplicado às instâncias práticas agrupadas.

Soluções ótimas foram encontradas para todas as 7 instâncias. É interessante notar também que, mesmo para instâncias que podemos considerar grandes, o algoritmo HÍBRIDOm apresentou tempo de execução aceitável. O tempo requerido pelo FFDWB para resolver cada uma destas 7 instâncias foi inferior a 0,01 segundos.

Na construção civil, o aço utilizado nas estruturas de concreto armado é adquirido por peso. Por isso, para ter uma idéia mais aproximada do desperdício, precisamos levar em conta o peso dos vergalhões de aço nas suas diferentes bitolas. A Tabela 4.8 apresenta a quantidade de aço especificada nas plantas e as quantidades utilizadas nas soluções apresentadas nas tabelas 4.6 e 4.7.

Vale salientar que, em obras deste tipo, esta construtora desperdiça, em média, algo em torno de 10% do aço. Este percentual é baixo se comparado ao desperdício médio verificado na indústria nacional de construção civil. Agrupando as instâncias, reduzimos o desperdício de aço a cerca de 1%, o que representa uma economia considerável.

A Tabela 4.9 apresenta uma comparação entre os resultados obtidos pelos algoritmos híbridos

aqui propostos e o algoritmo FFD, quando aplicados às instâncias estratificadas e às instâncias práticas. A coluna *Instâncias Práticas* da Tabela 4.9 inclui as instâncias práticas e as instâncias práticas agrupadas. Verificamos que o algoritmo HÍBRIDOm foi o que encontrou soluções ótimas para o maior número de instâncias.

Bitola	Peso (kg/m)	Quantidade de aço nas plantas (kg)	Tabela 4.6		Tabela 4.7	
			Quantidade de aço (kg)	Desperdício	Quantidade de aço (kg)	Desperdício
5.0	0,16	5062,008	5078,400	0,324%	5063,040	0,020%
6.3	0,25	15834,380	16062,000	1,438%	15837,000	0,017%
8.0	0,40	24415,352	25948,800	6,281%	24417,600	0,009%
10.0	0,63	23799,497	25666,200	7,843%	23814,000	0,061%
12.5	1,00	24235,060	25560,000	5,467%	24300,000	0,268%
16.0	1,60	15305,872	18470,400	20,675%	16070,400	4,995%
20.0	2,50	17969,525	20130,000	12,023%	18420,000	2,507%
Totais		126621,695	136915,800	8,130%	127922,040	1,027%

Tabela 4.8: Desperdício de aço nas soluções encontradas para as instâncias práticas.

Os resultados obtidos, tanto nos testes realizados com instâncias geradas aleatoriamente, quanto nos testes com instâncias práticas, demonstraram a eficiência dos algoritmos híbridos propostos em termos de qualidade da solução e tempo de execução. Ademais, os resultados dos testes serviram para fortalecer a conjectura MIRUP.

Algoritmo	$m = 10$	$m = 20$	$m = 30$	$m = 50$	$m = 100$	Instâncias Práticas	Total
Total de instâncias	400	400	400	400	400	216	2216
HÍBRIDOm	399	400	398	395	391	215	2198
HÍBRIDO	400	400	398	385	333	216	2132
FFD	57	33	22	13	5	91	221

Tabela 4.9: Soluções inteiras ótimas obtidas pelos algoritmos FFD, HÍBRIDO e HÍBRIDOm aplicados às instâncias estratificadas e às instâncias práticas.

Capítulo 5

Uma Nova Variante do Problema de Corte Unidimensional

Visando conhecer de perto o processo de corte do aço utilizado nas estruturas de concreto armado, visitamos construtoras e canteiros de obra e conversamos com engenheiros, mestres-de-obra e ferreiros. Conhecendo mais a fundo o processo de corte manual do aço, percebemos não ser desejável que a solução do problema de corte unidimensional associado ao processo de corte apresente padrões com um grande número de itens distintos.

No corte manual do aço, os padrões devem conter poucos itens diferentes, digamos no máximo 4 ou 5, pois do contrário os ajustes necessários no equipamento utilizado para o corte acabariam por tornar a solução desinteressante e, eventualmente, impraticável.

Dessa forma resolvemos estudar o problema de corte unidimensional impondo uma nova restrição: *o número de itens distintos que podem ocorrer num padrão viável é limitado por uma constante inteira* δ. É fácil perceber que esta restrição faz com que o número de padrões viáveis esteja limitado por $\lfloor \frac{L}{l_{min}} \rfloor^{\delta}$, onde L é o comprimento da barra e l_{min} o comprimento do menor item. Como desejamos que δ assuma valores pequenos (no máximo 5), esta restrição limita bastante o número de padrão viáveis.

Podemos então esperar que, com esta restrição, os problemas possam ser resolvidos mais rapidamente, com o ônus de uma queda considerável na qualidade das soluções encontradas. Veremos no entanto que, empiricamente, estas duas expectativas não se confirmam. A seguir descrevemos as modificações que devem ser feitas de modo a incorporar esta nova restrição aos algoritmos híbridos propostos no Capítulo 3.

5.1 Adaptando os Algoritmos

Basicamente precisamos evitar que padrões com mais do que δ itens distintos sejam gerados. Padrões são gerados no passo 1 do algoritmo SimplexGC, e nos algoritmos MOCHILA, FFDe e FFDWB. Este último algoritmo usa o algoritmo MOCHILAe como subrotina.

No passo 1 do algoritmo simplexGC os padrões gerados possuem apenas um item. Precisamos então adaptar apenas os algoritmos MOCHILA, MOCHILAe, FFDe e FFDWB, obtendo os algoritmos MOCHILA$_\delta$, MOCHILAe_δ, FFDe_δ e FFDWB$_\delta$, respectivamente.

As adaptações necessárias nos algoritmos SimplexGC, HÍBRIDO e HÍBRIDOm se resumem a utilizar os algoritmos MOCHILA$_\delta$, MOCHILAe_δ, FFDe_δ e FFDWB$_\delta$.

5.1.1 O Algoritmo MOCHILA$_\delta$

Na Subseção 3.1.1 discutimos o algoritmo MOCHILA. Deve ter ficado claro para o leitor que tal algoritmo executa uma busca em profundidade numa árvore, sendo que a solução encontrada é um padrão viável descrito por um ramo da árvore, desde a raiz até uma folha. Cada nível desta árvore está associado a um item, e o valor de cada nó indica quantas vezes o item associado ao nível em que aquele nó se encontra deve ser cortado no padrão.

Portanto, o que precisamos fazer para incorporar a nova restrição ao algoritmo é, em cada ramo da árvore, limitar em δ o número de nós cujo valor é maior que zero. Isto pode ser facilmente implementado modificando os passos 2 e 3.2 do algoritmo MOCHILA. Introduzimos a variável d que representará o número de nós no ramo corrente com valor maior que zero. No passo 2 fazemos $d = 0$ e $j = 0$. A seguir calculamos a primeira solução fazendo, enquanto $j < m$ e $d < \delta$, $z_j = \lfloor (L - \sum_{i=1}^{j-1} l_i z_i)/l_j \rfloor$, $j = j + 1$, e se $z_j > 0$ então fazemos $d = d + 1$. Como este laço será repetido somente enquanto $d < \delta$, garantimos que a solução inicial gerada possuirá no máximo δ itens distintos.

No passo 3.2 introduzimos uma modificação similar. Fazemos $j = k + 1$, e se $z_k = 0$ então fazemos $d = d - 1$. De modo a atualizar o número de nós com valor maior que zero, para i variando de $k + 1$ até m, se $z_i > 0$ fazemos $d = d - 1$. Calculamos o restante do ramo fazendo, enquanto $j < m$ e $d < \delta$, $z_j = \lfloor (L - \sum_{i=1}^{j-1} l_i z_i)/l_j \rfloor$, $j = j+1$, e se $z_j > 0$ então fazemos $d = d+1$. O fato deste laço também se repetir somente enquanto $d < \delta$, garante que as soluções geradas pelo algoritmo possuirão no máximo δ itens distintos.

Eventualmente a solução encontrada pelo algoritmo MOCHILA$_\delta$ apresentará comprimento não utilizado maior que l_{min}. Tais soluções poderiam fazer o método de geração de colunas parar antes de encontrar uma solução ótima do problema relaxado. No entanto os testes que apresentaremos na próxima seção indicam que, mesmo para valores pequenos de δ, digamos 4

ou 5, as soluções encontradas pelo algoritmo não comprometem, na maioria dos casos, o método de geração de colunas.

A seguir apresentamos o algoritmo MOCHILA$_\delta$ que incorpora as modificações explicadas anteriormente.

Algoritmo MOCHILA$_\delta$

Entrada: $(L, l_1, \ldots, l_m, y_1, \ldots, y_m, \delta)$.
Saída: $z_1, \ldots, z_m \in I\!N$ tais que $\sum_{i=1}^{m} l_i z_i \leq L$, $\sum_{i=1}^{m} y_i z_i$ é máximo e a cardinalidade do conjunto $\{z_i \mid z_i > 0 \ (i = 1, ..., m)\}$ é menor ou igual a δ.

1 Ordene os itens em ordem decrescente de custo relativo $(\frac{y_i}{l_i})$.
2 Faça $d = 0$ e $j = 0$.
  2.1 Enquanto $j < m$ e $d < \delta$ faça $z_j = \lfloor (L - \sum_{i=1}^{j-1} l_i z_i)/l_j \rfloor$, $j = j + 1$, e se $z_j > 0$ então faça $d = d + 1$.
  2.2 Faça $z^* = z$ e $M = \sum_{i=1}^{m} y_i z_i^*$.
3 Faça $k = max\{i \mid z_i > 0 \ e \ \sum_{j=1}^{i} y_j \bar{z}_j \ + \ \frac{y_{i+1}}{l_{i+1}}(L - \sum_{j=1}^{i} l_j \bar{z}_j) > M \ (i = m-1, \ldots, 1)\}$.
  3.1 Se não existe tal k então devolva z^* e pare.
  3.2 Caso contrário, faça $z_k = z_k - 1$ e $j = k + 1$. Se $z_k = 0$ faça $d = d - 1$.
    3.2.1 Para $i = k + 1, \ldots, m$ faça se $z_i k > 0$ então $d = d - 1$.
    3.2.2 Enquanto $j < m$ e $d < \delta$ faça $z_j = \lfloor (L - \sum_{i=1}^{j-1} l_i z_i)/l_j \rfloor$, $j = j + 1$, e se $z_j > 0$ então faça $d = d + 1$.
4 Se $M \leq \sum_{i=1}^{m} y_i z_i$ faça $z^* = z$ e $M = \sum_{i=1}^{m} y_i z_i^*$.
5 Retorne ao passo 3.

5.1.2 O Algoritmo MOCHILAe_δ

O algoritmo MOCHILAe é utilizado como subrotina pelo algoritmo FFDWB. O algoritmo FFDWB basicamente executa uma busca em profundidade numa árvore, sendo que a solução encontrada é um ramo da árvore, desde a raiz até uma folha, onde cada nó representa um padrão viável. Cada vez que um padrão é gerado, ou seja, ao visitar cada nó da árvore, o algoritmo MOCHILAe é utilizado para estimar o desperdício inerente aos ramos que passam por aquele nó. Esta estimativa permite determinar *a priori* se aquele nó pode participar da solução desejada. Em caso negativo, podamos a árvore efetuando um retrocesso. Precisamos então adaptar o algoritmo MOCHILAe de modo a fazê-lo considerar apenas os padrões que obedecem a restrição proposta neste capítulo.

Podemos adaptar o algoritmo MOCHILAe efetuando as mesmas modificações explicadas na subseção anterior. O algoritmo MOCHILAe_δ, apresentado a seguir, incorpora estas modificações.

Algoritmo MOCHILAe_δ

Entrada: $(L, l_1, \ldots, l_m, y_1, \ldots, y_m, d_1, \ldots, d_m, \delta)$.
Saída: $z_1, \ldots, z_m \in I\!N$ tais que $\sum_{i=1}^{m} l_i z_i \leq L$, $\sum_{i=1}^{m} y_i z_i$ é máximo e a cardinalidade do conjunto $\{z_i \mid z_i > 0 \ (i = 1, ..., m)\}$ é menor ou igual a δ.

1 Ordene os itens em ordem decrescente de custo relativo $(\frac{y_i}{l_i})$.
2 Faça $d = 0$ e $j = 0$.
  2.1 Enquanto $j < m$ e $d < \delta$ faça $z_j = min(d_j, \lfloor (L - \sum_{i=1}^{j-1} l_i z_i)/l_j \rfloor)$, $j = j + 1$, e se $z_j > 0$ então faça $d = d + 1$.
  2.2 Faça $z^* = z$ e $M = \sum_{i=1}^{m} y_i z_i^*$.
3 Faça $k = max\{i \mid z_i > 0 \ e \ \sum_{j=1}^{i} y_j \bar{z}_j \ + \ \frac{y_{i+1}}{l_{i+1}}(L - \sum_{j=1}^{i} l_j \bar{z}_j) \geq M + 1 \ (i = m - 1, \ldots, 1)\}$.
  3.1 Se não existe tal k então devolva z^* e pare.
  3.2 Caso contrário, faça $z_k = z_k - 1$ e $j = k + 1$. Se $z_k = 0$ faça $d = d - 1$.
    3.2.1 Para $i = k + 1, \ldots, m$ faça se $z_i k > 0$ então $d = d - 1$.
    3.2.2 Enquanto $j < m$ e $d < \delta$ faça $z_j = min(d_j, \lfloor (L - \sum_{i=1}^{j-1} l_i z_i)/l_j \rfloor)$, $j = j + 1$, e se $z_j > 0$ então faça $d = d + 1$.
4 Se $M \leq \sum_{i=1}^{m} y_i z_i$ faça $z^* = z$ e $M = \sum_{i=1}^{m} y_i z_i^*$.
5 Retorne ao passo 3.

5.1.3 O Algoritmo SimplexGC$_\delta$

Adaptamos o algoritmo simplexGC simplesmente substituindo a chamada ao algoritmo MOCHILA por uma chamada ao algoritmo MOCHILA$_\delta$.

Algoritmo SimplexGC$_\delta$

Entrada: $(L, l_1, \ldots, l_m, d_1, \ldots, d_m, \delta)$.
Saída: Uma solução ótima de: $\min \sum_{j=1}^{n} x_j$
sujeito a $Ax = d$
$x_j \geq 0 \qquad j = 1, \ldots, n,$
onde cada coluna j de A é tal que $\sum_{i=1}^{m} a_{ij} l_i \leq L$ e possui no máximo δ elementos não nulos.

1 Para $i = 1$ até m faça.
  1.1 Para $j = 1$ até m se $i = j$ então faça $a_{ij} = \lfloor \frac{L}{l_i} \rfloor$; caso contrário, faça $a_{ij} = 0$.
2 Faça $x_i = \frac{d_i}{a_{ii}}$ $(i = 1, \ldots, m)$.
3 Resolva $yA = c$.
4 Execute o algoritmo MOCHILA$_\delta$ com parâmetros $L, l_1, \ldots, l_m, y_1, \ldots, y_m, \delta$.
5 Se $\sum_{i=1}^{m} y_i z_i \leq 1$, retorne x e pare.
6 Caso contrário, resolva $Ab = z$.
7 Calcule $t = min\{\frac{x_i}{b_i} \mid b_i > 0 \ (i = 1, \ldots, m)\}$ e $s = min\{i \mid \frac{x_i}{b_i} = t \ (i = 1, \ldots, m)\}$.
8 Para $i = 1$ até m faça $a_{is} = z_i$ $(i = 1, \ldots, m)$.
  8.1 Se $i = s$ então faça $x_i = t$; caso contrário, faça $x_i = x_i - b_i t$.
9 Retorne ao passo 3.

5.1.4 O Algoritmo FFDe_δ

Vimos, na Subseção 3.2.2, que o algoritmo FFD*e* é um algoritmo padrão orientado, ou seja, é um algoritmo em que primeiramente geramos os padrões e depois decidimos quantas vezes utilizá-los. Esta característica facilita sobremaneira adaptar este algoritmo à restrição ora proposta.

Para adaptar o algoritmo FFD*e* precisamos apenas, ao gerar cada padrão, contar o número de itens utilizados no padrão, restringindo este número a δ. Isto pode ser implementado introduzindo três pequenas alterações nos passos 1, 2.2 e 2.3 do algoritmo. No passo 1 inicializamos com zero a variável d, que representará o número de itens distintos cortados no padrão até o momento. No passo 2.2, além de verificar se existe o j que procuramos no passo 2.1, verificamos também se $d \leq \delta$. Em caso afirmativo fazemos $d = d + 1$. Finalmente, ao terminar de gerar cada padrão, no passo 2.3, fazemos $d = 0$ novamente. A seguir apresentamos o algoritmo FFD especializado$_\delta$.

Algoritmo FFDe_δ

Entrada: $(L, l_1, \ldots, l_m, d_1, \ldots, d_m, \delta)$.
Saída: Uma solução com no máximo δ itens distintos em cada padrão.

1 Ordene $l_1, \ldots, l_m$ em ordem decrescente, faça $k = 1$, $\wp_k = \emptyset$ e $d = 0$.
2 Repita até que $d_i = 0$ $(i = 1, \ldots, m)$.
  2.1 Procure $j = min\{h \mid l_h \leq c(\wp_k)\ (h = 1, \ldots, m)\}$.
  2.2 Se existir tal j e $d \leq \delta$ então faça $d = d + 1$, $f = min(d_j, \lfloor \frac{c(\wp_k)}{l_j} \rfloor)$ e empacote f vezes o item j em $\wp_k$, fazendo f vezes $\wp_j = \wp_j \cup \{i\}$.
  2.3 Caso contrário, faça $d = 0$ e $r_k = min\{\lfloor \frac{d_i}{f_i(\wp_k)} \rfloor,\ i \in \wp_k\}$.
    2.3.1 Para todo $i \in \wp_k$ faça $d_i = d_i - f_i(\wp_k) r_k$. Faça $k = k + 1$ e $\wp_k = \emptyset$.
3 Retorne $\wp_1, \ldots, \wp_k$ e $r_1, \ldots, r_k$ e pare.

5.1.5 O Algoritmo FFDWB$_\delta$

No algoritmo FFDWB os padrões são gerados no procedimento PACKING. Precisamos então efetuar modificações apenas neste procedimento de modo a adequá-lo, e consequentemente também o FFDWB, à nova restrição. A natureza recursiva do procedimento PACKING facilita bastante sua adaptação.

Incluímos na chamada do procedimento os parâmetros δ e d, que representam o número máximo de itens distintos que podem ocorrer num padrão e o número de itens já cortados no padrão corrente, respectivamente. É suficiente alterar os passos 3.1, 3.4, 5.1 e 5.2.

No passo 3.1, se existir item com demanda maior que zero que caiba em $\wp_k$, ou seja, se existir i' que procuramos no passo 3, verificamos se d é estritamente menor que δ. Em caso afirmativo, podemos empacotar mais itens no padrão; portanto calculamos $f = min(d_{i'}, \lfloor \frac{c(\wp_k)}{l_{i'}} \rfloor)$ e desviamos a execução para o passo 5. Do contrário, apesar de o padrão ainda deixar espaço para mais itens, não podemos aproveitar este espaço para que d não ultrapasse δ. Isto indica que a chamada ao procedimento mostrou-se infrutífera, e por isso desviamos o fluxo de execução para o passo 4.

No passo 3.4, incluímos na chamada ao procedimento PACKING os parâmetros δ e 0, visto que o padrão $\wp_{k+1}$ ainda está vazio. No passo 5.1, executamos PACKING$_\delta(D, k, f', i', \delta, d)$, se $f' = 0$; caso contrário, executamos PACKING$_\delta(D, k, f', i', \delta, d+1)$. Se a chamada recursiva feita no passo 5.1 resultar numa solução, o procedimento pára. Se não, como a chamada recursiva mostrou-se infrutífera, no passo 5.2, atualizamos a demanda fazendo $d_{i'} = d_{i'} + f'$ e removemos de $\wp_k$ as f' ocorrências do item de comprimento $l_{i'}$.

As modificações propostas nos dois parágrafos anteriores são suficientes para adaptar o procedimento PACKING à nova restrição. No entanto, podemos efetuar mais algumas mudanças de modo a melhorar a performance do procedimento.

Na fase de testes verificamos que, ao contrário do que poderíamos esperar, a nova restrição imposta ao problema faz com que os algoritmos vistos nas subseções 5.1.1 e 5.1.2, tenham muito mais dificuldade em achar uma solução para o problema da mochila. Isto pode parecer estranho pois a nova restrição limita o espaço de busca ao reduzir o número de padrões viáveis. No entanto, investigando mais profundamente este fenômeno, percebemos que ele pode ser explicado pelo seguinte fato: ao resolver o problema sem a restrição, apesar da árvore de busca ser maior, a solução é *quase sempre* encontrada nos primeiros ramos percorridos. Por outro lado, ao resolvermos o problema com a nova restrição precisamos percorrer uma parte significativamente maior da árvore.

Não é possível deixar de usar o algoritmo MOCHILA$_\delta$ no método de geração de colunas. Por outro lado, podemos deixar de utilizar o algoritmo MOCHILAe_δ, visto que este algoritmo é utilizado no procedimento PACKING$_\delta$ apenas como um dos testes usados na poda da árvore, podendo então ser descartado.

Podemos incluir ainda um novo teste para ser utilizado na poda da árvore. Ao gerarmos um novo padrão, calculamos R, que representa o número de itens remanescentes, ou seja, o número de itens com demanda maior que zero. Para isto, fazemos $R = 0$, no passo 3.2. Em seguida, no passo 3.3, para $i = l, \ldots, m$, fazemos $R = R + 1$ se $d_i > 0$. É necessário que $\delta \geq \lceil \frac{R}{C-k} \rceil$, senão pelo menos um dos padrões que ainda serão gerados terá que possuir mais do que δ itens distintos. Incluímos portanto mais esta condição no passo 3.4.

Como visto anteriormente, uma das condições que tem de ser satisfeita para aceitarmos um padrão é $c(\wp_k) \leq \lceil \frac{D_k}{C-k+1} \rceil$. Esta condição determina que o desperdício no padrão gerado ($c(\wp_k)$) não pode exceder o desperdício médio aceitável no restante da solução (incluso o padrão gerado). Ao construir uma solução, ou seja, ao percorrer um ramo da árvore, optamos por começar pelos padrões que apresentam desperdício menor, permitindo um desperdício maior à medida que descemos no ramo da árvore. Conforme vimos na Subseção 3.2.3, tal procedimento não nos impede de encontrar uma solução, se ela existir. A seguir apresentamos o procedimento PACKING$_\delta$.

Procedimento PACKING$_\delta(D_k, k, f, i, \delta, d)$

1 Empacote f vezes o item i em $\wp_k$, fazendo f vezes $\wp_k = \wp_k \cup \{i\}$, e faça $d_i = d_i - f$.
2 Procure $j = min\{h \mid d_h > 0 \ (h = 1, \ldots, m)\}$. Se não existir tal j retorne $\wp_1, \ldots, \wp_k$ e pare.
3 Procure $i' = min\{h \mid d_h > 0 \ e \ l_h \leq c(\wp_k) \ (h = i+1, \ldots, m)\}$.
  3.1 Se existir i' então
    3.1.1 Se $d \leq \delta$ faça $f = min(d_{i'}, \lfloor \frac{c(\wp_k)}{l_{i'}} \rfloor)$ e vá para o passo 5; caso contrário, vá para o passo 4.
  3.2 Faça $R = 0$.
  3.3 Para $i = l$ até m se $d_i > 0$ faça $R = R + 1$.
  3.4 Se $k < C$ e $c(\wp_k) \leq \lfloor \frac{D_k}{C-k+1} \rfloor$ e $\delta \geq \lceil \frac{R}{C-k} \rceil$ então execute PACKING$_\delta(D_k - c(\wp_k), k+1, min(d_j, \lfloor \frac{L}{l_j} \rfloor), j, \delta, 0)$.
4 Faça $f = -1$. {Para que o laço no passo seguinte não seja efetuado}
5 Para $f' = f$ até 0 faça:
  5.1 Se $f' = 0$ execute PACKING$_\delta(D_k, k, f', i', \delta, d)$; caso contrário, execute PACKING$_\delta(D, k, f', i', \delta, d+1)$
  5.2 Faça $d_{i'} = d_{i'} + f'$ e remova de $\wp_k$ as f' ocorrências do item i', fazendo f vezes $\wp_k = \wp_k - \{i'\}$.

Para adaptar o algoritmo FFDWB precisamos apenas modificar o passo 3, fazendo uma chamada ao procedimento PACKING$_\delta$ com os parâmetros $D, 1, f', 1, \delta$ e 0. Obtemos assim o algoritmo FFDWB$_\delta$, descrito abaixo.

Algoritmo FFDWB$_\delta$

Entrada: $(L, l_1, \ldots, l_m, d_1, \ldots, d_m, C, \delta)$.
Saída: Se existir, uma solução inteira de valor no máximo C na qual nenhum padrão contém mais do que δ itens distintos. Caso contrário, o valor 0.

1 Coloque os itens em ordem crescente de $min(d_i, \lfloor \frac{L}{l_i} \rfloor)$ $(i = 1, \ldots, m)$.
2 Faça $D = CL - \sum_{i=1}^{m} l_i d_i$ e $f = min(d_1, \lfloor \frac{L}{l_1} \rfloor)$.
3 Para $f' = f - 1$ até 0 execute PACKING$_\delta(D, 1, f', 1, \delta, 0)$.
4 Retorne 0 e pare.

5.1.6 O Algoritmo HÍBRIDO$_\delta$

O algoritmo HÍBRIDO$_\delta$ resulta do algoritmo HÍBRIDO substituindo-se as chamadas aos algoritmos SimplexGC, FFDWB e FFDe por chamadas aos algoritmos simplexGC$_\delta$, FFDWB$_\delta$ e FFDe_δ, respectivamente.

Algoritmo HÍBRIDO$_\delta$

Entrada: $(L, l_1, \ldots, l_m, d_1, \ldots, d_m, \delta)$.
Saída: Uma solução de: $\min \sum_{j=1}^{n} x_j$
sujeito a $Ax = d$
$x_j \geq 0$ e inteiro $\quad j = 1, \ldots, n$,
onde cada coluna j de A é tal que $\sum_{i=1}^{m} a_{ij} l_i \leq L$ e possui no máximo δ elementos não nulos.

1 Faça $v_{rl} = 0$ e $v_{ip} = 0$.
2 Execute o algoritmo SimplexGC$_\delta$ com parâmetros $L, l_1, \ldots, l_m, d_1, \ldots, d_m, \delta$. Se $\sum_{i=1}^{m} x_i > v_{rl}$ então faça $v_{rl} = \sum_{i=1}^{m} x_i$.
3 Para $i = 1$ até m faça $x_i^* = \lfloor x_i \rfloor$. Faça $v_{ip} = v_{ip} + \sum_{i=1}^{m} x_i^*$
4 Se $x_i^* > 0$ para algum $i = 1, ..., m$ então
  4.1 Retorne A e $x_1^*, \ldots, x_m^*$.
  4.2 Para $i = 1$ até m faça
    4.2.1 Para $j = 1$ até m faça $d_i = d_i - a_{ij} x_j^*$.
  4.3 Faça $m' = 0$.
  4.4 Para $i = 1$ até m faça
    4.3.1 Se $d_i > 0$ faça $m' = m' + 1$, $l_{m'} = l_i$ e $d_{m'} = d_i$.
  4.5 Se $m' = 0$ então pare.
  4.6 Faça $m = m'$ e retorne ao passo 2.
5 Faça $C = \lceil v_{rl} \rceil - v_{ip}$. Execute o FFDWB$_\delta$ com parâmetros $L, l_1, \ldots, l_m, d_1, \ldots, d_m, C, \delta$.
6 Se o algoritmo FFDWB$_\delta$ não retornou 0 então retorne $\wp_1, \ldots, \wp_C$ e pare.
7 Caso contrário, execute o FFDWB$_\delta$ com parâmetros $L, l_1, \ldots, l_m, d_1, \ldots, d_m, C + 1, \delta$.
8 Se o algoritmo FFDWB$_\delta$ não retornou 0 então retorne $\wp_1, \ldots, \wp_{C+1}$ e pare.
9 Caso contrário, execute o algoritmo FFDe_δ com parâmetros $L, l_1, \ldots, l_m, d_1, \ldots, d_m, \delta$, retorne a solução encontrada e pare.

5.1.7 O Algoritmo HÍBRIDOm_δ

De forma análoga ao que realizamos no algoritmo HÍBRIDO obtendo o algoritmo HÍBRIDO$_\delta$, obtemos o algoritmo HÍBRIDOm_δ a partir do algoritmo HÍBRIDOm substituindo as chama-

das aos algoritmos SimplexGC, FFDWB e FFDe por chamadas aos algoritmos SimplexGC$_\delta$, FFDWB$_\delta$ e FFDe_δ, respectivamente.

Algoritmo HÍBRIDOm_δ

Entrada: $(L, l_1, \ldots, l_m, d_1, \ldots, d_m, \delta)$.
Saída: Uma solução de: $\min \sum_{j=1}^{n} x_j$
sujeito a $Ax = d$
$x_j \geq 0$ e inteiro $\quad j = 1, \ldots, n,$
onde cada coluna j de A é tal que $\sum_{i=1}^{m} a_{ij} l_i \leq L$ e possui no máximo δ elementos não nulos.

1 Faça $v_{rl} = 0$ e $v_{ip} = 0$.
2 Execute o algoritmo SimplexGC$_\delta$ com parâmetros $L, l_1, \ldots, l_m, d_1, \ldots, d_m, \delta$.
3 Se $\sum_{i=1}^{m} x_i > v_{rl}$ então faça $v_{rl} = \sum_{i=1}^{m} x_i$.
  3.1 Para $i = 1$ até m se $x_i - \lfloor x_i \rfloor \leq \frac{1}{2}$ faça $x_i^* = \lfloor x_i \rfloor$.
  3.2 Para $k = 1$ até m faça
    3.2.1 Para $i = 1$ até m faça $d_i' = \sum_{j=1}^{k-1} a_{ij} x_j^* + a_{ik} \lceil x_k \rceil + \sum_{j=k+1}^{m} a_{ij} x_j^*$.
    3.2.2 Se $d_i' \leq d_i$ para todo $i = 1, \ldots, m$ então faça $x_k^* = \lceil x_k \rceil$.
    3.2.3 Caso contrário, faça $x_k^* = \lfloor x_k \rfloor$.
  3.3 Faça $v_{ip} = v_{ip} + \sum_{i=1}^{m} x_i^*$
4 Se $x_i^* > 0$ para algum $i = 1, ..., m$ então
  4.1 Retorne A e $x_1^*, \ldots, x_m^*$.
  4.2 Para $i = 1$ até m faça
    4.2.1 Para $j = 1$ até m faça $d_i = d_i - a_{ij} x_j^*$.
  4.3 Faça $m' = 0$.
  4.4 Para $i = 1$ até m faça
    4.3.1 Se $d_i > 0$ faça $m' = m' + 1$, $l_{m'} = l_i$ e $d_{m'} = d_i$.
  4.5 Se $m' = 0$ então pare.
  4.6 Faça $m = m'$ e retorne ao passo 2.
5 Faça $C = \lceil v_{rl} \rceil - v_{ip}$. Execute o FFDWB$_\delta$ com parâmetros $L, l_1, \ldots, l_m, d_1, \ldots, d_m, C, \delta$.
6 Se o algoritmo FFDWB$_\delta$ não retornou 0 então retorne $\wp_1, \ldots, \wp_C$ e pare.
7 Caso contrário, execute o FFDWB$_\delta$ com parâmetros $L, l_1, \ldots, l_m, d_1, \ldots, d_m, C + 1, \delta$.
8 Se o algoritmo FFDWB$_\delta$ não retornou 0 então retorne $\wp_1, \ldots, \wp_{C+1}$ e pare.
9 Caso contrário, execute o algoritmo FFDe_δ com parâmetros $L, l_1, \ldots, l_m, d_1, \ldots, d_m, \delta$, retorne a solução encontrada e pare.

5.2 Resultados dos Testes

Repetimos os mesmos testes descritos no capítulo 4, utilizando $\delta = 5$, $\delta = 4$ e $\delta = 3$. Escolhemos utilizar sempre o algoritmo HÍBRIDOm_δ, visto que o algoritmo HÍBRIDOm mostrou-se mais eficiente, em termos de tempo, que o algoritmo HÍBRIDO, especialmente para instâncias maiores.

Nestes testes, o nosso interesse foi centralizado na avaliação do tempo requerido para resolver os problemas e da qualidade das soluções encontradas. Dessa forma, nas tabelas apresentadas nesta seção, comparamos o tempo dispendido e a qualidade da solução encontrada, com $\delta = m$, $\delta = 5$, $\delta = 4$ e $\delta = 3$. Obviamente, comparamos as soluções encontradas pelo algoritmo HÍBRIDOm_δ com as soluções obtidas pelo FFDe_δ.

5.2.1 Testes Aleatórios

Na Tabela 5.1 comparamos o tempo médio, o ganho médio em relação às soluções encontradas pelo FFDe_δ, e o desperdício médio verificado ao aplicar o algoritmo HÍBRIDOm_δ às instâncias de Wäscher e Gau, com $\delta = m$, $\delta = 5$, $\delta = 4$ e $\delta = 3$.

	Tamanho	Tempo (seg)				Ganho sobre FFDe_δ (em %)				Desperdício (em %)			
m	dos itens	$\delta = m$	$\delta = 5$	$\delta = 4$	$\delta = 3$	$\delta = m$	$\delta = 5$	$\delta = 4$	$\delta = 3$	$\delta = m$	$\delta = 5$	$\delta = 4$	$\delta = 3$
10	0-100%	0,04	0,02	0,04	0,05	0,345	0,345	0,345	0,342	13,734	13,734	13,734	13,740
	0-75%	0,05	0,06	0,07	0,08	0,565	0,565	0,565	0,545	15,089	15,089	15,089	15,120
	0-50%	0,13	0,10	0,17	0,15	2,721	2,714	2,701	2,547	3,098	3,104	3,118	3,324
	0-25%	0,32	0,33	0,39	0,39	1,590	1,550	1,351	0,846	1,388	1,428	1,628	2,347
20	0-100%	0,17	0,14	0,18	0,14	0,273	0,273	0,273	0,266	9,873	9,873	9,873	9,881
	0-75%	0,24	0,23	0,28	0,34	0,639	0,639	0,639	0,624	7,452	7,452	7,452	7,483
	0-50%	0,70	0,89	0,95	0,94	2,319	2,315	2,267	2,039	0,698	0,701	0,756	1,007
	0-25%	1,05	1,31	2,52	5,33	0,864	0,818	0,645	0,150	0,665	0,732	0,964	1,774
30	0-100%	0,43	0,40	0,51	0,41	0,224	0,224	0,225	0,222	10,047	10,047	10,046	10,051
	0-75%	0,69	1,05	0,90	0,83	0,545	0,545	0,541	0,537	6,376	6,376	6,381	6,410
	0-50%	2,67	6,55	3,25	3,12	1,887	1,882	1,855	1,572	0,362	0,366	0,395	0,717
	0-25%	2,39	3,31	4,57	19,75	0,551	0,533	0,281	-0,305	0,451	0,487	0,775	1,661
40	0-100%	0,94	0,95	0,93	0,93	0,197	0,198	0,197	0,195	9,905	9,904	9,905	9,908
	0-75%	1,79	1,96	1,94	1,83	0,631	0,631	0,629	0,619	4,304	4,304	4,306	4,533
	0-50%	6,63	7,08	8,10	7,68	1,453	1,449	1,412	1,140	0,210	0,213	0,253	0,565
	0-25%	4,12	5,83	9,29	34,79	0,435	0,438	0,255	-0,417	0,348	0,374	0,607	1,547
50	0-100%	1,94	1,50	1,93	1,48	0,226	0,225	0,226	0,225	7,176	7,177	7,176	7,178
	0-75%	3,52	4,18	4,86	3,42	0,580	0,579	0,574	0,563	4,401	4,402	4,407	4,423
	0-50%	16,40	19,25	18,39	16,08	1,157	1,147	1,107	0,833	0,157	0,169	0,211	0,524
	0-25%	16,83	11,88	18,64	48,87	0,363	0,374	0,261	-0,477	0,275	0,299	0,497	1,487

Tabela 5.1: Desempenho do algoritmo HÍBRIDOm_δ aplicado às instâncias de Wäscher e Gau, com $\delta = m$, $\delta = 5$, $\delta = 4$ e $\delta = 3$.

Percebemos que, ao contrário do que poderíamos esperar, o tempo médio, de uma forma geral, tende a aumentar um pouco, à medida que δ diminui. No entanto, a Figura 5.1 mostra que este aumento no tempo médio é quase (visualmente) imperceptível para as classes nas quais $\beta \geq 75$. Observamos que este aumento no tempo médio é mais notório nas classes onde $\beta = 25$. A Figura 5.1 ilustra este comportamento.

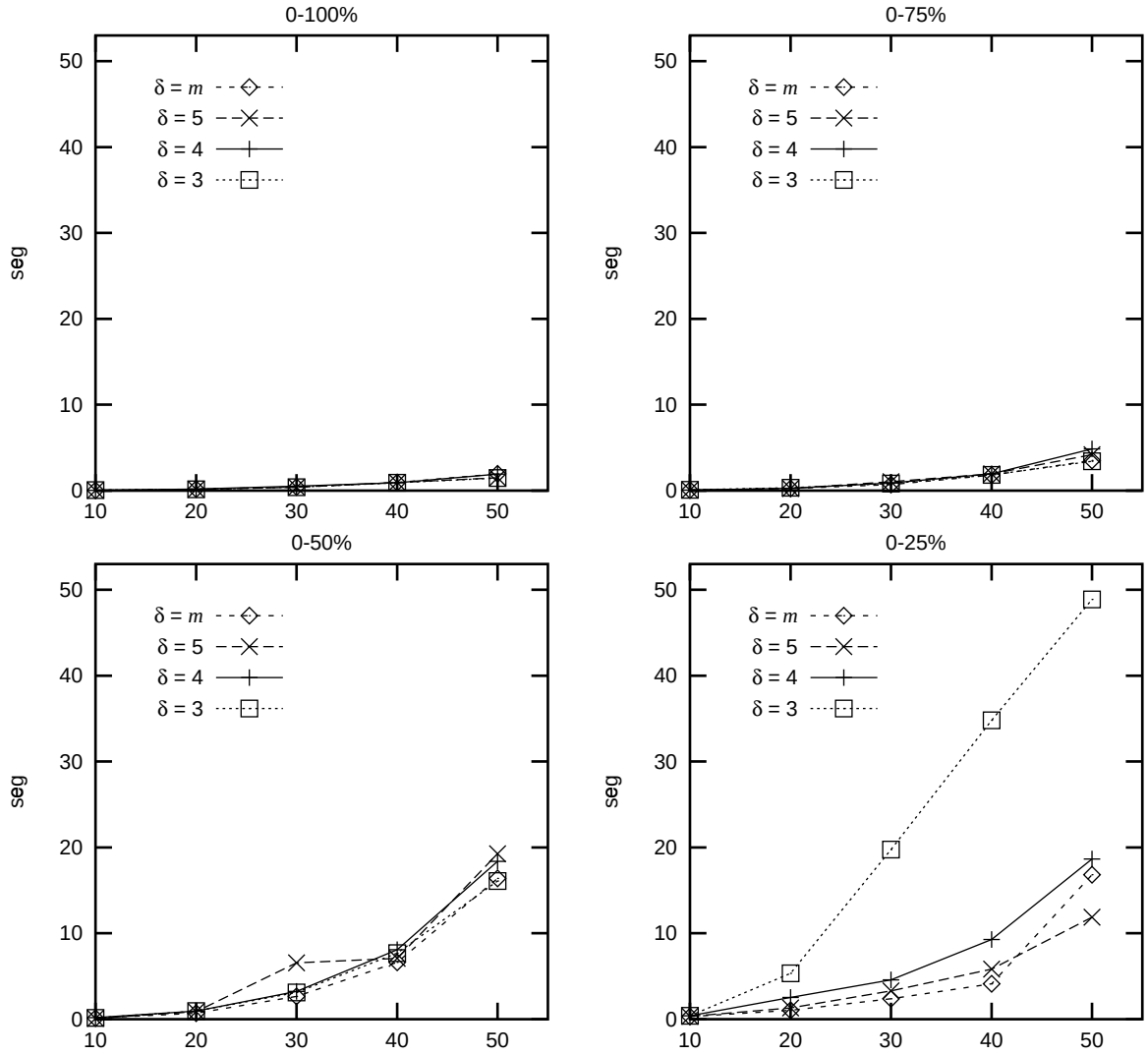

Figura 5.1: Tempo médio requerido pelo algoritmo HÍBRIDOm_δ para resolver as instâncias de Wäscher e Gau, com $\delta = m$, $\delta = 5$, $\delta = 4$ e $\delta = 3$.

Na seção anterior comentamos a explicação para este fenômeno. À medida que δ decresce, o algoritmo MOCHILA$_\delta$ encontra maior dificuldade para achar uma solução, pois torna-se necessário percorrer uma largura muito maior da árvore de busca. Tal árvore, embora fique menor à medida que δ diminui, continua tendo tamanho exponencial (para $\delta \geq 2$). Portanto a busca

acaba por ser um pouco mais demorada.

Verificamos ainda que o tempo médio, em segundos, gasto pelo FFDWB_δ ao resolver as 4000 instâncias é de aproximadamente 0,001, 0,12, 0,26 e 0,83, para $\delta = m$, $\delta = 5$, $\delta = 4$ e $\delta = 3$, respectivamente. Apesar de o algoritmo FFDWB_δ requerer mais tempo à medida que δ diminui, a fase de geração de colunas ainda é responsável pela maior parte do tempo requerido pelo algoritmo HÍBRIDOm_δ.

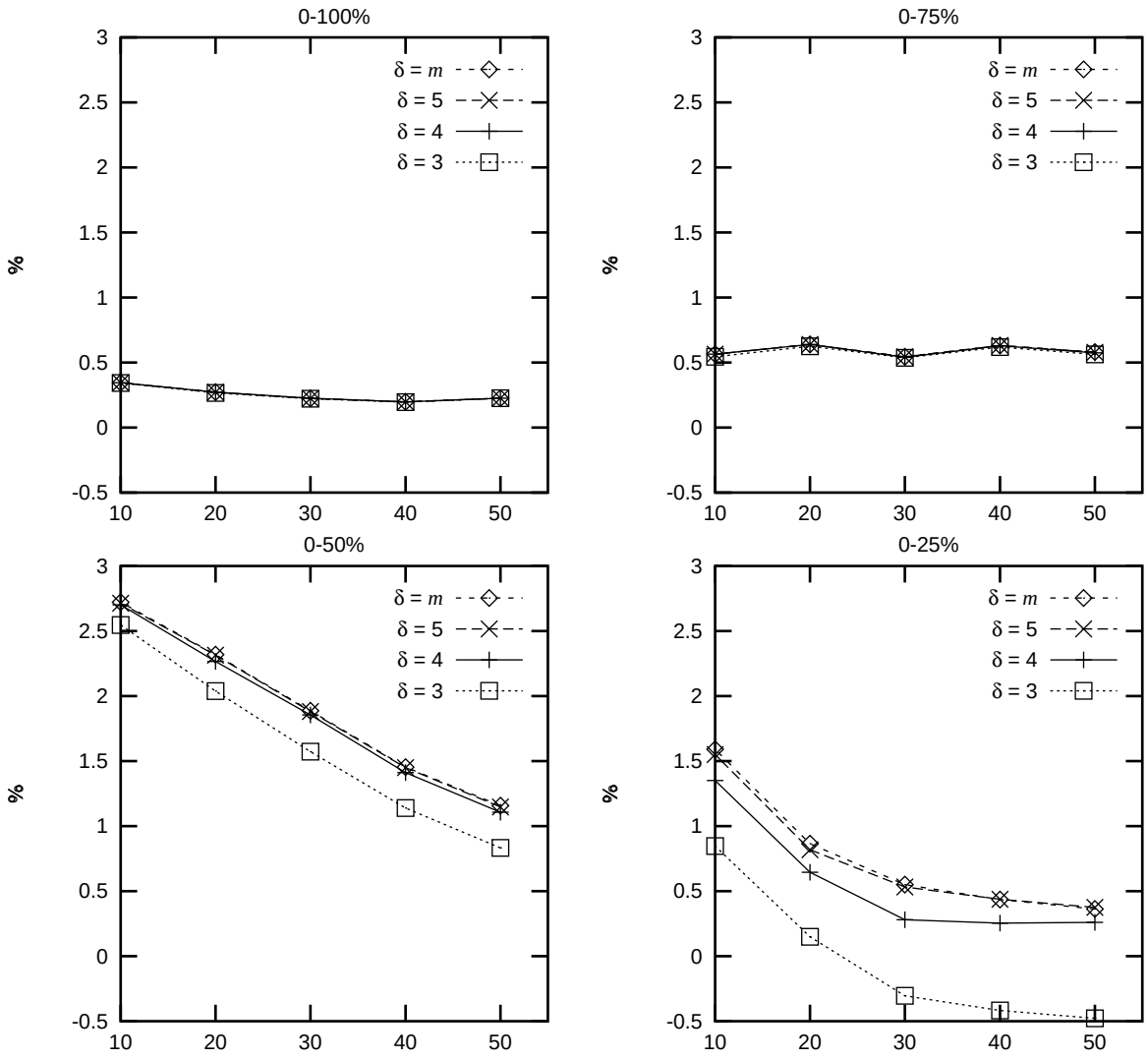

Figura 5.2: Ganho médio em relação ao FFD obtido pelo algoritmo HÍBRIDOm_δ ao resolver as instâncias de Wäscher e Gau, com $\delta = m$, $\delta = 5$, $\delta = 4$ e $\delta = 3$.

Já o ganho médio em relação ao FFD_δ diminui levemente para valores menores de δ. Esta diminuição na qualidade da solução só é mais sensível nas classes onde $\beta \leq 50$. Somente em

algumas poucas instâncias o ganho em relação ao FFD_δ aumentou à medida que δ diminuiu. Algumas vezes isto aconteceu porque o FFD_δ encontrou solução inferior à do FFD. Temos exemplos disso nas classes caracterizadas por ($m = 40$, $\beta = 25$) e ($m = 50$, $\beta = 25$), com $\delta = 5$. Outras vezes o algoritmo HÍBRIDOm_δ encontrou uma solução melhor que o algoritmo HÍBRIDOm. Isto ocorreu e uma das instâncias da classe caracterizada por ($m = 40$, $\beta = 100$), com $\delta = 5$.

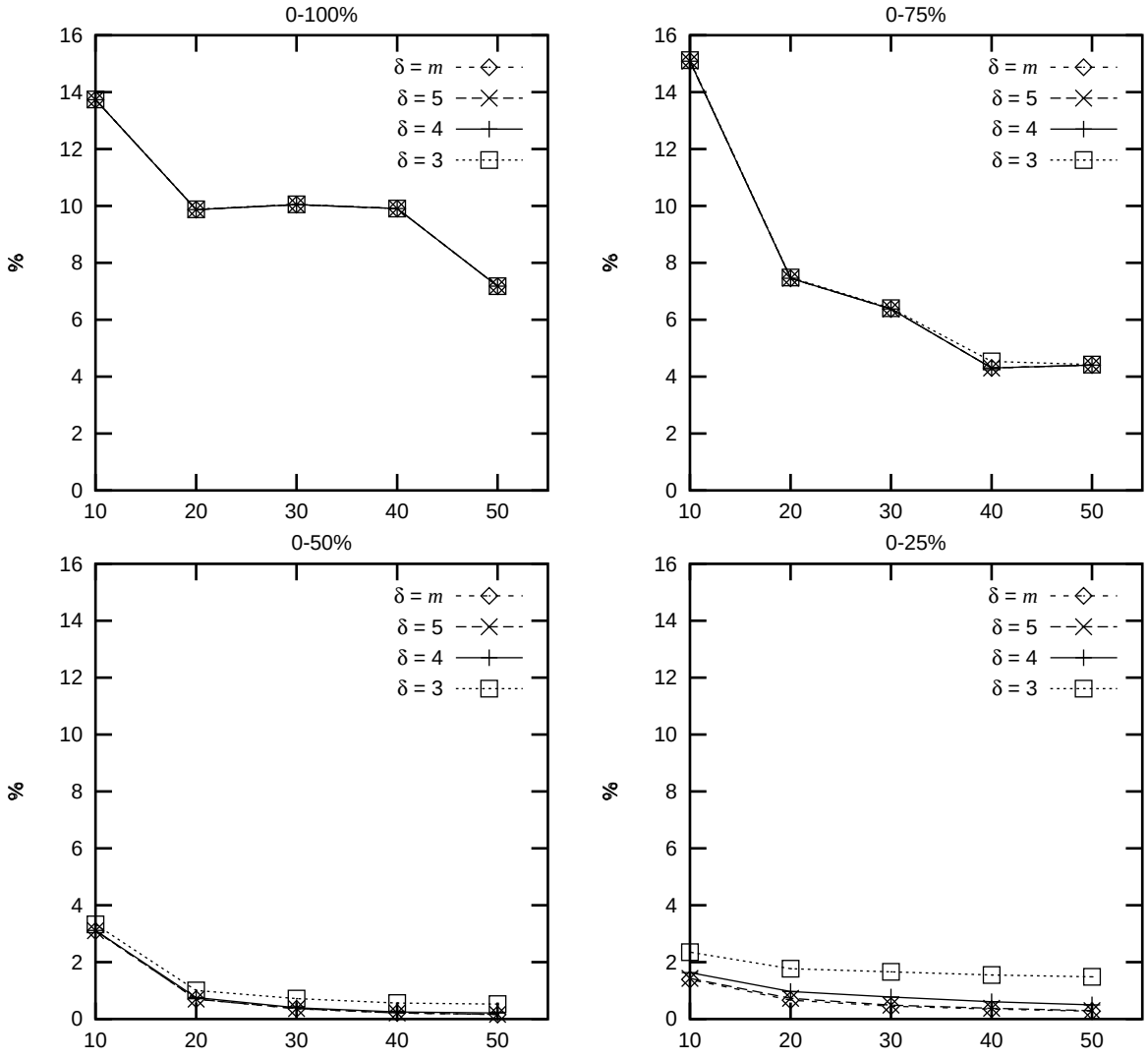

Figura 5.3: Desperdício médio obtido pelo algoritmo HÍBRIDOm_δ ao resolver as instâncias de Wäscher e Gau, com $\delta = m$, $\delta = 5$, $\delta = 4$ e $\delta = 3$.

A Figura 5.2 mostra também que, para as classes em que $\beta \geq 75$, a variação de δ dentro do intervalo [3,m] quase não afeta o desempenho do algoritmo HÍBRIDOm_δ em relação ao FFDe_δ. Observamos ainda que, quando $\delta = 3$, as soluções encontradas pelo algoritmo HÍBRIDOm_δ ao

resolver as classes com $m \geq 30$ e $\beta = 25$ e são, em média, inferiores às soluções encontradas pelo algoritmo FFDe_δ.

Verificamos ainda que, de uma forma geral, o desperdício médio aumentou muito pouco, à medida que δ diminuiu. Para as classes nas quais $\beta \geq 75$, este aumento foi quase (visualmente) imperceptível, como pode ser verificado pela Figura 5.3. Já nas classes com $\beta = 25$, este aumento foi um pouco mais acentuado. Somente em uma instância, pertencente à classe caracterizada por ($m = 40$, $\beta = 100$), o desperdício foi menor quando $\delta = 5$ do que quando $\delta = m$. Isto ocorreu porque, para esta instância, o algoritmo HÍBRIDOm_δ encontrou solução melhor que o algoritmo HÍBRIDOm.

Na Tabela 5.2 comparamos o tempo médio, o ganho médio em relação às soluções encontradas pelo FFDe_δ, e o desperdício médio verificado ao aplicar o algoritmo HÍBRIDOm_δ às instâncias estratificadas, com $\delta = m$, $\delta = 5$, $\delta = 4$ e $\delta = 3$.

	Tamanho	Tempo (seg)				Ganho sobre FFDe_δ				Desperdício			
m	dos itens	$\delta = m$	$\delta = 5$	$\delta = 4$	$\delta = 3$	$\delta = m$	$\delta = 5$	$\delta = 4$	$\delta = 3$	$\delta = m$	$\delta = 5$	$\delta = 4$	$\delta = 3$
10	10%-70%	0,05	0,07	0,04	0,05	0,845	0,845	0,845	0,838	11,258	11,258	11,258	11,265
	20%-50%	0,04	0,06	0,04	0,05	4,642	4,642	4,642	4,642	5,632	5,632	5,632	5,632
	5%-20%	0,34	0,36	0,37	0,35	2,353	2,353	2,344	2,297	0,057	0,057	0,065	0,112
	1%- 5%	0,12	0,16	0,15	0,65	0,500	0,500	0,500	0,492	0,034	0,034	0,034	0,042
20	10%-70%	0,23	0,29	0,30	0,25	1,082	1,082	1,082	1,080	7,198	7,198	7,198	7,205
	20%-50%	0,26	0,25	0,30	0,28	5,867	5,867	5,867	5,867	3,392	3,392	3,392	3,393
	5%-20%	1,72	1,29	1,98	5,85	1,856	1,856	1,854	1,840	0,004	0,004	0,005	0,019
	1%- 5%	0,26	0,30	0,53	3,15	0,347	0,347	0,347	0,344	0,017	0,017	0,017	0,020
30	10%-70%	0,71	0,87	0,76	0,88	1,242	1,242	1,242	1,240	3,992	3,992	3,992	3,995
	20%-50%	0,86	0,89	0,81	0,95	6,379	6,379	6,379	6,377	2,247	2,247	2,247	2,249
	5%-20%	2,11	2,62	3,19	25,71	1,587	1,587	1,585	1,579	0,003	0,003	0,004	0,010
	1%- 5%	0,53	0,53	0,86	3,39	0,284	0,284	0,284	0,283	0,012	0,012	0,012	0,013
50	10%-70%	2,99	3,16	2,50	2,72	1,378	1,378	1,378	1,376	2,227	2,227	2,227	2,229
	20%-50%	4,15	4,31	3,75	4,13	6,661	6,661	6,661	6,660	1,361	1,361	1,361	1,362
	5%-20%	5,12	6,25	9,84	44,50	1,378	1,378	1,377	1,372	0,002	0,002	0,002	0,007
	1%- 5%	2,13	2,35	3,41	9,69	0,237	0,237	0,237	0,234	0,007	0,007	0,007	0,010
100	10%-70%	26,17	24,28	25,49	23,01	1,471	1,471	1,471	1,471	0,731	0,731	0,731	0,732
	20%-50%	51,74	51,67	58,44	54,70	6,854	6,854	6,854	6,854	0,761	0,761	0,761	0,761
	5%-20%	21,89	28,43	64,77	56,92	1,232	1,232	1,231	1,227	0,001	0,001	0,002	0,006
	1%- 5%	42,67	19,83	24,22	26,98	0,203	0,203	0,203	0,199	0,004	0,004	0,004	0,007

Tabela 5.2: Desempenho do algoritmo HÍBRIDOm_δ aplicado às instâncias estratificadas, com $\delta = m$, $\delta = 5$, $\delta = 4$ e $\delta = 3$.

Verificamos que o tempo tende a aumentar à medida que δ diminui, especialmente para as classes de instâncias onde aparecem itens pequenos ($\alpha \leq 5$). Este aumento no tempo médio requerido pelo algoritmo HÍBRIDOm_δ é mais sensível nas classes onde $\beta \leq 50$. A Figura 5.4

ilustra o tempo requerido pelo algoritmo HÍBRIDOm_δ para encontrar a solução, em função de m.

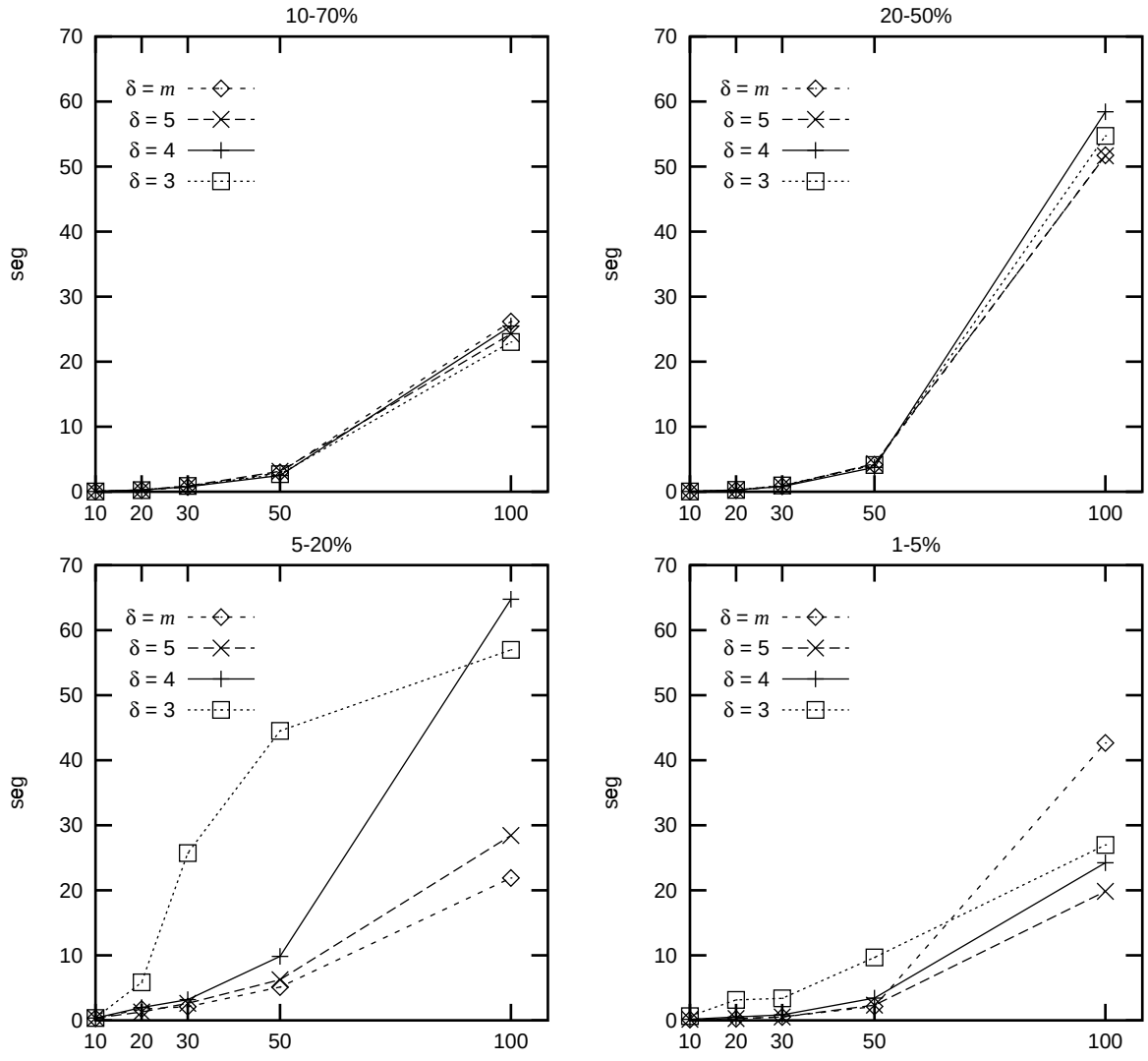

Figura 5.4: Tempo médio requerido pelo algoritmo HÍBRIDOm_δ para resolver as instâncias estratificadas, com $\delta = m$, $\delta = 5$, $\delta = 4$ e $\delta = 3$.

Verificamos ainda que o tempo médio, em segundos, gasto pelo FFDWB$_\delta$ ao resolver as 2000 instâncias é de aproximadamente 0,01, 0,08, 1,21 e 1,73, para $\delta = m$, $\delta = 5$, $\delta = 4$ e $\delta = 3$, respectivamente. Isto significa que a maior parte do tempo requerido para encontrar a solução, foi gasto no procedimento de geração de colunas.

O ganho médio em relação ao FFDe_δ e o desperdício médio permanecem praticamente inalterados em todas as classes de instâncias testadas. Isto pode ser facilmente percebido olhando-se

as figuras 5.5 e 5.6. Apenas nas classes onde $\beta \leq 20$, o desperdício médio em relação ao FFDe_δ aumenta ligeiramente à medida que δ diminui.

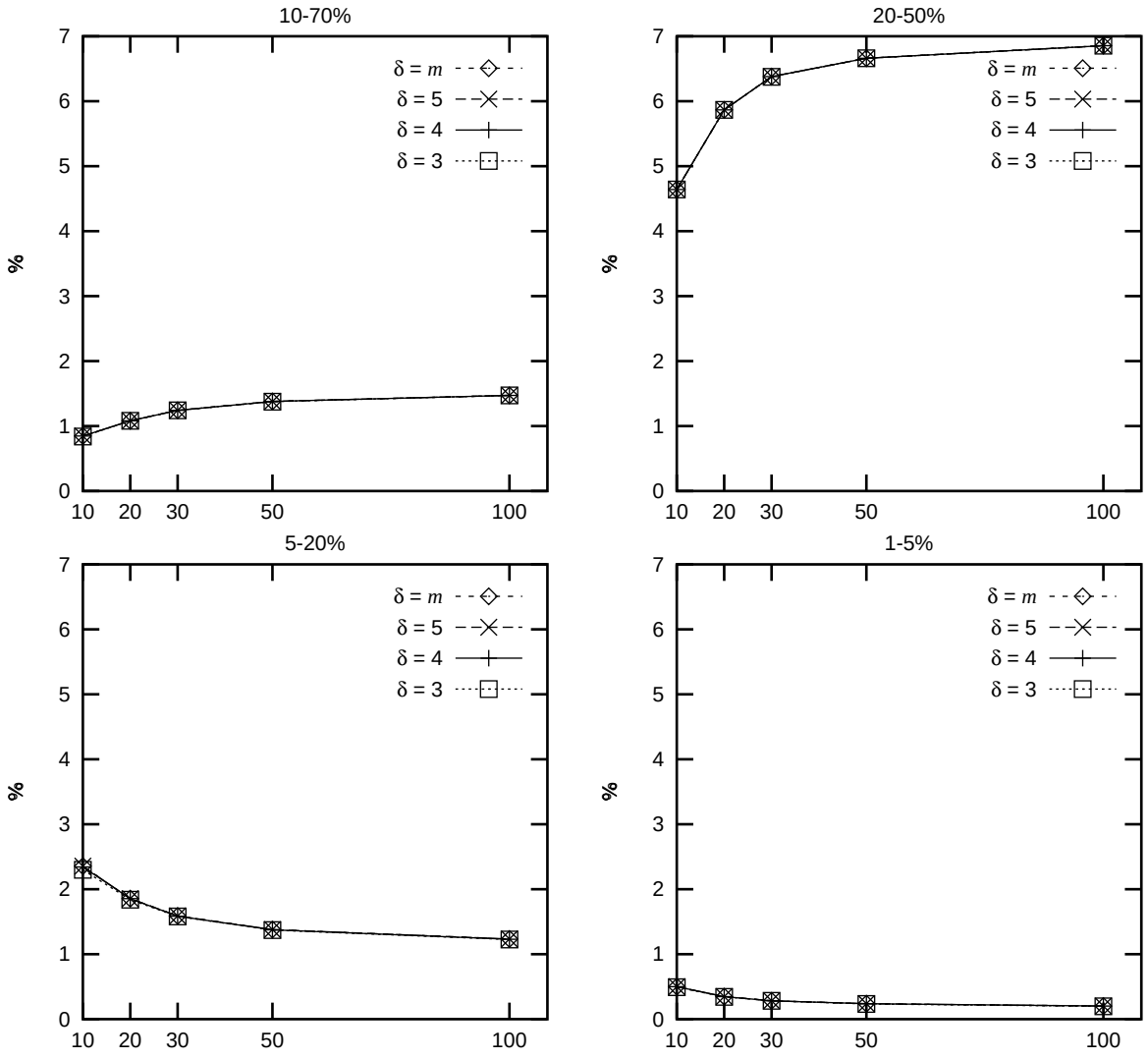

Figura 5.5: Ganho médio em relação ao FFD obtido pelo algoritmo HÍBRIDOm_δ ao resolver as instâncias estratificadas, com $\delta = m$, $\delta = 5$, $\delta = 4$ e $\delta = 3$.

Os testes realizados com estas 6000 instâncias geradas aleatoriamente indicam que o desempenho do algoritmo HÍBRIDOm_δ é pouco afetado mesmo quando δ assume valores pequenos como 3, 4 ou 5, exceto nas classes de instâncias onde item pequenos são muito frequentes ($\beta \leq 25$). Apenas nas classes onde $m \geq 30$ e $\beta = 25$, com $\delta = 3$, o algoritmo HÍBRIDOm_δ mostrou-se pior que o algoritmo FFDe_δ.

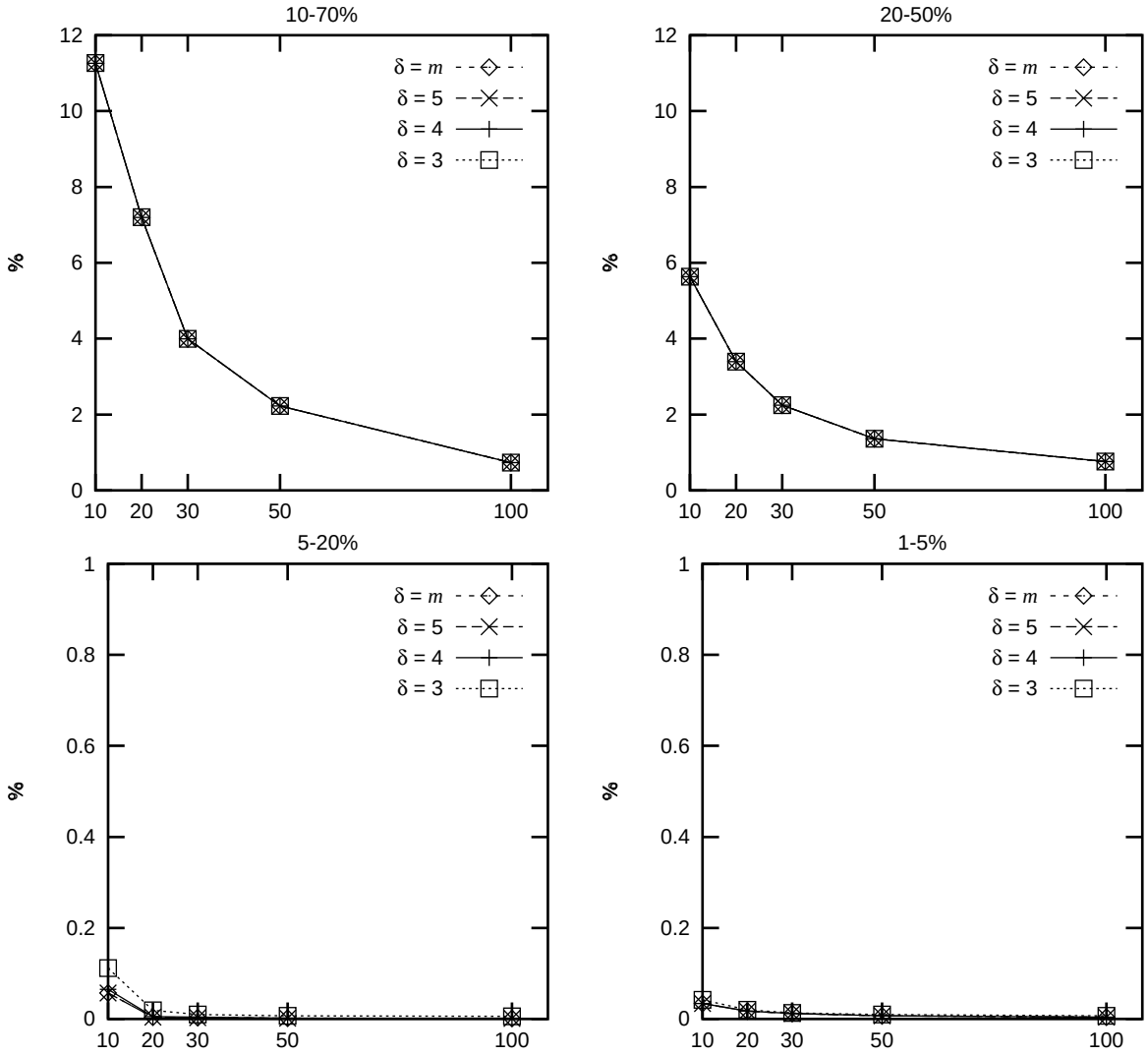

Figura 5.6: Desperdício médio obtido pelo algoritmo HÍBRIDOm_δ ao resolver as instâncias estratificadas, com $\delta = m$, $\delta = 5$, $\delta = 4$ e $\delta = 3$.

5.2.2 Testes Práticos

Na Tabela 5.3 comparamos o tempo médio, o ganho médio em relação às soluções encontradas pelo FFDe_δ, e o desperdício médio verificado ao aplicar o algoritmo HÍBRIDOm_δ às instâncias práticas com $\delta = m$, $\delta = 5$, $\delta = 4$ e $\delta = 3$.

As instâncias práticas foram resolvidas em tempo muito curto, de forma que, devido à imprecisão na medição do tempo, fica difícil analisar o que acontece com o tempo à medida que δ diminui. Pelos valores auferidos, o tempo médio diminuiu um pouco para $\delta = 3$, exceto para as instâncias cuja bitola do aço é 16mm. O tempo requerido pelo algoritmo FFDWB$_\delta$ foi inferior a 0,01 segundos.

O ganho médio em relação ao FFDe_δ permaneceu inalterado ou foi levemente inferior, à medida que δ diminui, exceto nas instâncias cuja bitola do aço é 10mm. Para estas instâncias, o ganho em relação ao FFDe_δ aumentou quando $\delta = 3$. Isso se deu porque as soluções encontradas pelo FFDe_δ com $\delta = 3$ foram, em média, piores que as soluções encontradas pelo FFDe_δ com $\delta \geq 3$. Já o desperdício médio permaneceu inalterado ou foi levemente superior, à medida que δ diminuiu.

	m	Tempo (seg)				Ganho sobre FFDe_δ				Desperdício			
Bitola	médio	$\delta = m$	$\delta = 5$	$\delta = 4$	$\delta = 3$	$\delta = m$	$\delta = 5$	$\delta = 4$	$\delta = 3$	$\delta = m$	$\delta = 5$	$\delta = 4$	$\delta = 3$
5.0	7	0,23	0,31	0,31	0,08	0,945	0,945	0,945	0,945	0,324	0,324	0,324	0,324
6.3	11	0,30	0,33	0,44	0,30	1,868	1,849	1,830	1,715	1,438	1,453	1,472	1,605
8.0	12	0,35	0,33	0,42	0,42	1,128	1,128	1,128	1,016	6,281	6,281	6,281	6,438
10.0	20	0,94	0,89	1,06	0,71	0,825	0,825	0,825	0,942	7,843	7,843	7,843	7,939
12.5	14	0,53	0,33	0,56	0,47	2,207	2,207	2,207	2,158	5,467	5,467	5,467	5,566
16.0	13	0,35	0,23	0,23	0,42	1,247	1,247	1,247	1,247	20,675	20,675	20,675	20,675
20.0	8	0,12	0,15	0,19	0,19	0,894	0,894	0,894	0,744	12,023	12,023	12,023	12,190

Tabela 5.3: Desempenho do algoritmo HÍBRIDOm_δ aplicado às instâncias práticas, com $\delta = m$, $\delta = 5$, $\delta = 4$ e $\delta = 3$.

Na Tabela 5.4 comparamos o tempo, o ganho em relação às soluções encontradas pelo FFDe_δ, e o desperdício verificado ao aplicar o algoritmo HÍBRIDOm_δ às instâncias práticas agrupadas, com $\delta = m$, $\delta = 5$, $\delta = 4$ e $\delta = 3$.

Novamente é difícil analisar o que acontece com o tempo ao variarmos δ. Em algumas instâncias o tempo diminuiu, em outras o tempo aumentou. No entanto, em algumas instâncias o tempo gasto pelo algoritmo FFDWB$_\delta$ aumentou sensivelmente à medida que δ diminuiu. Para $\delta = m$, o tempo gasto pelo FFDWB$_\delta$ foi inferior a 0,01 segundos, para todas as instâncias. Já para $\delta = 3$, os tempo gasto pelo FFDWB$_\delta$ para resolver as instâncias cuja bitola é 6.3mm e 8mm foi de 8 segundos e 2 segundos, respectivamente.

O ganho em relação ao FFDe_δ permaneceu inalterado ou foi levemente inferior, à medida que δ diminui, exceto na instância cuja bitola do aço é 10mm. Nesta instância, o ganho em relação ao FFDe_δ aumentou quando $\delta = 3$. Isso se deu porque a solução encontrada pelo FFDe_δ com $\delta = 3$ foi pior que as soluções encontradas pelo FFDe_δ com $\delta \geq 3$. Já o desperdício permaneceu inalterado ou foi levemente superior, à medida que δ diminuiu.

		Tempo (seg)				Ganho sobre FFDe_δ				Desperdício			
Bitola	m	$\delta = m$	$\delta = 5$	$\delta = 4$	$\delta = 3$	$\delta = m$	$\delta = 5$	$\delta = 4$	$\delta = 3$	$\delta = m$	$\delta = 5$	$\delta = 4$	$\delta = 3$
5.0	48	42	41	46	51	0,607	0,607	0,607	0,569	0,020	0,020	0,020	0,058
6.3	209	40	36	53	64	0,474	0,474	0,474	0,474	0,017	0,017	0,017	0,017
8.0	264	200	195	297	166	1,278	1,278	1,258	1,238	0,009	0,009	0,019	0,068
10.0	365	1601	1752	1586	1450	1,365	1,365	1,365	1,397	0,061	0,061	0,061	0,061
12.5	301	484	658	840	685	1,481	1,481	1,481	1,531	0,268	0,268	0,268	0,268
16.0	218	169	165	176	148	1,673	1,673	1,673	1,673	4,995	4,995	4,995	4,995
20.0	105	32	25	28	32	0,489	0,489	0,489	0,489	2,507	2,507	2,507	2,507

Tabela 5.4: Desempenho do algoritmo HÍBRIDOm_δ aplicado às instâncias práticas agrupadas, com $\delta = m$, $\delta = 5$, $\delta = 4$ e $\delta = 3$.

Os testes realizados com as instâncias práticas também indicam que o desempenho do algoritmo HÍBRIDOm_δ é pouco afetado mesmo quando δ assume valores pequenos como 3, 4 ou 5.

Concluímos ser razoável, para as instâncias práticas aqui resolvidos, adotar a nova restrição proposta neste capítulo, que é limitar o número de itens distintos que podem ocorrer num padrão a uma constante inteira δ. Tal restrição visa adequar as soluções obtidas à restrição prática imposta pelo processo manual de corte do aço. Verificamos que a qualidade das soluções e o tempo requerido para encontrá-las praticamente não se alteraram, mesmo quando assumimos $\delta = 3$.

Capítulo 6

Considerações Finais

Nesta dissertação, apresentamos diversos algoritmos que desenvolvemos para o problema de corte unidimensional. Tais algoritmos são adequados para diferentes classes de problemas e podem ser classificados, quanto à qualidade das soluções encontradas, como algoritmos de aproximação, exatos, quase-exatos e heurísticos.

O algoritmo FFDe é equivalente ao algoritmo FFD, em termos de qualidade das soluções encontradas e de tempo computacional requerido. Na verdade, mostramos que, aplicados a uma mesma instância, os dois algoritmos encontram a mesma solução. Portanto, os resultados conhecidos sobre os limites de desempenho absoluto, assintótico e empírico válidos para o FFD [46, 87, 48, 3, 11], valem também para o FFDe. No entanto, o algoritmo FFD é voltado à classe de problemas 1/V/I/M, enquanto que o FFDe é adequado à classe de problemas 1/V/I/R. Assim como o FFD, o FFDe também pode ser implementado de forma a ter complexidade de tempo $\mathcal{O}(n \log n)$. Tratam-se de algoritmos de aproximação.

Discorremos sobre resultados teóricos [67] e práticos [69, 85] que sugerem ser verdadeira a conjectura MIRUP. Os testes que realizamos com instâncias geradas aleatoriamente e instâncias práticas também servem para fortalecer esta conjectura. Assumindo que esta conjectura seja verdadeira, temos um excelente limitante para o valor de uma solução inteira ótima. Utilizando este limitante, propusemos o algoritmo exato FFDWB, que mostrou-se eficiente ao resolver instâncias pequenas do problema de corte unidimensional.

O algoritmo HÍBRIDO que desenvolvemos reúne algoritmos de diversas naturezas, como o método Simplex com geração de colunas, o FFDWB e o FFDe. Deste fato deriva o seu nome. Esta abordagem diversificada tem se mostrado promissora, e tem sido bastante explorada em trabalhos recentes [74, 69, 86, 35]. Mostramos também que a diferença entre o valor da solução encontrada pelo algoritmo HÍBRIDO e o valor de uma solução inteira ótima é no máximo 1, se verdadeira a conjectura MIRUP. Nesse sentido, dizemos que o algoritmo HÍBRIDO é um algoritmo *quase-exato*.

Repetimos os testes realizados por Wäscher e Gau [86] com 4000 instâncias (com até 50 itens) geradas aleatoriamente e constatamos que o algoritmo HÍBRIDO obteve desempenho superior ao de todos os algoritmos testados por estes autores. O algoritmo HÍBRIDO encontrou uma solução inteira ótima para pelo menos 99,7% dessas instâncias e o tempo médio requerido para resolver cada uma das 4000 instâncias foi inferior a 4 segundos.

Resolvemos ainda outras 2000 instâncias geradas aleatoriamente, incluindo desta vez instâncias maiores (com até 100 itens). Nossa intenção era verificar se haveria um aumento brutal no tempo de computação. Novamente o desempenho do algoritmo HÍBRIDO mostrou-se satisfatório. Encontramos soluções ótimas para pelo menos 96,95% dessas instâncias e o tempo médio requerido foi de aproximadamente 15 segundos.

Devido ao fato de o algoritmo FFDWB ser adequado apenas para instâncias pequenas, introduzimos uma modificação no algoritmo HÍBRIDO com o objetivo de reduzir o tamanho do problema residual a ser resolvido pelo FFDWB, obtendo assim o que chamamos de algoritmo HÍBRIDOm.

Algoritmo	Classe	Tipo
FFDe	1/V/I/R	de aproximação
FFDWB	1/V/I/M	exato
HÍBRIDO	1/V/I/R	quase-exato
HÍBRIDOm	1/V/I/R	heurístico
MOCHILA$_\delta$	1/B/O	exato
MOCHILAe_δ	1/B/O	exato
SimplexGC$_\delta$	1/V/I/R	heurístico
FFDe_δ	1/V/I/R	de aproximação
FFDWB$_\delta$	1/V/I/M	exato
HÍBRIDO$_\delta$	1/V/I/R	quase-exato
HÍBRIDOm_δ	1/V/I/R	heurístico

Tabela 6.1: Algoritmos propostos nesta dissertação.

Ao resolver as 4000 instâncias de Wäscher e Gau, o algoritmo HÍBRIDOm mostrou-se ainda mais rápido que o algoritmo HÍBRIDO, requerendo em média aproximadamente 3 segundos para cada instância. O algoritmo HÍBRIDOm também apresentou desempenho superior ao de todos os algoritmos testados por Wäscher e Gau, perdendo apenas para o algoritmo HÍBRIDO.

O algoritmo HÍBRIDOm mostrou-se mais rápido e apresentou melhor qualidade nas soluções encontradas do que o algoritmo HÍBRIDO, ao resolver as 2000 instâncias que mencionamos ante-

riormente. Resolvemos também 216 instâncias práticas, incluindo uma instância com 365 itens, sendo que o algoritmo HÍBRIDOm só não encontrou uma solução ótima para uma instância.

Motivados por restrições de ordem prática verificadas no processo de corte manual do aço utilizado na indústria de construção civil, investigamos o problema de corte unidimensional onde o número de itens distintos que podem ocorrer num padrão é limitado por uma constante inteira δ. Os algoritmos MOCHILA$_\delta$, MOCHILAe_δ, SimplexGC$_\delta$, FFDe_δ, FFDWB$_\delta$, HÍBRIDO$_\delta$ e HÍBRIDOm_δ, incorporam esta nova restrição.

Os resultados obtidos ao resolver as 6000 instâncias aleatórias e as 216 instâncias práticas citadas anteriormente mostraram que o desempenho do algoritmo HÍBRIDOm_δ praticamente não é afetado, mesmo quando δ assume valores pequenos, como 3 ou 4.

O algoritmo HÍBRIDO apresentado nesta dissertação é similar ao algoritmo proposto por Wäscher e Gau em [86]. A diferença básica reside no fato de que estes autores utilizaram o algoritmo MTP para resolver o último problema residual, enquanto nós utilizamos o algoritmo FFDWB. Visto que os resultados que obtivemos foram superiores aos obtidos por Wäscher e Gau ao resolver as mesmas instâncias, parece ser interessante comparar o desempenho do FFDWB em relação ao MTP, ao BISON [71] e a outros algoritmos exatos propostos para o *bin packing problem*.

O esquema utilizado nos algoritmos híbridos propostos nesta dissertação, podem ser adaptados para o problema de corte bi- e tridimensional. Para isto é necessário dispor de métodos para achar uma solução inicial, gerar novas colunas e resolver o problema residual final. Tais métodos são encontrados na literatura. Um desdobramento natural do trabalho de pesquisa que resultou nesta dissertação seria integrar estes métodos de forma a obter algoritmos capazes de encontrar soluções inteiras para o problema de corte bi- e tridimensional.

Referências Bibliográficas

[1] M. Arenales and R. Morábito, *An and/or-graph approach to the solution of two-dimensional non-guillotine cutting problems*, European Journal of Operations Research, 84 (1995), pp. 599–617.

[2] R. W. Ashford and R. C. Daniel, *Some lessons in solving practical integer programs*, Journal of the Operational Research Society, 43 (1992), pp. 425–433.

[3] B. S. Baker, *A new proof for the first-fit decreasing bin-packing algorithm*, Journal of Algorithms, 6 (1985), pp. 49–70.

[4] B. S. Baker, D. J. Brown, and H. P. Katseff, *A $\frac{5}{4}$ algorithm for two-dimensional packing*, Journal of Algorithms, 2 (1981), pp. 348–368.

[5] B. S. Baker, E. G. Coffman Jr., and R. L. Rivest, *Orthogonal packings in two-dimensions*, SIAM Journal on Computing, 9 (1980), pp. 846–855.

[6] S. Baum and L. E. T. Jr., *Integer rounding for polymatroid and branching optimization problems*, SIAM J. Alg. Disc. Meth., 2 (1981), pp. 416–425.

[7] E. M. L. Beale, *Cycling in the dual simplex algorithm*, Naval Res. Logist. Quart., 2 (1955), pp. 269–275.

[8] J. E. Beasley, *Algorithms for unconstrained two-dimensional guillotine cutting*, Journal of the Operational Research Society, 36 (1985), pp. 297–306.

[9] R. G. Bland, *New finite pivoting rules for the simplex method*, Math. Oper. Res., 2 (1977), pp. 103–107.

[10] A. Bortfeldt and H. Gehring, *Applying tabu search to container loading problems*, in Operations Research Proceedings, 1997, pp. 533–538.

[11] J. Bramel, W. T. Rhee, and D. Simchi-Levi, *Average-case analysis of the bin-packing problem with general cost structures*, Naval Res. Logist., 44 (1997), pp. 673–686.

[12] N. Christofides and C. Whitlock, *An algorithm for two dimensional cutting prblems*, Operations Research, 25 (1977), pp. 30–44.

[13] F. R. K. Chung, M. R. Garey, and D. S. Johnson, *On packing two-dimensional bins*, SIAM J. Algebraic Discrete Methods, 3 (1982), pp. 66–76.

[14] V. Chvátal, *Linear Programming*, W. H. Freeman and Company, New York, 1980.

[15] E. G. Coffman Jr, M. R. Garey, and D. S. Johnson, *An application of bin-packing to multiprocessor scheduling*, SIAM Journal on Computing, 7 (1978), pp. 1–17.

[16] E. G. Coffman, Jr., M. R. Garey, and D. S. Johnson, *Approximation algorithms for bin packing - an updated survey*, in Algorithms design for computer system design, G. Ausiello, M. Lucertini, and P. Serafini, eds., Spring-Verlag, New York, 1984, pp. 49–106.

[17] E. G. Coffman Jr., M. R. Garey, D. S. Johnson, and R. E. Tarjan, *Performance bounds for level oriented two-dimensional packing algorithms*, SIAM Journal on Computing, 9 (1980), pp. 808–826.

[18] CPLEX, *Using the CPLEX Callable Library and CPLEX Mixed Integer Library*, CPLEX Optimization, Inc, 1995.

[19] J. Csirik, *The parametric behavior of the first-fit decreasing bin packing algorithm*, Journal of Algorithms, 15 (1993), pp. 1–28.

[20] J. Csirik and G. J. Woeginger, *Shelf algorithms for on-line strip packing*, Information Processing Letters, 63 (1997), pp. 171–175.

[21] G. B. Dantzig, *Discrete-variable extremum problems*, Operations Research, 5 (1957), pp. 266–277.

[22] A. Diegel, *Integer lp solution for large trim problems*, Paper presented at the EURO/TIMS Conference, Paris, July 6-8, (1988).

[23] W. B. Dowsland, *Two and three dimensional packing problems and solution methods*, New Zealand Journal of Operational Research, 13 (1985), pp. 1–18.

[24] H. Dyckhoff, *A typology of cutting and packing problems*, European Journal of Operations Research, 44 (1990), pp. 145–159.

[25] H. Dyckhoff and U. Finke, *Cutting and Packing in Production and Distribution.*, Springer-Verlag, Heidelberg, 1992.

[26] H. Dyckhoff, H. J. Kruse, D. Abel, and T. Gal, *Trim loss and related problems*, Omega, 13 (1985), pp. 59–72.

[27] S. Eilon and N. Christofides, *The loading problems*, Management Science, 17 (1971), pp. 259–268.

[28] E. Falkenauer, *A hybrid grouping genetic algorithm for bin packing*, Journal of Heuristics, 2 (1996), pp. 5–30.

[29] W. Fernandez de la Vega and G. S. Lueker, *Bin packing can be solved within $1+\epsilon$ in linear time*, Combinatorica, 1 (1981), pp. 349–355.

[30] M. Fieldhouse, *The duality gap in trim problems*, SICUP-bulletin, 5 (1990).

[31] M. R. Garey, R. L. Graham, and D. S. Johnson, *On a number theoretic bin packing conjecture*, in Proc. 5th Hungarian Combinatorics Colloquium, Amsterdam, 1978, North-Holland, pp. 377–392.

[32] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of $\mathcal{NP}$-Completeness*, W. H. Freeman and Co., San Fransisco, 1979.

[33] ——, *Approximation algorithms for bin packing problems: a survey*, in: D. Ausiello and M. Lucertini (eds.), Analysis and Design of Algorithms in Combinatorial Optimization, CISM Courses and Lectures, 266 (1981), pp. 147–172.

[34] T. Gau, *Quasi-exact and heuristic algorithms for the standard one-dimensional cutting stock problem*, tech. report, Technische Universitaet Braunschweig, 1994.

[35] ——, *Solution methods for the standard one-dimensional cutting stock problem*, Physica, Heidelberg, 1997.

[36] T. Gau and G. Wäsher, *CUTGEN1: A problem generator for the standard one-dimensional cutting stock problem*, European Journal of Operations Research, 84 (1995), pp. 572–579.

[37] P. Gilmore and R. Gomory, *A linear programming approach to the cutting stock problem*, Operations Research, 9 (1961), pp. 849–859.

[38] ——, *A linear programming approach to the cutting stock problem - part II*, Operations Research, 11 (1963), pp. 863–888.

[39] ——, *Multistage cutting stock problems of two and more dimensions*, Operations Research, 13 (1965), pp. 94–120.

[40] B. L. Golden, *Approaches to the cutting stock problem*, AIIE Trans., 8 (1976), pp. 265–274.

[41] R. W. Haessler and F. B. Talbot, *Load planning for shipments of low density products*, European Journal of Operations Research, 44 (1990), pp. 289–299.

[42] K. Heicken and W. König, *Integration eines heuristisch-optimierenden Verfahrens zur Lösung eines eindimensionalen Verschnittproblems in einem EDV-gestüzten Produktionsplanungs-und-steuerungssystems*, OR Spektrum, 1 (1980), pp. 251–259.

[43] J. C. Herz, *A recursive computational procedure for two-dimensional stock-cutting*, IBM Journal of Research Development, (1972), pp. 462–469.

[44] A. I. Hinxman, *The trim-loss and assortment problems: a survey*, European J. Oper. Res., 5 (1980), pp. 8–18.

[45] E. Horowitz and S. Sahni, *Computing partitions with applications to the knapsack problem*, J. Association Comput. Mach., 21 (1974), pp. 277–294.

[46] D. S. Johnson, *Near-optimal bin packing algorithms*, PhD thesis, Massachusetts Institute of Technology, Cambridge, Mass., 1973.

[47] ——, *Fast algorithms for bin packing*, J. Comput. Syst. Sci., 8 (1974), pp. 272–314.

[48] D. S. Johnson, A. Demars, J. D. Ullman, M. R. Garey, and R. L. Graham, *Worst-case performance bounds for simple one-dimensional packing algorithms*, SIAM Journal on Computing, 3 (1974), pp. 299–325.

[49] N. Karmakar and R. M. Karp, *An efficient approximation scheme for the one dimensional bin packing problem*, in Proceedings, 23rd Ann. Symp. on Foundations of Computer Science, Los Angeles, 1982, IEEE Computer Society, pp. 312–320.

[50] V. Klee and G. J. Minty, *How good is the simplex algorithm?*, in Inequalities, III (Proc. Third Sympos., Univ. California, Los Angeles, Calif., 1969; dedicated to the memory of Theodore S. Motzkin), New York, 1972, Academic Press, pp. 159–175.

[51] K. L. Krause, Y. Y. Shen, and H. D. Schwetman, *Analysis of several task-scheduling algorithms for a model of multiprogramming computer systems*, J. Association Comput. Mach., 22 (1975), pp. 522–550.

[52] C. C. Lee and D. T. Lee, *A simple on-line bin-packing algorithm*, J. Association Comput. Mach., 32 (1985), pp. 562–572.

[53] K. Li and K.-H. Cheng, *A generalized harmonic algorithm for on-line multidimensional bin packing*, TR UH-CS-90-2, University of Houston, January 1990.

[54] ——, *Heuristic algorithms for on-line packing in three dimensions*, Journal of Algorithms, 13 (1992), pp. 589–605.

[55] J. Lorie and L. J. Savage, *Three problems in capital rationing*, Journal of Business, 28 (1955), pp. 229–239.

[56] O. Marcotte, *The cutting stock problem and integer rounding*, Mathematical Programming, 33 (1985), pp. 82–92.

[57] ——, *An instance of the cutting stock problem for which the rounding property does not hold*, Oper. Res. Lett., 4 (1986), pp. 239–243.

[58] S. Martello and P. Toth, *Optimal and canonical solutions of the change making problem*, European Journal of Operations Research, 4 (1980), pp. 322–329.

[59] S. Martello and P. Toth, *Knapsack problems*, Wiley-Interscience Series in Discrete Mathematics and Optimization, John Wiley & Sons Ltd., Chichester, 1990. Algorithms and computer implementations.

[60] F. K. Miyazawa, *Algoritmos de aproximação para problemas de empacotamento*, PhD thesis, Instituto de Matemática e Estatística, São Paulo, 1997.

[61] F. K. Miyazawa and Y. Wakabayashi, *An algorithm for the three-dimensional packing problem with asymptotic performance analysis*, Algorithmica, 18 (1997), pp. 122–144.

[62] ——, *Orthogonal z-oriented three-dimensional packing algorithms*, SIAM Journal on Computing, (1997). To appear.

[63] R. Morábito and M. N. Arenales, *Performance of two heuristics for solving large scale two-dimensional guillotine cutting problems*, INFOR, 33 (1995), pp. 145–155.

[64] K. Neumann and M. Morlock, *Operations Research*, Carl Hanser Verlag, Munich, 1993.

[65] J. F. Oliveira and J. S. Ferreira, *An improved version of Wang's algorithm for two-dimensional cutting problems*, European Journal of Operations Research, 44 (1990), pp. 256–266.

[66] J. F. Pierce, *Some large-scale production scheduling problems in the paper industry*, Prentice-Hall, Englewood Cliffs, 1964.

[67] G. Scheithauer and J. Terno, *About the gap between the optimal values of the integer and continuous relaxation one-dimensional cutting stock problem*, in Operations Research Proceedings, Berlin, 1992, Springer-Verlag.

[68] G. Scheithauer and J. Terno, *A branch & bound algorithm for solving one-dimensional cutting stock problems exactly*, Appl. Math. (Warsaw), 23 (1995), pp. 151–167.

[69] G. Scheithauer and J. Terno, *The modified integer round-up property of the one-dimensional cutting stock problem*, European Journal of Operations Research, 84 (1995), pp. 562–571.

[70] I. Schiermeyer, *Reverse-fit: A 2-Optimal algorithm for packing rectangles*, Lecture Notes in Computer Science, 855 (1994).

[71] A. Scholl, R. Klein, and C. Jürgens, *Bison: A fast hybrid procedure for exactly solving the one-dimensional bin packing problem*, Working Paper, Technische Hochschule Darmstadt, (1996).

[72] P. Schwerin and G. Wäscher, *The bin-packing problem: A problem generator and some numerical experiments with FFD packing and MTP*, International Transactions in Operational Research, 4 (1997), pp. 377–389.

[73] D. Simchi-Levi, *New worst-case results for the bin-packing problem*, Naval Res. Logist., 41 (1994), pp. 579–585.

[74] H. Stadtler, *A one-dimensional cutting stock problem in the aluminium industry and its solution*, European Journal of Operations Research, 44 (1990).

[75] F. Stehling, *Der Bedarf an Münzen in verschiedenen Münzsystemen*, Methods of Operations Research, 46 (1983), pp. 509–521.

[76] A. Steinberg, *A strip-packing algorithm with absolute performance bound 2*, SIAM Journal on Computing, 26 (1997), pp. 401–409.

[77] P. E. Sweeny and E. R. Paternoster, *Cutting and packing problems: a categorized, application-oriented research bibliography*, Journal of the Operational Research Society, 43 (1992), pp. 691–706.

[78] P. H. Vance, *Branch-and-price algoritms for the one-dimensional cutting stock problem*, Computational Optimization and Applications, 9 (1998), pp. 211–228.

[79] P. H. Vance, C. Barnhart, E. L. Johnson, and G. L. Nemhauser, *Solving binary cutting stock problems by column generation and branch and bound*, Computational Optimization and Applications, 3 (1994), pp. 111–130.

[80] F. Vanderbeck, *Computational study of a column generation algorithm for bin packing and cutting stock problems*, Research Papers in Management Studies, 14 (1996). Engineering Department and Judge Institute of Management Studies, University of Cambridge.

[81] P. Y. Wang, *Two algorithms for constrained two dimensional cutting stock problems*, Operations Research, 31 (1983), pp. 573–586.

[82] T. S. Wee and M. J. Magazine, *Assembly line balancing as generalized bin packing*, Oper. Res. Lett., 1 (1982), pp. 56–58.

[83] G. Wäscher, *An LP-based approach to cutting stock problems with multiple objectives*, European Journal of Operations Research, 44 (1990), pp. 175–184.

[84] G. Wäscher and T. Gau, *Test data for the one-dimensional cutting stock problem*, Working Paper, Technische Universitaet Braunschweig, (1993).

[85] ——, *Two approaches to the cutting stock problem*, in IFORS'93 Conference, Lisbon, 1993.

[86] ——, *Heuristics for the integer one-dimensional cutting stock problem: a computational study*, OR Spektrum, 18 (1996), pp. 131–144.

[87] M. Yue, *A simple proof of the inequality* $\mathrm{FFD}(L) \leq \frac{11}{9}\mathrm{OPT}(L) + 1$, $\forall L$ *for the FFD bin-packing algorithm*, Acta Math. Appl. Sin., Engl. Ser., 4 (1991), pp. 321–331.