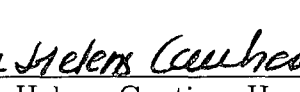
GERAÇÃO DE COLUNAS EM PROBLEMAS
DE OTIMIZAÇÃO COMBINATÓRIA

Elivelton Ferreira Bueno

TESE SUBMETIDA AO CORPO DOCENTE DA COORDENAÇÃO DOS PROGRAMAS DE PÓS-GRADUAÇÃO DE ENGENHARIA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

Aprovada por:

Prof. Nelson Maculan Filho, D. Sc.



Prof. Maria Helena Cautiero Horta Jardim, D. Sc.

Prof. Luiz Satoru Ochi, D. Sc.

RIO DE JANEIRO, RJ - BRASIL
MARÇO DE 2005

BUENO, ELIVELTON FERREIRA

Geração de Colunas em Problemas de Otimização Combinatória [Rio de Janeiro] 2005

XII, 102 p. 29,7 cm (COPPE/UFRJ, M.Sc., Engenharia de Sistemas e Computação, 2005)

Tese - Universidade Federal do Rio de Janeiro, COPPE

1. Otimização Combinatória
2. Geração de Colunas
3. Algoritmo *branch-and-price*
4. Problema de Particionamento
5. Problema de *bin-packing*

I. COPPE/UFRJ II. Título (série)

Para Carmenlúcia, que mudou
seu caminho em função do meu.

Para João Ilídio, que teria
gostado de ver este trabalho.

Agradecimentos

Muitas pessoas, algumas cientes disto e outras não, contribuíram para que esta tese se concretizasse. Diante disto, e dada esta parte (opcional) da tese, eu não poderia deixar de citar alguns nomes, mesmo que sejam apenas alguns, porque há aqueles que escolheram ajudar-me anonimamente e também porque, certamente, não vou recordar de todos neste momento. Então, agradeço: ao professor Maculan, pela disposição em me orientar, pelos ensinamentos, sugestões e palavras de ânimo, pelo apoio nas participações de eventos científicos e, claro, pelos momentos reservados quando realmente precisei, apesar de sua agenda tão cheia; à professora Maria Helena e ao professor Satoru, por aceitarem tão prontamente ao convite de participação na banca de defesa desta tese e pelas palavras que a tornaram melhor; ao professor Marco Antonio, da Universidade Catótica de Goiás, pelas sugestões e pelo zelo e paciência ao me mostrar esse caminho que estou seguindo na Ciência e, antes de tudo, pela amizade sempre segura; ao professor Adilson Xavier, pela recepção na minha chegada à COPPE e ao Rio de Janeiro, por ser tão amigavelmente acessível e pelos comentários interessantes e ações incentivadoras; à professora Márcia Fampa, por todos os ensinamentos sobre programação linear e programação linear inteira, por sua disposição em ajudar sempre que precisei e pelos comentários tão oportunos, principalmente no momento da decisão do tema desta tese; à Fátima, pela sabedoria que transmite tão naturalmente, pela grande presteza com que atendeu a inúmeros pedidos de ajuda que fiz, pelo cuidado para que tudo desse certo no processo de defesa da tese, pela companhia sempre amiga e pelo carisma; às secretárias do PESC, particularmente à Cláudia, à Lúcia, à Solange e à Sônia; à Juliana Fernandes, que muito me ajudou e me tolerou no seu espaço e no seu dia-a-dia, principalmente nos primeiros meses da minha permanência no Rio de Janeiro;

ao Eduardo e à Ângela, verdadeiros amigos, que sempre se empenharam ao máximo para me auxiliar, até mesmo antes que eu fizesse algum pedido, nas inúmeras vezes que precisei deles; ao Henrique, colega e primeiro amigo carioca, de caráter excepcionalmente admirável, que me ajudou em vários momentos de estudos e, além disso, me fez sentir como se eu fosse mais um membro da sua família; aos colegas também freqüentadores do Laboratório de Otimização (LabOtim), particularmente ao Luidi, à Talita, à Ádria, ao Yuri, à Juliana, à Rosa, à Ana Lúcia, ao Ronaldo, ao Pedro e à Michele, que muitas vezes até sem perceberem, colaboraram em muitas partes deste trabalho; ao Luciano, pelas palavras de ânimo, e pelos inúmeros sermões também, que certamente me tornaram um pouco melhor para a vida; à Margareth, muito amiga, extremamente ética e profissional, com quem aprendi muitas lições para a vida toda e que literalmente abriu as portas da sua casa e me ajudou em um dos momentos que mais precisei; à toda a minha família no Rio, especialmente à Guinez, à Terezinha, à Glória e à tia Isabel, pela dedicação única com que cuidaram de mim, e também à Simone e ao José, à Eunice e ao Salvador, ao Oliveira, ao Jerônimo e à Norma, à Christiane e ao Orivelson, à Viviane, ao Marcelo, à Carla e ao Hugo que, desde o dia em que os conheci, se dispuseram tão carinhosamente a me auxilar; ao Leizer, pelo companheirismo e por desempenhar tão bem o papel de irmão, como se de fato fosse um; à minha mãe, Carmenlúcia, e também ao Vilmar, à Elisângela e ao Fúlvio, à Eliane, ao Aikon e à Yhara, por me darem inspiração para os trabalhos do Curso e por entenderem os longos períodos de ausência devido ao mestrado. Muito especialmente, também agradeço ao Vinícius Sandes, cuja a atenção e palavras cheias de certeza me plantaram esperanças firmes de que vale a pena lutar para alcançarmos cada alvo que definimos nas diversas áreas da nossa vida. E, finalmente, agradeço ao CNPq pela bolsa de estudos concedida durante a maior parte do mestrado.

Resumo da Tese apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc.)

GERAÇÃO DE COLUNAS EM PROBLEMAS DE OTIMIZAÇÃO COMBINATÓRIA

Elivelton Ferreira Bueno
Março/2005

Orientador: Nelson Maculan Filho

Programa: Engenharia de Sistemas e Computação

O estudo e desenvolvimento de técnicas eficientes para Otimização Combinatória, tem tornado possível a obtenção da solução ótima de uma abrangente quantidade e tamanhos de problemas de Programação Linear Inteira (PLI). Várias pesquisas têm focalizado seus esforços na obtenção de relaxações mais justas para esses problemas, no sentido de que elas forneçam uma solução mais próxima da solução ótima. Decomposições e reformulações de problemas de PLI têm sido intensamente desenvolvidas com esse propósito. Em geral, a decomposição e a reformulação de problemas de PLI resultam em um número excessivamente grande de variáveis (colunas). Este trabalho discute essas técnicas em um método exato de enumeração implícita, usualmente denominado *branch-and-price*, que combina apropriadamente geração de colunas em problemas de programação linear com *branch-and-bound*, já clássico em problemas de PLI. Muitos problemas de otimização combinatória podem ser formulados como problemas de particionamento. Além disso, grande parte dos algoritmos de geração de colunas para problemas de PLI têm sido desenvolvidos para formulações baseadas nesse problema. Assim, o principal resultado deste trabalho consiste na implementação de um esquema de *branching* proposto para o caso particular do problema de particionamento com variáveis binárias. Os resultados se referem à implementação do *branch-and-price* para o problema de *bin packing*. Algumas alternativas que têm influência significativa no desempenho do algoritmo também são discutidas.

Abstract of Thesis presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degreee of Master of Science (M.Sc.)

COLUMN GENERATION FOR COMBINATORIAL OPTIMIZATION PROBLEMS

Elivelton Ferreira Bueno

March/2005

Advisor: Nelson Maculan Filho

Department: Systems Engineering and Computer Science

The study and development of efficient techniques for Combinatorial Optimization has made it possible to achieve the optimal solution of a wider range and size of integer programs. Several investigations have particularly focused on obtaining tight relaxations for integer programs, so that it is possible to provide a solution that is closer to the integer optimal solution. Problem decompositions and reformulations have been intensively developed for this purpose. Usually, integer program decompositions and reformulations result in an extremely large number of variables (columns). This paper discusses these techniques in an exact approach of implicit enumeration, the so called branch-and-price, that combines appropriate column generation in problems of linear programming with branch-and-bound, a classic method in integer programs. Several important combinatorial optimization problems can be formulated as set partitioning problems. Furthermore, many column generations have been developed for formulations based on those problems. Thus, the main result of this work consists of the implementation of a branching scheme proposed for the particular case of the set partitioning problem with binary variables. Results refer to the implementation of branch-and-price for the bin packing problem. A few algorithmic choices that have a significant influence on the performance of the algorithm have also been discussed.

Sumário

Introdução

Uma grande variedade de problemas práticos podem ser formulados através de certos modelos de otimização com uma vasta teoria disponível na literatura e que podem ser resolvidos com técnicas específicas para cada um desses modelos. A Otimização Combinatória trata da alocação eficiente de recursos limitados para alcançar determinado objetivo quando algumas ou todas as variáveis são restritas a assumirem apenas valores inteiros. Restrições impostas aos recursos determinam as alternativas possíveis, ou seja, as soluções que são consideradas viáveis. Geralmente, esses problemas apresentam um número muito grande de alternativas possíveis, de forma que, em muitos casos, determinar qual dentre elas é a melhor se torna uma tarefa extremamente difícil.

Os modelos de Otimização Combinatória são bastante comuns em várias situações que envolvem problemas práticos com atividades e recursos indivisíveis. Além disso, esses problemas, em geral, apresentam um número finito de alternativas possíveis e, conseqüentemente, podem ser apropriadamente formulados como problemas de Otimização Combinatória, muitas vezes como problemas de particionamento.

Em um problema geral de particionamento, temos um conjunto base de elementos e certas regras para a geração de subconjuntos viáveis e seus respectivos custos. O objetivo é encontrar um particionamento do conjunto base em subconjuntos viáveis de modo a obtermos o menor (maior) custo (lucro) possível, caso o problema seja de minimização (maximização). Balas

e Padberg [2] apresentam uma lista de artigos, relatórios e livros que tratam de aplicações desse problema em diversas áreas.

Os algoritmos que muitas vezes têm apresentado melhor desempenho na resolução de problemas de otimização combinatória são aqueles que utilizam técnicas de Programação Linear Inteira, com um vasto número de resultados teóricos e aplicações concretas, e, conseqüentemente, técnicas de Programação Inteira. Aqui, o termo "programação" se refere ao "planejamento" das decisões a serem tomadas a partir da solução do problema.

Dentre os métodos desenvolvidos para a resolução de problemas de Programação Linear Inteira, os que utilizam técnicas de geração de colunas tem sido bastante estudados e implementados nos últimos anos. Essas técnicas foram propostas inicialmente como parte dos métodos para problemas de Programação Linear de grande porte, que geralmente envolvem um grande número de variáveis. Na época da sua proposta inicial, por volta de 1960, por certo a principal motivação para o uso dessas técnicas estava baseada no fato de que nem sempre um computador é capaz de armazenar todos os dados de uma aplicação. Neste contexto, no método simplex, as colunas a entrarem na base são geradas por meio da resolução de um problema auxiliar.

Veremos que a técnica de geração de colunas está baseada em reformulações que envolvem um número geralmente muito grande de variáveis. Grande parte dos algoritmos de geração de colunas para problemas de programação linear inteira têm sido desenvolvidos para formulações baseadas no problema de particionamento, conforme Barnhart *et al.* [6]. Nesta tese e, em particular, na nossa implementação, estaremos interessados em resolver o problema de particionamento para o caso em que cada coluna da matriz de restrições satisfaz a uma dada inequação, que é formada pela soma de certos pesos associados às suas linhas. Neste sentido, dizemos que o problema tem colunas tipo "mochila" (ou *knapsack*). Ele é usualmente denominado

problema de *bin packing* [62].

Formulações extensivas de um problema de programação inteira podem ser obtidas como resultado de alguma decomposição do seu conjunto viável. O desenvolvimento de métodos de resolução de problemas de programação linear que exploram a estrutura particular do modelo foi proposto inicialmente por Ford e Fukerson [26]. Muitos pesquisadores consideram esse trabalho como o marco inicial da abordagem de geração de colunas e reconhecidamente inspirou Dantzig e Wolfe [20] a proporem um esquema de decomposição de problemas gerais de programação linear.

Land e Doig [42] introduziram, em 1960, um algoritmo enumerativo para problemas de programação linear inteira, denominado *branch-and-bound*, que utiliza estimativas no valor da solução ótima para evitar a enumeração de todas as soluções possíveis. Algoritmos exatos, como esse, estão fundamentados na obtenção de sucessivas soluções de problemas mais fáceis, denominados *relaxações* e obtidos a partir de uma operação conhecida por *branching*, fornecendo os *limitantes*, ou *bounds*, da solução ótima do problema inteiro. Desde então, várias abordagens têm sido desenvolvidas para a resolução de problemas inteiros: enumeração implícita [5], decomposição [8], relaxação lagrangeana [30] e heurísticas [65], por exemplo. Atualmente, grande parte das pesquisas têm se concentrado no desenvolvimento de técnicas e algoritmos para classes de problemas bem particulares. Muitos dos algoritmos desenvolvidos estão baseados nas idéias do método *branch-and-bound*, que também descrevemos nesta tese.

Para a resolução de um problema de programação inteira, a associação da técnica de geração de colunas com algoritmos do tipo *branch-and-bound*, proposta por Desrosiers, Soumis e Desrochers [22], reforçou a importância da decomposição proposta por Dantzig e Wolfe [20] e, desde então, a abordagem de resolução por geração de colunas tornou-se uma das principais técnicas

utilizadas para problemas de programação inteira. Hoje, esse método é usualmente denominado *branch-and-price*.

Os métodos do tipo *branch-and-bound* são mais eficientes se tivermos boas estimativas (*bounds*) para o problema inteiro. A abordagem convencional desses métodos envolve o uso de uma *relaxação linear* do problema inteiro para a obtenção desses *bounds*. A resolução de um problema de programação inteira resultante da relaxação de todas as restrições de integralidade da sua formulação, é geralmente fácil (se usarmos o algoritmo simplex, por exemplo), mas a estimativa obtida para o problema inteiro normalmente é ruim.

Existem outras técnicas, que não abordaremos nesta tese, muito utilizadas na obtenção de *bounds* melhores. A *relaxação lagrangeana*, descrita por Reinoso e Maculan [56] e por Krieken *et al.* [40], por exemplo, consiste na relaxação de algumas restrições da formulação do problema, em vez de todas as suas restrições de integralidade. Há, também, métodos que procuram melhorar as relaxações lineares acrescentando apropriadamente outras inequações à formulação do problema. Uma vasta lista de referências bibliográficas que tratam desses métodos é apresentada por Ralphs e Galati [55], por exemplo.

A resolução de problemas de particionamento tem sido abordada por pesquisadores há várias décadas. No entanto, Hoffman e Padberg [38] são considerados os primeiros a proporem um algoritmo que foi capaz de resolver problemas de particionamento de grande porte até a otimalidade. Desde então, outras pesquisas têm sido publicadas sobre algoritmos exatos para problemas de particionamento, conforme Borndörfer [10], por exemplo.

Em geral, os algoritmos que são rápidos e capazes de resolver problemas de particionamento muito grandes estão baseados em métodos para problemas de programação linear, principalmente o método simplex, que trabalha com a idéia de soluções básicas. No entanto, as relaxações lineares de

problemas de particionamento maiores são altamente degenerados, no sentido em que certas variáveis básicas assumem valores iguais a zero numa solução básica viável. Sabemos, da teoria da Programação Linear, que isto dificulta (e até pode impedir) a convergência do algoritmo simplex. Portanto, precisamos usar um bom resolvedor para Programação Linear na resolução do problema de particionamento. Neste trabalho, optamos pelo XPRESS-MP, um dos pacotes computacionais para otimização de grande porte mais eficientes e usados atualmente. Apesar de alguns conceitos e comentários básicos, ao longo desta tese, estamos supondo que o leitor ou a leitora tenha conhecimentos básicos de Álgebra Linear, dos fundamentos da Programação Linear e, em particular, do método simplex (Dantzig [15]) para a resolução de problemas de programação linear. Podemos encontrar, na literatura, uma vasta teoria a respeito deste problema e de diferentes métodos desenvolvidos para sua resolução: Dantzig e Thapia [19], Maculan e Fampa [48] e Goldbarg e Luna [34], por exemplo.

O principal resultado do nosso trabalho consiste na implementação, em linguagem de programação C++, da regra de *branching* de Ryan e Foster [59] para problemas de particionamento 0-1. Os comentários sobre a implementação e os resultados obtidos nos permitem verificar vários pontos importantes no contexto da resolução de um problema de programação inteira, além de reforçar o conteúdo teórico sobre geração de colunas. Este trabalho servirá como base para a implementação de técnicas mais recentemente desenvolvidas no contexto da geração de colunas e que ainda precisam ter sua eficiência verificada na prática, conforme sugere, por exemplo, Briant *et al.* [11]. Procuramos, aqui, desenvolver um texto didático como facilitador para aqueles interessados na implementação de algoritmos do tipo *branch-and-price* para problemas de otimização combinatória.

Para atingirmos nosso objetivo, dividimos este trabalho em cinco capítulos.

No Capítulo 1, apresentamos alguns dos principais problemas de otimização, e algumas definições a eles associadas e que fazemos referência no decorrer da nossa tese.

No Capítulo 2, descrevemos os fundamentos dos métodos de decomposição para problemas de programação linear, em particular, do método de Dantzig e Wolfe, e comentamos sobre reformulações de problemas inteiros.

No Capítulo 3, tratamos de algoritmos, em particular, um *branch-and-bound*, para problemas de programação inteira; apresentamos os fundamentos de geração de colunas em programação linear; e descrevemos sobre o algoritmo *branch-and-price* para problemas inteiros.

No Capítulo 4, apresentamos nossa implementação do algoritmo *branch-and-price* para o problema de *bin-packing* e os principais resultados obtidos; abordamos, também, a estratégia de *branching* que utilizamos e damos uma vião geral do algoritmo implementado.

E, na última parte da tese, registramos nossas considerações finais.

Iniciemos nosso propósito, então, com os principais problemas de Otimização e algumas definições que estaremos referenciando no decorrer desta tese.

Capítulo 1

O Problema

Nosso objetivo, neste capítulo, é apresentar alguns dos principais problemas de Otimização Combinatória e algumas definições a eles associadas.

1.1 A Otimização Combinatória

A Otimização Combinatória trata da alocação eficiente de recursos limitados para alcançar determinado objetivo quando algumas ou todas as variáveis são restritas a assumirem apenas valores inteiros. Os modelos, nesta área, são bastante comuns em várias situações que envolvem problemas com atividades e recursos cuja representação fracionária não tem sentido prático. Esses problemas, em geral, apresentam um número finito (mas possivelmente muito grande) de alternativas viáveis e, conseqüentemente, podem ser apropriadamente formulados como problemas de Otimização Combinatória.

Neste caso, estamos interessados em minimizar (ou maximizar) o valor de uma função definida sobre um determinado domínio finito formado por entidades discretas. Ou ainda: dado um conjunto base E com $|E|$ elementos e uma família $\mathcal{F}$ de subconjuntos S_1, S_2, ..., S_K de E com os respectivos custos $c(S_1)$, $c(S_2)$, ..., $c(S_K)$, queremos encontrar uma solução $S^* \in \mathcal{F}$ de menor (ou maior) custo (lucro) possível. No entanto, obter essa solução para

o problema, denominada *solução ótima*, pode se tornar uma tarefa bastante difícil. Mesmo podendo ser conceitualmente fácil enumerar os subconjuntos em $\mathcal{F}$, o desafio vem do fato que já citamos anteriormente: a quantidade desses subconjuntos geralmente é muito grande. Isto inviabiliza o uso de uma abordagem que envolva apenas a enumeração explícita das soluções viáveis do problema. Este fato justifica o estudo e o desenvolvimento de algoritmos com custo computacional tão reduzido quanto possível e capazes de obter soluções ao menos próximas da solução ótima do problema.

Nesta tese, estamos considerando que tanto a função a ter seu valor otimizado (minimizado ou maximinizado), denominada *função objetivo*, quanto as restrições (equações ou inequações) que definem as possíveis soluções do problema, são lineares. Cooper and Farhangian [12], por exemplo, tratam de problemas combinatórios que envolvem restrições e função objetivo não-lineares.

Dentre os algoritmos que têm apresentado melhor desempenho na resolução de problemas de otimização combinatória estão aqueles que utilizam técnicas de Programação Linear Inteira, uma subárea da Programação Matemática com um vasto número de resultados teóricos e aplicações concretas bem sucedidas.

Em um modelo de Programação Linear Inteira, cada subconjunto de $S_j \in \mathcal{F}$, $j = 1, 2, ..., K$, está associado a um vetor de incidência (ou seja, para cada subconjunto, a componente associada a determinado elemento nele assume valor 1 e, caso não esteja presente, essa componente assume valor 0). Então, qualquer solução viável é representada por um vetor de incidência que satisfaz a todas as restrições presentes na formulação do problema.

Na próxima seção, apresentaremos os principais problemas e definições que faremos referência ao longo desta tese.

1.2 Alguns problemas de otimização linear

Uma grande variedade de problemas práticos podem ser formulados através de certos modelos de otimização com uma vasta teoria disponível na literatura e que podem ser resolvidos com técnicas específicas para cada um desses modelos.

Um problema geral de otimização, no conjunto dos números reais, pode ser escrito da seguinte forma:

$$\begin{array}{lll} [PO] & \textit{minimizar} & z = f(x) \\ & \textit{sujeito a:} & x \in \mathcal{S}, \end{array}$$

onde são conhecidos uma função $f : \mathbb{R}^n \to \mathbb{R}$ com $x \mapsto f(x) = z$ e um conjunto $\mathcal{S} \subseteq \mathbb{R}^n$.

Definições e notações: Considerando um problema de otimização, como em seu formato geral $[PO]$, seguem-se algumas definições usuais e notações:

(a) *Função objetivo*: $\begin{array}{ll} f : R^n \to & R \\ x \longmapsto & f(x) = z. \end{array}$

(b) *Conjunto viável*: $\mathcal{S} \subseteq R^n$.

(c) *Solução viável*: $x \in \mathcal{S}$.

(d) *Valor ótimo*: $z^* = \min\{f(x);\ x \in \mathcal{S}\}$.

(e) *Conjunto das soluções ótimas*: $\mathcal{S}^* = \{x \in \mathcal{S};\ f(x) = z^*\}$.

(f) *Solução ótima*: $x^* \in \mathcal{S}^*$.

Dizemos, ainda, que o problema $[PO]$ é um *problema inviável*, quando seu conjunto viável, $\mathcal{S}$, é vazio. Além disso, chamamos $[PO]$ de *problema ilimitado*, quando existe uma seqüência (x^k) tal que $x^k \in \mathcal{S}$ e $f(x^k) \to -\infty$, quando $k \to \infty$.

Um problema de otimização pode assumir formatos padrão com características particulares e com resultados mais específicos. Naturalmente, isto leva a uma maneira mais didática de classificar esses problemas para facilitar o estudo e desenvolvimento de técnicas de resolução eficientes. Não há um consenso geral a respeito dessa classificação, mas apresentaremos nas próximas subseções os modelos de programação linear, programação linear inteira, programação inteira mista, programação inteira, programação binária e otimização combinatória, inspirados fortemente em Wolsey [66].

Iniciaremos o propósito desta seção com o problema de programação linear, um dos mais estudados desde a década de 50 [17, 15] e de grande importância no contexto da otimização combinatória.

1.2.1 O problema de programação linear

O *problema de programação linear*, no formato padrão usual, é o seguinte problema de otimização:

$$\begin{array}{lll} [PL] & \textit{minimizar} & c^T x \\ & \textit{sujeito a:} & Ax = b \\ & & x \geq 0, \end{array}$$

onde a matriz $A \in \mathbb{R}^{m \times n}$ e os vetores $b \in \mathbb{R}^m$ e $c \in \mathbb{R}^n$ são dados com $0 < m \leq n$. Suponhamos, sem perda de generalidade, que $posto(A) = m$.

Além das definições e notações apresentadas anteriormente para um problema geral de otimização, agora listaremos outras que são usuais para o caso particular do problema $[PL]$.

Definições e notações: Consideremos o problema $[PL]$, em que a matriz A é formada pelas colunas a_j, para todo $j \in I = \{1, 2, ..., n\}$. Denominamos *matriz base* (ou simplesmente *base*), qualquer submatriz quadrada $B = [a_{B(1)}\ a_{B(2)}\ ...\ a_{B(m)}]$ formada por m colunas de A que sejam linear-

mente independentes (ou seja, $det(B) \neq 0$). Associadas a uma tal base B, listamos as seguintes definições:

(a) *Índices base*: $I_B = \{B(1),\ B(2),\ ...,\ B(m)\}$.

(b) *Índices não-base*: $I_N = \{N(1),\ N(2),\ ...,\ N(n-m)\} = I \setminus I_B$.

(c) *Variáveis básicas*: $x_B = [x_{B(1)}\ x_{B(2)}\ ...\ x_{B(m)}]^T \in \mathbb{R}^m$.

(d) *Variáveis não básicas*: $x_N = [x_{N(1)}\ x_{N(2)}\ ...\ x_{N(n-m)}\ ]^T \in \mathbb{R}^{n-m}$.

(e) *Solução básica*: $x^T = [x_B^T\ x_N^T] \in \mathbb{R}^{1\times n}$; $x_B = B^{-1}b$ e $x_N = 0$.

(f) *Solução básica viável*: $x^T = [x_B^T\ x_N^T] \in \mathbb{R}^{1\times n}$; $x_B = B^{-1}b \geq 0$ e $x_N = 0$.

(g) *Solução básica degenerada*: $x^T = [x_B^T\ x_N^T] \in \mathbb{R}^{1\times n}$; $x_B = B^{-1}b$ e $\exists j;\ x_{B(j)} = 0$.

(h) *Vetor de variáveis duais*: $u \in \mathbb{R}^m$; $u = c_B^T B^{-1}$,
onde $c_B^T = [c_{B(1)}\ c_{B(2)}\ ...\ c_{B(m)}]$.

(i) *Custos reduzidos*:

$$\bar{c}_j = \begin{cases} 0, & \text{se } j \in I_B \\ c_j - u^T a_j, & \text{se } j \in I_N \end{cases} \quad \text{para } j = 1,\ ...,\ n.$$

A interpretação do problema $[PL]$ é: dados a matriz A e os vetores b e c, encontrar, se existir, um ponto $x^* \in \mathcal{S}^*_{PL}$, ou certificar que $[PL]$ é um problema inviável ou um problema ilimitado.

Quando acrescentamos ao problema $[PL]$ algumas restrições, denominadas *restrições de integralidade*, exigindo que uma ou mais variáveis (mas não todas) assumam apenas valores inteiros, temos um novo problema denominado problema de programação (linear) mista, que apresentamos agora.

1.2.2 O problema de programação linear mista

O *Problema de Programação (Linear) Mista* pode ser escrito como o seguinte problema de otimização:

$$\begin{array}{ll} \textit{minimizar} & c^T x + h^T y \\ \textit{sujeito a:} & Ax + Gy = b \\ & x \geq 0,\ y \geq 0 \text{ e inteiro} \end{array}$$

onde as matrizes $A \in \mathbb{R}^{m \times n}$ e $G \in \mathbb{Q}^{m \times p}$ e os vetores $b \in \mathbb{Q}^m$, $c \in \mathbb{R}^n$ e $h \in \mathbb{Q}^p$ são dados com $m > 0$, $p > 0$ e $m \leq n + p$.

Quando acrescentamos restrições de integralidade a todas as variáveis do problema $[PL]$, teremos um problema de programação (linear) inteira, que apresentaremos agora.

1.2.3 O problema de programação inteira

Um *problema geral de programação inteira* pode ser escrito como

$$\begin{array}{ll} \textit{minimizar} & f(x) \\ \textit{sujeito a:} & x \in \mathcal{S} \subseteq \mathbb{Z}_+^n \end{array}$$

onde $\mathcal{S}$ é o conjunto de soluções viáveis do problema: pontos com n componentes inteiras e que satisfazem às restrições do problema, sejam elas lineares ou não.

Em particular, o *problema de programação linear inteira*, pode ser escrito como o seguinte problema de otimização:

$$[PI] \quad \begin{array}{ll} \textit{minimizar} & c^T x \\ \textit{sujeito a:} & Ax = b \\ & x \geq 0 \quad \textit{e inteiro,} \end{array}$$

onde a matriz $A \in \mathbb{Q}^{m \times n}$ e os vetores $b \in \mathbb{Q}^m$ e $c \in \mathbb{R}^n$ são dados com $0 < m \leq n$.

A partir deste ponto da tese, referenciaremos o problema de programação linear inteira também por *problema inteiro*, tão somente.

Ao menos à primeira vista, um problema inteiro é bastante parecido com um problema de programação linear. De fato, os resultados teóricos e os métodos desenvolvidos em programação linear são fundamentais para o entendimento e resolução de um problema de programação linear inteira. No entanto, como mostra Wolsey [66], a idéia de, por exemplo, arrendondar os valores fracionários das variáveis na solução ótima do problema linear para os inteiros mais próximos poderá ser desastrosa, levando a uma solução inteira possivelmente distante de uma solução ótima do problema inteiro.

Se desconsiderarmos as restrições de integralidade (dizemos que estamos relaxando essas restrições), o problema $[PI]$ se torna o problema linear $[PL]$, que é denominado *relaxação linear* de $[PI]$.

Quando todas as variáveis de um problema inteiro são restritas a assumirem apenas valores binários, digamos 0 ou 1, temos um problema de programação binária ou programação 0-1, que definiremos a seguir.

1.2.4 O problema de programação binária

O *Problema de Programação Binária (ou 0-1)*, que é um caso particular de problema inteiro, pode ser escrito como o seguinte problema de otimização:

$$\begin{array}{lll} [PB] & \textit{minimizar} & c^T x \\ & \textit{sujeito a:} & Ax = b \\ & & x \in \{0,\ 1\}^n, \end{array}$$

onde a matriz $A \in \mathbb{Q}^{m \times n}$ e os vetores $b \in \mathbb{Q}^m$ e $c \in \mathbb{Q}^n$ são dados com $0 < m \leq n$.

A relaxação linear do problema de programação binária $[PB]$ é o seguinte problema linear:

$$\begin{array}{lll} [\overline{PB}] & \textit{minimizar} & c^T x \\ & \textit{sujeito a:} & Ax = b \\ & & x \in [0,\ 1]^n. \end{array}$$

A seguir, formalizaremos a definição do problema de otimização combinatória.

1.2.5 O problema de otimização combinatória

Consideremos um dado conjunto finito $N = \{1, 2, ..., n\}$, custos (ou lucros) c_j para cada $j \in N$, e uma família $\mathcal{F}$ de subconjuntos viáveis de N. O problema que consiste em encontrar um subconjunto de custo mínimo (ou lucro máximo) é um *problema de otimização combinatória*, que pode ser escrito da seguinte forma:

$$[PC] \quad \begin{array}{ll} \textit{minimizar} & \sum_{j \in S} c_j \\ \textit{sujeito a:} & S \in \mathcal{F} \end{array}$$

onde $S \subseteq N$.

Em geral, um problema de otimização combinatória pode ser modelado como um problema de programação inteira ou, não muito raro, de programação binária. Como em muitas aplicações reais podemos ver um problema de otimização como uma escolha de uns poucos subconjuntos dentre um número muito grande deles, temos a possibilidade de formularmos o problema diretamente como um problema inteiro com um grande número de variáveis. Antes de apresentarmos mais claramente um caso de tal formulação, dita *extensiva*, comentaremos o problema geral de particionamento e, para o nosso propósito mais particular, o problema da mochila.

1.3 O problema de particionamento

Muitos problemas de otimização combinatória podem ser formulados como problemas de particionamento. Goldbarg e Luna [34] e Balas e Padberg [2] apresentam uma lista de artigos, relatórios e livros que tratam de aplicações desse problema em diversas áreas. Alguns exemplos são: recuperação de informações em banco de dados; problemas de despacho; alocação

de tripulações em linhas aéreas; distribuição do tráfego de comunicações em satélites; alocação de serviços de emergência; alocação de serviços diversos; roteamento de petroleiros; roteamento de veículos terrestres; distribuição de distritos de venda; problema de coloração; caminho em grafos; planejamento de tarefas; projeto de circuitos; exploração de petróleo em campos submarinos; análise de amostra de sangue.

Conforme Barnhart *et al.* [6], grande parte dos algoritmos de geração de colunas para problemas inteiros têm sido desenvolvidos para formulações baseadas no problema de particionamento, que descrevemos nesta seção.

Em um problema geral de particionamento, temos um conjunto base de elementos e certas regras para a geração de subconjuntos viáveis e seus respectivos custos. O objetivo é encontrar um particionamento do conjunto base em subconjuntos viáveis de modo a obtermos o menor custo possível.

Seja I um conjunto formado por m elementos. Consideremos uma certa família $\mathcal{F}$ de subconjuntos de I, que denominaremos *subconjuntos viáveis*. A cada subconjunto viável $I^j \subseteq I$ (ou ainda $I^j \in \mathcal{F}$) associemos um vetor de incidência $a_j \in \{0,\ 1\}^m$, em que $a_{ij} = 1$ se $i \in I^j$ e $a_{ij} = 0$, caso contrário, e um custo c_j. Ou seja, um subconjunto I^j viável sempre estará associado a um vetor viável a_j com custo c_j. No nosso caso, e geralmente, o número de subconjuntos viáveis é muito grande: ele cresce exponencialmente em relação ao número de elementos m. Além disso, freqüentemente o conjunto de vetores viáveis é bem definido, sendo possível enumerar todos os subconjuntos viáveis, implícita ou explicitamente.

Para apresentarmos um modelo formal desse problema, consideremos $c : \{0,1\}^n \to \mathbb{R}$ uma função tal que $\lambda \mapsto c(\lambda) = \sum_{j=1}^{n} c_j \lambda_j$. O problema de particionamento padrão pode ser escrito da seguinte forma:

$$
\begin{array}{lll}
[PP] & \textit{minimizar} & c(\lambda) \\
 & \textit{sujeito a:} & A\lambda = \mathbf{1} \\
 & & \lambda \in \{0,\ 1\}^n
\end{array}
$$

onde $\mathbf{1}$ é um vetor com todas as suas m componentes iguais a 1, n é o número de subconjuntos viáveis e $A \in \{0,\ 1\}^{m \times n}$ é uma matriz binária em que cada coluna é um vetor viável $a_j \in \{0,\ 1\}^m$, conforme descrevemos no parágrafo anterior.

Cada coluna da matriz de restrições A satisfaz ao menos a alguma restrição conhecida. Na nossa implementação, uma coluna é formada por uma solução viável para o problema da mochila 0-1, que descrevemos na próxima seção.

1.4 O problema da mochila 0-1

O problema da mochila 0-1, ou problema *knapsack* 0-1, é um dos mais importantes e mais estudados problemas de otimização combinatória. Ele tem muitas aplicações práticas e freqüentemente aparece como um subproblema na resolução de problemas mais complicados. Este é o caso do problema da nossa implementação, o que justifica os comentários desta seção. Conforme Garey e Johnson [29], o *knapsack* 0-1 é NP-completo, ou seja, resolvê-lo pode ser tão difícil quanto resolver qualquer problema de programação inteira. Apesar disto,, os melhores resultados práticos para relaxações lineares, estudos de algoritmos do tipo *branch-and-bound* e métodos de programação dinâmica, descrição da envoltória convexa das soluções viáveis, algoritmos aproximativos e heurísticas, métodos e análises probabilísticas têm sido obtidos para o problema da mochila, conforme lembra Dudziński e Walukiewicz [23].

O problema da mochila 0-1 consiste na seleção de itens tais que a soma dos valores a eles associados é maximizada e a soma de seus pesos não excedam à capacidade da mochila. Para a formulação do problema, consideremos um conjunto $\mathcal{C}$ de m itens, que representaremos por $\mathcal{C} = \{1,\ 2,\ ...,\ m\}$, e associemos a cada item $i \in \mathcal{C}$ um peso ℓ_i e um valor de utilidade u_i. O pro-

blema consiste, então, em determinarmos que subconjunto S' dos itens em C pode ser obtido de forma que a soma dos pesos dos elementos em S' seja menor ou igual a uma capacidade L e que a soma dos valores de utilidade desses elementos seja a maior possível. O problema da mochila 0-1 pode, então, ser enunciado como o seguinte problema de otimização:

$$[KP] \quad maximizar \quad z_{KP} = \sum_{i=1}^{m} u_i \alpha_i$$

$$sujeito\ a: \quad \sum_{i=1}^{m} \ell_i \alpha_i \leq L$$

$$\alpha_i \in \{0, 1\}, \quad para\ j = 1,\ 2,\ ...,\ m.$$

Suponhamos, sem perda de generalidade, que o valor L e cada ℓ_i, para $i \in C$, são inteiros positivos; e que $\ell_i \leq L$ com $\sum_{i=1}^{m} \ell_i > L$.

Duas publicações clássicas sobre problemas da mochila e, em particular, da mochila 0-1, se referem aos trabalhos de Martello e Toth [51, 50], de 1990. Gilmore e Gomory [33] tratam da solução do problema da mochila por meio de programação dinâmica. Através de métodos de programação dinâmica é possível resolver o problema $[PK]$ em tempo peseudo-polinomial, no sentido em que essa ordem de complexidade depende da capacidade L.

Consideremos, agora, o problema de particionamento $[PP]$ da seção anterior. Quando cada coluna da sua matriz de restrições A é uma solução viável de um problema da mochila 0-1, $[PK]$, diremos que temos um problema de particionamento com colunas *knapsack*, ou do tipo "mochila". Esse é o problema que apresentaremos na próxima seção e que estamos particularmente interessados em resolver na nossa implementação do algoritmo de geração de colunas.

1.5 O problema *bin packing*

Como veremos no próximo capítulo, a técnica de geração de colunas, bastante usual na resolução de problemas em otimização combinatória, está baseada em certas reformulações apropriadas do problema inteiro (a serem apresentadas no próximo capítulo). Em geral, essas reformulações são mais extensivas, ou seja, envolve um número muito maior de variáveis. Nesta tese e, em particular, na nossa implementação, estaremos interessados em resolver o problema de particionamento para o caso em que cada coluna da matriz de restrições satisfaz a uma dada inequação, que é formada pela soma de certos pesos associados às suas linhas. É neste sentido que dizemos que o problema tem colunas tipo "mochila" (ou *knapsack*). Ele é usualmente denominado problema de *bin packing* [62].

Como ilustração para o problema, considere um conjunto finito de objetos, digamos $I = \{1, 2, ..., m\}$, um tamanho ℓ_i associado a cada objeto $i \in I$ e caixas com tamanho máximo L. O problema *bin packing* consiste, então, em empacotarmos todos os m objetos no menor número possível de caixas homogênias com capacidade igual a L.

Para formularmos o *bin packing* da nossa ilustração como um problema de programação inteira, definimos uma variável de decisão x_i^k, que assumirá o valor 1 se o item i estiver na caixa k e zero caso contrário, e a variável y^k que será 1 se a caixa k for utilizada e zero caso contrário. Então, a formulação, que diremos *compacta*, do problema *bin packing* é a seguinte:

$$\textit{minimizar} \quad \sum_{k=1}^{m} y^k$$

$$\textit{sujeito a}: \quad \sum_{k=1}^{m} x_i^k = 1 \qquad \textit{para todo } i = 1, 2, ..., m,$$

$$\sum_{i=1}^{m} \ell_i x_i^k \leq L y^k \qquad \textit{para todo } k = 1, 2, ..., m,$$

$$x_i^k,\ y^k \in \{0,\ 1\} \quad \textit{para todo } i = 1, 2, ..., m,\ k = 1, 2, ..., m,$$

onde ℓ_i é o peso associado ao item $i \in I = \{1,\ 2,\ ...,\ m\}$ e estamos considerando a disponibilidade (pior caso) de m caixas de capacidade L.

Alternativamente, podemos considerar todas as maneiras possíveis de colocarmos objetos em uma caixa. Assim, se $a_q \in \{0,1\}^m$ é o vetor de incidência que representa uma tal maneira, então devemos ter $\sum_{i=1}^{m} \ell_i a_{iq} \leq L$. E, daí, considerando o conjunto $\mathcal{Q}$ de todas essas maneiras "viáveis", podemos reescrever o problema de *bin packing* da nossa ilustração como o seguinte problema de otimização:

$$\begin{aligned} &\textit{minimizar} && \sum_{q \in \mathcal{Q}} \lambda_q \\ &\textit{sujeito a}: && \sum_{q \in \mathcal{Q}} a_{iq}\lambda_q = 1 && \textit{para todo } i = 1, 2, ..., m, \\ & && \lambda_q \in \{0,\ 1\} && \textit{para todo } q \in \mathcal{Q}. \end{aligned}$$

A cardinalidade do conjunto $\mathcal{Q}$ e, conseqüentemente, o número de variáveis nesta última formulação, depende dos dados do problema, mas pode ser muito grande: geralmente, exponencial em relação ao número de objetos m.

Resumindo, podemos concluir que dados um inteiro $m > 0$ (número de itens), um valor L (capacidade) e valores ℓ_i, para cada $i = 1,\ 2,\ ...,\ m$, o problema *bin packing*, na usualmente denominada *formulação extensiva*, é o seguinte problema de particionamento:

$$\begin{aligned} [BP] \quad &\textit{minimizar} && \mathbf{1}^T\lambda \\ &\textit{sujeito a:} && A\lambda = \mathbf{1} \\ & && \lambda \in \{0,\ 1\}^n \end{aligned}$$

onde $\mathbf{1}$ é um vetor (de dimensão apropriada) com todas as suas componentes iguais a 1, e A é uma matriz binária com n colunas $a_q \in \{0,\ 1\}^m$, $q \in \{1,\ 2,\ ...,\ n\}$ tais que

$$\sum_{i=1}^{m} \ell_i a_{iq} \leq L.$$

Suponhamos, sem perda de generalidade, que os dados do problema $[BP]$ são todos inteiros positivos, isto é: $\ell_i \in \mathbb{Z}_+$, para todo $i \in \{1, 2, ..., m\}$, e $L \in \mathbb{Z}_+$.

O problema de *bin packing* é NP-difícil, conforme Garey e Johnson [29], por exemplo. No entanto, algoritmos exatos cada vez mais eficientes, como os de geração de colunas, têm sido propostos com o objetivo de resolver ao menos certas classes de instâncias do *bin packing*.

Formulações extensivas de um problema de programação inteira podem ser obtidas como resultado de alguma decomposição do seu conjunto viável. No próximo capítulo estaremos interessados em apresentar as idéias básicas de uma decomposição e suas principais vantagens na resolução de problemas de otimização combinatória.

Capítulo 2

A Formulação

O desenvolvimento de métodos de resolução de problemas de programação linear que exploram a estrutura particular da sua formulação matemática foi proposto inicialmente por Ford e Fukerson [26]. Muitos pesquisadores consideram esse trabalho como o marco inicial da abordagem de geração de colunas e reconhecidamente inspirou Dantzig e Wolfe [20] a proporem, em 1960, um esquema de decomposição de problemas gerais de programação linear, que é usualmente denominado *decomposição de Dantzig-Wolfe.*

O trabalho pioneiro na formulação de um problema de programação inteira a partir da decomposição do problema e de sua resolução por meio de técnicas de geração (implícita) de colunas foi apresentado por Gilmore e Gomory [31, 32]. Ralphs e Galati [55], por exemplo, discorrem sobre métodos de decomposição tradicionais para problemas de programação inteira e suas aplicações no cálculo de limites do valor ótimo.

A associação da técnica de geração de colunas com algoritmos tipo *branch-and-bound*, proposta por Desrosiers, Soumis e Desrochers [22], em 1984, reforçou a importância da decomposição de Dantzig-Wolfe e, desde então, a abordagem de resolução por geração de colunas tornou-se uma das principais técnicas utilizadas para problemas de programação inteira.

Antes de abordarmos a reformulação de problemas inteiros, comentaremos sobre as idéias básicas de uma decomposição e, em particular, sobre a decomposição de Dantzig-Wolfe.

2.1 Idéias básicas de uma decomposição

Para apresentarmos a idéia geral da decomposição, consideremos um problema inteiro no seguinte formato:

$$\begin{array}{lll} \textit{minimizar} & c^T x & \\ \textit{sujeito a:} & Ax = b & \\ & Dx \leq d & \\ & x \geq 0 & \textit{e inteiro,} \end{array}$$

em que as restrições estão particionadas em uma classe de restrições $Ax = b$ e outra, mais específica, representada pelo subsistema $Dx \leq d$ e x inteiro. Este subsistema é tal que possivelmente somos capazes de enumerar todas as suas soluções e, conseqüentemente, reformularmos o problema inteiro inicial em termos dessas soluções. Claramente, foi o que fizemos para obtermos a formulação extensiva do problema de *bin packing* no final do capítulo anterior, página 18.

Citaremos, agora, três situações em que uma decomposição é geralmente usada. A primeira delas é quando as restrições $Ax = b$ dificultam em muito a resolução do problema original e o modelo é bem conhecido e facilmente resolvido sobre as restrições do subsistema $Dx \leq d$, $x \geq 0$. A segunda situação é quando as restrições $Dx \leq d$ apresentam uma estrutura bloco-diagonal (conforme Lübbecke *et al.* [45], por exemplo) e esse subsistema pode ser decomposto em subsistemas menores. Na nossa implementação, esse subsistema do problema de *bin packing* foi decomposto em restrições idênticas do tipo "mochila". E a terceira situação ocorre em algumas aplicações que só podem ser formuladas através de um número muito grande de variáveis, conforme apresentado por Barnhart *et al.* [6], por exemplo.

Muitos problemas de otimização podem ser representados como um problema de programação inteira em sua forma compacta. No entanto, os métodos do tipo *branch-and-bound*, como é o caso do *branch-and-price* que apresentaremos no próximo capítulo, não têm um desempenho aceitável para grande parte de instâncias reais. A estimativa (*bound*) do valor ótimo inteiro dada por cada relaxação linear é, em geral, muito ruim. Além disso, esses métodos são bastante ineficientes para problemas que apresentam uma estrutura simétrica, que é comum na formulação compacta de um problema inteiro. Existe simetria quando diferentes soluções, mesmo tendo valores distintos para as variáveis de decisão, correspondem a um mesmo particionamento, na prática. Neste caso, um algoritmo do tipo *branch-and-bound* pode se tornar computacionalmente infeficiente, visto que essas soluções equivalentes são exploradas em diferentes nós da árvore de *branch* associada à execução do algoritmo.

Ao decompor um problema, procuramos explorar a estrutura da matriz de restrições, de modo que a nova formulação forneça um *bound* possivelmente mais próximo do valor ótimo inteiro e que elimine as dificuldades associadas à simetria do problema. Então, neste sentido, um problema com uma formulação extensiva é um bom candidato para ser resolvido através de métodos de geração de colunas para programação inteira.

Inicialmente, descreveremos sobre o método de decomposição proposto por Dantzig e Wolfe [20] em 1960 para problemas de programação linear.

2.2 A decomposição de Dantzig e Wolfe

O método simplex, apesar de sua complexidade exponencial (Klee e Minty [41]), tem se mostrado bastante eficiente na resolução de problemas de programação linear na prática e, por isto, tem sido bastante implementado por diversos pacotes computacionais para otimização. No entanto, devido

a algum limite (de dimensão, por exemplo) previamente estabelecido ou capacidade, esses pacotes não podem ser utilizados para muitos problemas de grande porte.

A decomposição de Dantzig-Wolfe, conhecida desde 1960, foi uma importante ferramenta para resolução de problemas lineares de grande porte em computadores com capacidade de armazenamento e manipulação de dados bastante limitada. Neste sentido, ela se tornou uma metodologia auxiliar para a aplicação do método simplex a problemas de programação linear maiores. No entanto, com o desenvolvimento de novos computadores ao longo de poucas décadas, aumento de memória e de velocidade de processamento, por exemplo, os pacotes para otimização linear passaram a resolver até mesmo esses problemas lineares de grande porte. Com isto, os métodos de decomposição foram perdendo sua aplicabilidade.

Mas, em 1984, Desrosiers, Soumis e Desrochers [22] reforçaram a importância da decomposição de Dantzig-Wolfe, associando técnicas de geração de colunas com o método de *branch-and-bound* para a resolução de problemas de programação inteira. Assim, antes de tratarmos da geração de colunas no capítulo seguinte, apresentaremos mais detalhadamente a decomposição de Dantzig-Wolfe.

Basicamente, a decomposição de Dantzig-Wolfe consiste na representação de grupos de variáveis como uma combinação convexa de pontos extremos. Nessa decomposição, retiramos parte das restrições do problema de programação linear, geralmente estruturas bem definidas, e as tratamos separadamente, em um problema auxiliar. Assim, podemos reduzir em muito o número de restrições do problema original. No entanto, o problema passa a conter um número excessivamente grande de variáveis, geralmente exponencial em relação ao número de restrições no problema original.

Inicialmente, essa idéia pode parecer nada razoável, devido ao número

muito grande de variáveis na nova formulação do problema. No entanto, ela será fundamental para aplicarmos os métodos de geração de colunas na resolução de um problema de programação linear e, conseqüentemente, de programação linear inteira, como deverá ficar claro ao longo desta tese.

Dados duas matrizes $A \in \mathbb{R}^{m_1 \times n}$ e $D \in \mathbb{R}^{m_2 \times n}$, e três vetores $b \in \mathbb{R}^{m_1}$, $d \in \mathbb{R}^{m_2}$ e $c \in \mathbb{R}^n$, consideremos o seguinte problema de programação linear:

$$\begin{array}{lll} [PL] & \textit{minimizar} & z_{PL} = c^T x \\ & \textit{sujeito a:} & Ax = b \\ & & Dx \leq d \\ & & x \geq 0. \end{array}$$

A idéia básica da decomposição de Dantzig-Wolfe é representar certo conjunto convexo e limitado de soluções viáveis como uma combinação linear convexa dos pontos extremos e também das direções extremas desse conjunto. Então, inicialmente devemos escolher o grupo de restrições que definem um tal subconjunto viável convexo e limitado, substituindo-o em seguida pela combinação apropriada de suas direções extremas.

Formalmente, essa idéia está baseada no resultado enunciado a seguir, que foi proposto por Minkowski [52] e é demonstrado, por exemplo, por Nemhauser e Wolsey [53] e também por Maculan e Fampa [48].

Proposição 1 *Consideremos o conjunto* $\mathcal{X} = \{x \in \mathbb{R}^n_+;\ Dx \leq d\}$ *e denotemos por* $V(\mathcal{X}) = \{v^1,\ v^2,\ ...,\ v^p\}$ *o conjunto (finito) dos* p *pontos extremos de* $\mathcal{X}$ *e por* $R(\mathcal{X}) = \{r^1,\ r^2,\ ...,\ r^q\}$ *as suas* q *direções extremas. Então* $x \in \mathcal{X}$ *se, e somente se,*

$$x = \sum_{j=1}^{p} \lambda_j v^j + \sum_{i=1}^{q} \mu_i r^i,$$

para certos vetores

$$\lambda \in \mathbb{R}^p_+,\ \textit{tal que} \sum_{j=1}^{p} \lambda_j = 1,\quad e\ \ \mu \in \mathbb{R}^q_+.$$

A decomposição de Dantzig-Wolfe consiste na substituição das variáveis x, no problema $[PL]$, pela expressão dada na Proposição 1. A formulação resultante é o seguinte problema de programação linear:

$$
\begin{aligned}
[\overline{PM}] \quad \textit{minimizar} \quad & z_{\overline{PM}} = \sum_{j=1}^{p}(c^T v^j)\lambda_j + \sum_{i=1}^{q}(c^T r^i)\mu_i \\
\textit{sujeito a}: \quad & \sum_{j=1}^{p}(Av^j)\lambda_j + \sum_{i=1}^{q}(Ar^i)\mu_i = b \\
& \sum_{j=1}^{p}\lambda_j = 1 \\
& \lambda_j \geq 0, \quad \textit{para} \quad j = 1,\ 2,\ ...,\ p \\
& \mu_i \geq 0, \quad \textit{para} \quad i = 1,\ 2,\ ...,\ q.
\end{aligned}
$$

O problema $[\overline{PM}]$ é usualmente denominado *problema mestre*. Claramente, da relação entre x, λ e μ, temos que o valor ótimo do problema original $[PL]$ é igual ao valor ótimo do problema mestre $[\overline{PM}]$, ou seja: $z^*_{PL} = z^*_{\overline{PM}}$.

Considerando o problema mestre $[\overline{PM}]$, definamos a matriz

$$
\bar{A} = \begin{bmatrix} Av^1 & Av^2 & ... & Av^p & Ar^1 & Ar^2 & ... & Ar^q \\ 1 & 1 & ... & 1 & 0 & 0 & ... & 0 \end{bmatrix} \in \mathbb{R}^{(m_1+1)\times(p+q)}
$$

e os vetores

$$
\bar{c}^T = [c^T v^1 \quad c^T v^2 \quad ... \quad c^T v^p \quad c^T r^1 \quad c^T r^2 \quad ... \quad c^T r^q] \in \mathbb{R}^{1\times(p+q)},
$$

$$
\bar{b} = \begin{bmatrix} b \\ 1 \end{bmatrix} \in \mathbb{R}^{m_1+1}, \quad \lambda^T = [\lambda_1 \ \ \lambda_2 \ ... \ \lambda_p] \in \mathbb{R}^{1\times p} \quad \text{e} \quad \mu^T = [\mu_1 \ \ \mu_2 \ ... \ \mu_q] \in \mathbb{R}^{1\times q}.
$$

Então, em notação matricial, o problema mestre $[\overline{PM}]$ é o seguinte problema de programação linear:

$$
\begin{aligned}
\textit{minimizar} \quad & \bar{c}^T\bar{x} \\
\textit{sujeito a}: \quad & \bar{A}\bar{x} = \bar{b} \\
& \bar{x} \geq 0,
\end{aligned}
$$

em que

$$
\bar{x} = \begin{bmatrix} \lambda \\ \mu \end{bmatrix} \in \mathbb{R}^{p+q}.
$$

O método de decomposição de Dantzig-Wolfe para problemas de programação linear também tem seu equivalente para problemas de Programação Linear Inteira (PLI). Este é o assunto da próxima seção.

2.3 A Formulação de um problema de PLI

Para resolvermos um problema de programação inteira, digamos $[PI]$, por meio de algum algoritmo do tipo *branch-and-bound*, como é nosso caso e que apresentaremos no próximo capítulo, precisaremos usar um procedimento que irá gerar um limite (*bound*) que potencialmente nos leve ao valor ótimo inteiro z^*_{PI}. Para isto, o método mais comumente usado é a resolução da relaxação linear, digamos $[PL]$, do problema inteiro, obtida quando desconsideramos as restrições de integralidade de sua formulação. Assim, como o conjunto viável do problema $[PI]$ é um subconjunto das soluções viáveis de $[PL]$ e os problemas são de minimização, o valor ótimo da relaxação linear é um limite inferior (*lower bound*) para o valor ótimo do problema inteiro. Em geral, a relaxação $[PL]$ é muito mais fácil de ser resolvida que o problema inteiro $[PI]$; no entanto, seu valor ótimo pode estar bastante distante do valor ótimo inteiro. Essa "distância" é usualmente denominada *gap de integralidade*. Precisamos considerar, então, algum procedimento mais efetivo na obtenção de melhores estimativas para a solução ótima do problema inteiro. Os métodos de decomposição em problemas de programação inteira foram desenvolvidos com este objetivo.

Assim como na decomposição de Dantzig-Wolfe, a idéia básica para um problema inteiro também é a representação de um conjunto convexo através de uma combinação de pontos extremos e raios extremos. A reformulação do problema (compacto) original é ainda um problema inteiro, mas com um número de variáveis em geral excessivamente grande. Esse novo problema é usualmente denominado *problema mestre inteiro*. Como cada variável está

associada a uma coluna da matriz de restrições, podemos pensar nessa matriz como tendo, de fato, muitas colunas no problema reformulado.

Há uma equivalência entre o problema mestre inteiro e sua formulação compacta original. Assim, resolver o problema mestre inteiro resultante da decomposição é equivalente a resolver o problema inteiro original, da mesma forma que, em programação linear, uma solução para o problema mestre está associada a uma equivalente no problema original.

Consideremos o seguinte problema de programação inteira:

$$\begin{array}{lll} [PI] & \textit{minimizar} & z_{PI} = c^T x \\ & \textit{sujeito a:} & Ax = b \\ & & Dx \leq d \\ & & x \in \mathbb{Z}^n_+, \end{array}$$

onde as matrizes $A \in \mathbb{Q}^{m_1 \times n}$ e $D \in \mathbb{Q}^{m_2 \times n}$, com $m_1 + m_2 = m$, e os vetores $b \in \mathbb{Q}^{m_1}$, $d \in \mathbb{Q}^{m_2}$ e $c \in \mathbb{Q}^n$ são dados. Analogamente ao caso da decomposição de Dantzig-Wolfe, descrita na seção anterior, suponhamos, também, que o conjunto $\mathcal{X} = \{x \in \mathbb{Z}^n_+;\ Dx \leq d\}$ seja tal que possamos considerar implicitamente todas as soluções inteiras nele contidas.

Para chegarmos à formulação do problema mestre inteiro, enunciaremos o seguinte resultado em relação ao conjunto de pontos inteiros $\mathcal{X}$, conforme enunciado por Vanderbeck [64] e demonstrado por Nemhauser e Wolsey [53], por exemplo.

Proposição 2 *Se* $\mathcal{S} = \{x \in \mathbb{R}^n_+;\ Dx \leq d\} \neq \emptyset$ *e* $\mathcal{X} = \mathcal{S} \cap \mathbb{Z}^n_+$, *onde* $D \in \mathbb{Q}^{m_2 \times n}$ *e* $d \in \mathbb{Q}^{m_2}$, *então existem um conjunto finito* $\mathcal{Q} = \{\hat{x}^1, \hat{x}^2, ..., \hat{x}^p\} \subseteq \mathcal{X}$ *de pontos inteiros e um conjunto finito de direções inteiras* $\mathcal{R} = \{\bar{x}^1, \bar{x}^2, ..., \bar{x}^q\} \setminus \{0\}$ *de* $\mathcal{S}$ *tais que*

$$\mathcal{X} = \{x \in \mathbb{R}^n_+;\ x = \sum_{j=1}^{p} \lambda_j \hat{x}^j + \sum_{i=1}^{q} \mu_i \bar{x}^i,\ \sum_{j=1}^{p} \lambda_j = 1,\ \lambda \in \mathbb{Z}^p_+,\ \mu \in \mathbb{Z}^q_+\}.$$

Portanto, podemos gerar todo o conjunto poliedral $\mathcal{X}$ tomando algum ponto em $\mathcal{Q}$ e uma combinação inteira de direções em $\mathcal{R}$.

Quando $\mathcal{X}$ for limitado, teremos $\mathcal{R} = \emptyset$ e $\mathcal{Q}$ será o próprio conjunto $\mathcal{X}$. Neste caso:

$$\begin{aligned}\mathcal{X} &= \{x \in \mathbb{Z}^n_+;\ Dx \leq d\} = \\ &= \{x \in \mathbb{R}^n_+;\ x = \sum_{j=1}^{p} \lambda_j \hat{x}^j,\ \sum_{j=1}^{p} \lambda_j = 1,\ \lambda \in \{0,1\}^p\} = \\ &= \{\hat{x}^1,\ \hat{x}^2,\ ...,\ \hat{x}^p\}.\end{aligned}$$

Por outro lado, se $\mathcal{X}$ for ilimitado, $\mathcal{R}$ será o conjunto de suas direções extremas e teremos $\mathcal{Q} = \mathcal{X} \cap \{x \in \mathbb{R}^n_+;\ x = \sum_{j=1}^{p} \beta_j \hat{x}^j + \sum_{i=1}^{q} \gamma_i \bar{x}^i,\ \sum_{j=1}^{p} \beta_j = 1,\ \beta_j \geq 0$ para todo $j = 1, 2, ..., p$, e $0 \leq \gamma_i < 1$ para todo $i = 1, 2, ..., q\}$, onde $\hat{x}^1, \hat{x}^2, ..., \hat{x}^p$ são os pontos extremos (vértices) de $\mathcal{S} = \{x \in \mathbb{R}^n_+;\ Dx \leq d\}$. Se, além disso, $\mathcal{S}$ for um cone ($d = 0$), então os pontos em $\mathcal{Q}$ podem ser vistos como direções extremas e, portanto, incluídos em $\mathcal{R}$, levando a $\mathcal{Q} = \emptyset$. Lembremos que, sem perda de generalidade, podemos supor qualquer $\bar{x} \in \mathcal{R}$ como sendo um vetor de inteiros, visto que $\mathcal{S}$ é um conjunto de racionais. Observemos, ainda, que se $\mathcal{X} = \mathcal{S} \cap \{0,1\}^n$, então $\mathcal{X}$ coincide com os pontos extremos da sua envoltória convexa, denotada por $conv(\mathcal{X})$.

Definindo $\hat{c}_j = c^T \hat{x}^j$ e $\hat{a}_j = A\hat{x}^j$ para cada $j \in \{1,\ 2,\ ...,\ p\}$, e $\bar{c}_i = c^T \bar{x}^i$ e $\bar{a}_i = A\bar{x}^i$ para cada $i \in \{1,\ 2,\ ...,\ q\}$ e substituindo $x \in \mathcal{X}$ por $x = \sum_{j=1}^{p} \lambda_j \hat{x}^j + \sum_{i=1}^{q} \mu_i \bar{x}^i$, $\sum_{j=1}^{p} \lambda_j = 1$, $\lambda \in \mathbb{Z}^p_+$, $\mu \in \mathbb{Z}^q_+$ no problema inteiro $[PI]$, teremos uma nova formulação, o *problema mestre inteiro*, a saber:

$$\text{[PM] minimizar} \quad z_{PM} = \sum_{j=1}^{p} \lambda_j \hat{c}_j + \sum_{i=1}^{q} \mu_i \bar{c}_i$$

$$sujeito\ a: \quad \sum_{j=1}^{p} \lambda_j \hat{a}_j + \sum_{i=1}^{q} \mu_i \bar{a}_i = b$$

$$\sum_{j=1}^{p} \lambda_j = 1$$

$$\lambda_j \in \mathbb{Z}_+, \qquad \text{para } j = 1,\ 2,\ ...,\ p$$

$$\mu_i \in \mathbb{Z}_+, \qquad \text{para } i = 1,\ 2,\ ...,\ q.$$

A restrição de convexidade $\sum_{j=1}^{p} \lambda_j = 1$, juntamente com a restrição de integralidade $\lambda \in \mathbb{Z}_+^p$, impõem que tenhamos $\lambda \in \{0,1\}^n$. Assim, em qualquer solução viável, existirá um índice $k \in \{1,\ 2,\ ...,\ n\}$ tal que $\lambda_k = 1$ e as outras componentes do vetor λ serão iguais a zero.

Resolver o problema mestre inteiro $[PM]$ é equivalente a resolver a formulação original $[PI]$. No entanto, as relaxações lineares dessas duas formulações são distintas. Em geral, a relaxação linear do problema mestre inteiro nos fornece uma estimativa melhor (*lower bound*) para o valor da solução ótima que a relaxação do problema no seu formato original $[PI]$.

A formulação compacta, o problema original $[PI]$, e a formulação extensiva, o problema mestre $[PM]$, são formulações do mesmo problema de programação inteira. Elas têm as mesmas soluções inteiras viáveis e, também, o mesmo valor ótimo. No entanto, a representação das solução é diferente e uma solução no problema original pode não corresponder a uma única solução no problema mestre correspondente, por exemplo. Ou seja, considerando a notação que apresentamos na Proposição 2, observamos que se $\mathcal{Q} \neq \emptyset$ e $\mathcal{R} \neq \emptyset$ (ou $\mathcal{X}$ não é limitado, ou não é um cone), e existem $\hat{x} \in \mathcal{Q}$ e $\bar{x} \in \mathcal{R}$ tais que $\hat{x} + \bar{x} \in \mathcal{Q}$, então a transformação de uma solução x de $[PI]$ para alguma solução λ de $[PM]$ não é única para certos pontos de $\mathcal{X}$.

No entanto, as duas formulações, original e mestre, diferem em suas relaxações lineares. Os limites (*bounds*) para a solução ótima inteira fornecidas pelas relaxações lineares de $[PI]$ e $[PM]$ são, respectivamente:

$$\bar{z}_{PI} = \min\{c^T x;\ Ax = b,\ Dx \leq d,\ x \geq 0\}$$

e

$$\bar{z}_{PM} = \min\{c^T x;\ Ax = b,\ x \in conv(\mathcal{X})\},$$

onde $\mathcal{X} = \{x \in \mathbb{Z}_+^n;\ Dx \leq d\}$. Portanto, denotando por z_{PI}^* o valor da

solução ótima inteira, teremos a seguinte relação:

$$\bar{z}_{PI} \leq \bar{z}_{PM} \leq z^*_{PI}.$$

Quando a formulação é tal que o subsistema $\mathcal{S} = \{x \in \mathbb{R}^n_+;\ Dx \leq d\}$ tem a propriedade da integralidade, ou seja, $conv(\{x \in \mathbb{Z}^n_+;\ Dx \leq d\}) = \{x \in \mathbb{R}^n_+;\ Dx \leq d\}$, teremos o mesmo valor ótimo para as relaxações lineares do problema original e do problema mestre: $\bar{z}_{PI} = \bar{z}_{PM}$. Neste caso, em um algoritmo *branch-and-price*, qualquer esquema de ramificação (*branching*) que resulta na modificação da estrutura do subproblema pode destruir sua propriedade de integralidade. No entanto, no caso da nossa implementação, a estratégia de *branching* que adotamos não irá modificar essa estrutura, como veremos no Capítulo 4.

Além disso, quando $conv(\{x \in \mathbb{Z}^n_+;\ Ax = b,\ Dx \leq d\}) = \{x \in \mathbb{R}^n_+;\ Ax = b\} \cap conv(\{x \in \mathbb{Z}^n_+;\ Dx \geq d\})$, então $\bar{z}_{PM} = z_{PI}$. No contexto de um algoritmo *branch-and-price*, este caso não irá exigir qualquer esquema de *branching*.

Logo, o emprego da reformulação do problema inteiro original $[PI]$ como um problema mestre $[PM]$ é aconselhado nos casos em que a formulação original do subsistema não apresenta a propriedade da integralidade e a relaxação linear do problema mestre não fornece uma solução inteira. Então, tipicamente, teremos a seguinte relação entre os valores ótimos:

$$\bar{z}_{PI} < \bar{z}_{PM} < z_{PI}.$$

A decomposição mais tradicional de um problema inteiro está baseada no Teorema de Minkowski, conforme enunciado através da Proposição 1. Essa decomposição, usualmente denominada *convexificação*, difere da que apresentamos anteriormente para problemas inteiros, que é uma *discretização* do conjunto $\mathcal{X}$. Enquanto na decomposição tradicional a reformulação é da

envoltória convexa do subsistema $\mathcal{X}$, $conv(\mathcal{X})$, na decomposição por discretização a reformulação é do próprio $\mathcal{X}$. Os pontos em $conv(\mathcal{X})$ são expressos como uma combinação convexa dos seus pontos extremos e das suas direções extremas, e os coeficientes da decomposição não estão restritos a serem inteiros, como exposto por Lübbecke e Desrosiers [45]. Essas duas alternativas para uma decomposição, a discretização e a convexificação, fornecem a mesma relaxação linear do problema mestre. Além disso, quando $\mathcal{X} \subseteq \{0,1\}^n$, como ocorre no caso particular do problema de particionamento da nossa implementação, a convexificação e a discretização são coincidentes.

Em suma, o método de decomposição de Dantzig-Wolfe para programação inteira consiste na reformulação de um subsistema $\mathcal{X}$ do problema $[PI]$, seguindo o resultado dado pela Proposição 2. Conseqüentemente, essa reformulação de $\mathcal{X}$ leva a uma reformulação do problema inteiro $[PI]$. Uma característica bastante útil desta reformulação de $\mathcal{X}$ é que sua relaxação linear fornece a envoltória convexa, $conv(\mathcal{X})$, o que explica a "qualidade" do limite fornecido pela relaxação linear do problema mestre. No entanto, para obtermos a propriedade de integralidade com a reformulação do subsistema, pagamos um alto preço em termo de número de variáveis, geralmente exponencial, no problema reformulado. A conseqüência é que precisamos tratá-las implicitamente por algum procedimento de geração de colunas.

Assim, no próximo capítulo, estaremos interessados em um método de geração de colunas para problemas de otimização combinatória.

Capítulo 3

A Geração de Colunas

Nosso objetivo, neste capítulo, é apresentarmos os fundamentos dos algoritmos do tipo geração de colunas para problemas de programação linear e, em particular, para problemas de programação inteira.

Logo no início do desenvolvimento da teoria da programação linear, na década de 40, já começaram também os estudos de problemas que apresentavam variáveis discretas [16]. Isto levou ao desenvolvimento dos primeiros algoritmos que utilizam a idéia de se adicionar novas desigualdades à formulação do problema, os chamados *algoritmos de planos de cortes*, propostos por Dantzig, Fulkerson e Johnson [18] e Gomory [36, 35]. Em seguida, Land e Doig [42] introduziram um algoritmo enumerativo, denominado *branch-and-bound*, que utiliza estimativas no valor da solução ótima para evitar a enumeração de todas as soluções possíveis. Desde então, várias abordagens têm sido desenvolvidas para a resolução de problemas inteiros: enumeração implícita [5], decomposição [8], relaxação lagrangeana [30] e heurísticas [65], por exemplo. Atualmente, grande parte das pesquisas têm se concentrado no desenvolvimento de técnicas e algoritmos para classes de problemas bem particulares. Muitos dos algoritmos desenvolvidos estão baseados nas idéias do método *branch-and-bound*, que apresentaremos mais adiante neste capítulo.

Os algoritmos exatos mais usuais e de maior sucesso na resolução de

um problema de programação inteira estão fundamentados na obtenção de sucessivas soluções de problemas mais fáceis, denominados *relaxações*. Essas soluções fornecem os *limitantes*, também denominados *bounds*, do valor da solução ótima do problema inteiro. Quando estivermos certos de que o valor da solução do problema inteiro não será inferior a determinado valor já conhecido, dizemos que este valor é um *limitante inferior* para o problema inteiro, ou ainda um *lower bound*. De forma análoga, dizemos que um certo valor é um *limitante superior*, ou *upper bound*, quando podemos afirmar que a solução ótima inteira não será maior que esse valor. Esses limitantes é que irão "direcionar" uma enumeração implícita de todas as soluções viáveis numa técnica denominada *branch-and-bound* e que, neste capítulo, será apresentada no contexto da otimização combinatória.

Os métodos *branch-and-bound* são mais eficientes se tivermos boas estimativas (*bounds*) para o problema inteiro. A abordagem convencional desses métodos envolve o uso da *relaxação linear* do problema inteiro para a obtenção desses *bounds*. A resolução de um problema de programação inteira resultante da relaxação de todas as restrições de integralidade da sua formulação, é geralmente fácil (se usarmos o algoritmo simplex, por exemplo), mas a estimativa obtida para o problema inteiro normalmente é ruim.

Existem outras técnicas muito utilizadas na obtenção de *bounds* melhores. A *relaxação lagrangeana* consiste na relaxação de algumas restrições da formulação do problema, em vez de todas as suas restrições de integralidade. Fleuren [25], por exemplo, usa a relaxação lagrangeana para determinar *lower bounds* para problemas de particionamento. Beasley e Cao [7] aplica a relaxação lagrangeana para um problema mais geral: o *crew scheduling*. Essa técnica é descrita, também, por Reinoso e Maculan [56] e por Krieken *et al.* [40]. Por outro lado, os *métodos poliedrais* procuram melhorar as relaxações lineares acrescentando apropriadamente outras inequações à formulação do problema.

No entanto, a técnica que estamos interessados nesta tese, e que também fornece boas aproximações para a solução ótima do problema inteiro, parte de uma formulação mais extensiva para o problema. Neste sentido, a nova formulação contém um número muito maior de variáveis.

3.1 Algoritmos em programação linear inteira

Vários resultados teóricos têm sido obtidos e técnicas gerais têm sido desenvolvidas para o problema de programação linear inteira. No entanto, esse problema ainda representa um modelo muito abrangente e, portanto, grande parte dos estudos atuais têm se concentrado no desenvolvimento de algoritmos especiais para subclasses bem particulares desse modelo geral.

Muitos desses algoritmos estão fundamentados em alguma metodologia enumerativa especial e utilizam alguma "relaxação" do problema inteiro original para obter, em um tempo aceitável, uma estimativa para o valor ótimo em cada caso da enumeração. A idéia da relaxação consiste na substituição do problema inteiro de minimização (ou de maximização), que geralmente é muito "difícil", por um outro problema de otimização mais simples e com valor ótimo menor (maior) ou igual ao do problema inteiro original. Nesta tese, descreveremos a respeito dos métodos enumerativos do tipo *branch-and-bound* e, em particular, o *branch-and-price* e comentamos sobre a relaxação linear, apenas.

Podemos "medir" a qualidade de uma certa estimativa para o valor ótimo com base, por exemplo, no quanto essa estimativa está próxima do valor ótimo do problema. Geralmente, os algoritmos que garantem as melhores estimativas para o valor ótimo de um problema são os que levam um tempo maior para obtê-las. Por outro lado, algoritmos com menor tempo de convergência, muitas vezes não garantem boas estimativas. Assim, a eficiência de um método enumerativo muitas vezes é determinada pela relação dada

entre o tempo gasto para o cálculo dos valores estimados e sua qualidade.

Diante deste fato, as pesquisas atualmente desenvolvidas na Otimização Combinatória têm se concentrado no desenvolvimento e estudo de formulações que potencialmente forneçam estimativas de valores ótimos cada vez melhores. Umas das principais abordagens que têm sido seguidas (mas não está no escopo desta tese) procura definir certas classes de desigualdades que serão adicionadas à relaxação linear $[PL]$ de um problema inteiro $[PI]$ numa tentativa de fazer o conjunto viável de $[PL]$ se aproximar cada vez mais da envoltória convexa das soluções de $[PI]$. Uma outra abordagem advém da decomposição do conjunto viável do problema em formatos equivalentes; em particular, da decomposição de Dantzig-Wolfe [20], que já descrevemos no capítulo anterior.

Apresentamos, agora, os fundamentos de um algoritmo *branch-and-bound*.

3.2 O algoritmo *branch-and-bound*

Como já citamos, o algoritmo *branch-and-bound* foi proposto por Land e Doig [42] para a resolução de problemas de programação inteira. Ele utiliza a estratégia "dividir para conquistar", no sentido em que está baseado na inspeção de tão somente partes do conjunto de soluções viáveis e, assim, obtendo estimativas para o valor da solução ótima do problema inteiro.

Dados uma matriz $A \in \mathbb{Q}^{m \times n}$ e vetores $b \in \mathbb{Q}^m$ e $c \in \mathbb{R}^n$, com $0 < m \leq n$, consideremos, nesta seção, o seguinte problema de programação inteira:

$$\begin{array}{lll} [PI] & \textit{minimizar} & z_{PI} = c^T x \\ & \textit{sujeito a:} & Ax = b \\ & & x \geq 0 \quad \textit{e inteiro}, \end{array}$$

cujo conjunto viável é

$$S_{PI} = \{x \in \mathbb{Z}_+^n;\ Ax = b\}.$$

Consideremos, também, a relaxação linear do problema $[PI]$, a saber:

$$\begin{array}{lll} [PL] & \textit{minimizar} & z_{PL} = c^T x \\ & \textit{sujeito a:} & Ax = b \\ & & x \geq 0, \end{array}$$

cujo conjunto viável é

$$\mathcal{S}_{PL} = \{x \in \mathbb{R}^n_+;\ Ax = b\}.$$

Uma operação de "ramificação", ou *branching*, consiste na definição de k poliedros $\mathcal{S}^1_{PL}$, $\mathcal{S}^2_{PL}$, ..., $\mathcal{S}^k_{PL}$, todos contidos em $\mathcal{S}_{PL}$, tais que $\cup^k_{i=1}(\mathcal{S}^i_{PL} \cap \mathbb{Z}^n_+) = \mathcal{S}_{PI}$. Assim, o problema inteiro original $[PI]$ definido sobre o poliedro $\mathcal{S}_{PL}$ pode ser resolvido através da resolução de k subproblemas inteiros em poliedros menores.

Consideremos $i = 1,\ 2,\ ...,\ k$. Uma "poda", ou *bounding*, consiste em encontrarmos a melhor solução contida em cada poliedro $\mathcal{S}^i_{PL}$ já obtido na operação de *branching*. O *bounding*, então, fornece um modo de calcularmos uma estimativa inferior para o valor de qualquer solução viável para o problema $[PI]$ definido sobre $\mathcal{S}^i_{PL}$, uma vez que para cada $\mathcal{S}^i_{PL} \subset \mathcal{S}_{PL}$ teremos a seguinte relação:

$$\min\{c^T x;\ x \in \mathcal{S}^i_{PL}\} \leq \min\{c^T x;\ x \in \mathcal{S}_{PI} \cap \mathcal{S}^i_{PL}\}.$$

O *branch-and-bound* é um algoritmo iterativo que mantém uma família $\mathcal{F}$ de subconjuntos de $\mathcal{S}_{PL}$, a melhor solução viável conhecida $x^* \in \mathcal{S}_{PI}$ e o seu valor $z^* = c^T x^*$ em cada uma das suas iterações. Inicialmente, quando $\mathcal{F} = \{\mathcal{S}_{PL}\}$, o algoritmo não dispõe de uma solução ótima e teremos $z^* = \infty$. Ao término do algoritmo, se existir alguma solução ótima para o problema $[PI]$, x^* será uma tal solução.

Consideremos, arbitrariamente, uma iteração i do algoritmo *branch-and-bound*. Seja $\mathcal{S}^i_{PL}$ um subconjunto obtido a partir da família corrente $\mathcal{F}$

de subconjuntos viáveis da relaxação linear $[PL]$. Então, o algoritmo deverá resolver o seguinte problema:

$$\begin{array}{lll} [PL]^i & \textit{minimizar} & z^i_{PL} = c^T x \\ & \textit{sujeito a:} & x \in \mathcal{S}^i_{PL}. \end{array}$$

Se o problema $[PL]^i$ é inviável ou se $z^i_{PL} \leq z^*$, então não existe qualquer solução viável $x \in \mathcal{S}_{PI} \cap \mathcal{S}^i_{PL}$ cujo valor da função objetivo seja menor que o melhor valor já conhecido até a iteração corrente. Assim, o algoritmo prossegue para a iteração $i+1$. Caso contrário, se a solução ótima $\hat{x}^i$ do problema $[PL]^i$ é uma solução viável para o problema $[PI]$ e $z^i_{PL} \leq z^*$, então devemos fazer a seguinte atualização: $x^* = \hat{x}^i$ e $z^* = z^i_{PL}$. Além disso, todos os subconjuntos $\mathcal{S}^i_{PL} \in \mathcal{F}$ tais que tenhamos $z^i_{PL} \geq z^*$ serão excluídos de $\mathcal{F}$. Por outro lado, se a solução ótima $\hat{x}^i$ do problema $[PL]^i$ não é viável para $[PI]$, devemos executar uma operação de *branching* no conjunto $\mathcal{S}^i_{PL}$, inserindo em $\mathcal{F}$ os subconjuntos resultantes, e prosseguir para a iteração $i+1$ do algoritmo. O critério de parada para o algoritmo é que tenhamos $\mathcal{F} = \emptyset$. Quando esse critério for satisfeito, o problema $[PI]$ será inviável se tivermos $z^* = \infty$; caso contrário, z^* será o seu valor ótimo.

Uma alternativa complementar para esse esquema padrão, é o uso de alguma heurística para encontrar uma solução inicial x^* associada a um valor z^*, antes mesmo da primeira iteração do algoritmo *branch-and-bound*. Isto, potencialmente, diminue o número total de iterações do algoritmo. No entanto, nesta tese, estaremos interessados apenas em algoritmos exatos.

Ao executarmos um algoritmo *branch-and-bound*, podemos associá-lo a uma árvore T. Seja N a última iteração de uma execução do algoritmo. A cada iteração $i \in \{1, 2, ..., N\}$, associamos um nó $v^i \in T$. O nó v^1 é a raiz da árvore. Tomemos, numa iteração $j > 1$, o subconjunto $\mathcal{S}^j_{PL} \in \mathcal{F}$ que tenha sido obtido a partir do *branching* de um conjunto $\mathcal{S}^i_{PL}$ na iteração $i < j$. Então, o nó j da árvore será um descendente (filho) para o nó i. Essa árvore é usualmente denominada *árvore de branch-and-bound*, ou simplesmente *árvore*

de branching.

Como apresentaremos no próximo capítulo, sobre nossa implementação, várias abordagens poderão ser adotadas nesse esquema de enumeração. O critério de escolha do subconjunto $\mathcal{S}_{PL}^i \in \mathcal{F}$ a cada iteração i e o esquema de *branching* de $\mathcal{S}_{PL}^i$ são determinantes para o tamanho da árvore de *branching* T e, portanto, para o tempo de execução do algoritmo. A escolha desses esquemas em geral é ditada pela experiência e conhecimento a respeito da estrutura particular do problema a ser resolvido.

Um esquema de *branching* clássico particiona o conjunto viável em dois subconjuntos através da imposição, respectivamente, das restrições $x_j \leq \lfloor \hat{x}_j \rfloor$ e $x_j \geq \lceil \hat{x}_j \rceil$, onde $\hat{x}_j$ é uma variável de valor fracionário da solução da relaxação linear $[PL]^i$ na iteração corrente i. A árvore de *branching*, neste caso, é binária.

Embora o critério de escolha do subconjunto $\mathcal{S}_{PL}^i \in \mathcal{F}$ a cada iteração i também possa depender da estrutura particular do problema, existem outros esquemas bastante usuais para qualquer aplicação. Esses esquemas dizem respeito à forma como a árvore de *branching* é percorrida. Neste sentido, a busca em profundidade (*depth-first*) escolhe o subconjunto incluído mais recentemente no conjunto $\mathcal{F}$; a busca em largura (*breadth-first*) escolhe sempre o mais antigo deles; e o *best-first* escolhe o subconjunto que fornece a melhor estimativa para o valor da solução ótima do problema inteiro, ou seja, o subconjunto $\mathcal{S}_{PL}^i \in \mathcal{F}$ tal que o valor z_{PL}^i seja mínimo.

Muitas vezes, um problema de programação linear inteira contém um número excessivamente grande de variáveis. Neste caso, o algoritmo *branch-and-bound* precisa ser apropriadamente adaptado através de um refinamento que resulta no *algoritmo branch-and-price*, que apresentaremos mais adiante neste capítulo.

Inicialmente, tratamos de geração de colunas em problemas de programação linear na próxima seção.

3.3 Geração de colunas em programação linear

Como já apresentamos no capítulo anterior, um problema de programação linear pode ser reformulado como um problema mestre, com muitas variáveis, a partir da decomposição de Dantzig-Wolfe. Definindo um conjunto de índices $J = \{1,\ 2,\ ...,\ p\}$, consideremos que esse problema mestre seja o seguinte problema linear:

$$[PL]\ \ minimizar\ \ z_{PL} = \sum_{j \in J} c_j x_j$$

$$sujeito\ a: \quad \sum_{j \in J} a_j x_j = b$$

$$x_j \geq 0, \quad para\ todo\ j \in J,$$

onde $b \geq 0 \in \mathbb{R}^m$, $a_j \in \mathbb{R}^m$ e $c_j \in \mathbb{R}$, para cada $j \in J$. Suponhamos que $a_j \in \mathcal{K} = \{a_1,\ a_2,\ ...,\ a_p\}$ e que existe uma função $f : \mathcal{K} \rightarrow \mathbb{R}$ tal que $c_j = f(a_j)$.

Como já sabemos, resolver o problema $[PL]$ não é uma tarefa fácil, visto que ele geralmente tem um número excessivamente grande de variáveis. Diante disto, muitas vezes é impossível considerar todos os dados do problema numa resolução prática. Então, a estratégia a ser adotada será resolver o problema gerando as colunas apenas se (e quando) elas forem realmente necessárias ao longo das iterações do algoritmo. Essa técnica é sugestivamente chamada *geração de colunas.*

Nessa abordagem de resolução, devemos conhecer, inicialmente, uma solução básica viável para o problema mestre linear $[PL]$. Tal solução inicial pode ser gerada através, por exemplo, da Fase 1 do método simplex. Ou seja,

acrescentamos apropriadamente à formulação do problema algumas variáveis artificiais e usamos um procedimento de geração de colunas para resolver o problema, agora aumentado, com uma função objetivo artificial que penaliza a presença das variáveis artificiais na base. Por outro lado, poderíamos gerar uma solução viável através de alguma heurística ou ainda combinando o problema Fase 1 com o Fase 2.

Inicialmente, a formulação do problema mestre contém apenas um subconjunto de suas possíveis colunas: as colunas que fazem parte da base viável inicial e, eventualmente, mais algumas outras colunas. Esse problema é usualmente denominado *problema mestre linear restrito.* Outras colunas são geradas conforme descreveremos a seguir.

Antes, considerando o problema $[PL]$, seja $J' \subseteq J$ um subconjunto de índices e o respectivo problema mestre restrito, a saber:

$$
\begin{aligned}
[PL]' \quad \textit{minimizar} \quad & z_{PL} = \sum_{j \in J'} c_j x_j \\
\textit{sujeito a:} \quad & \sum_{j \in J'} a_j x_j = b \\
& x_j \geq 0, \qquad \textit{para todo } j \in J'.
\end{aligned}
$$

Então, a partir de determinada base associada a uma solução viável para o problema $[PL]'$, a idéia básica do *branch-and-bound* consiste em tentar encontrar alguma coluna cujo custo reduzido sugere sua entrada na base. Se alguma coluna com tal custo reduzido não existir, então a base corrente é ótima para o problema restrito $[PL]'$ também para o problema original $[PL]$. A maior parte dos algoritmos baseados no *branch-and-bound* tomam a solução dual associada à base corrente para ser utilizada num procedimento particular de geração de colunas ou para se certificar da otimalidade do problema mestre (primal). Formalizaremos, agora, essas idéias.

Seguindo as definições e notações que já apresentamos no Capítulo 1, página 11, seja $B = [a_{B(1)}\ a_{B(2)}\ \ldots\ a_{B(m)}]$, tal que $det(B) \neq 0$, uma base

associada a uma solução básica viável $x^T = [x_B^T \; x_N^T] \in \mathbb{R}^{1\times n}$, onde $x_B = B^{-1}b \geq 0$ e $x_N = 0$. Sabemos que a base viável corrente está associada a uma solução ótima apenas quando todos os custos reduzidos são não negativos, isto é equivalente a dizer que a solução dual correspondente também é viável.

Então, para fazermos essa verificação, poderemos calcular, para cada $j \in J$, o custo reduzido dado por $\bar{c}_j = c_j - u^T a_j$, onde $u = c_B^T B^{-1} \in \mathbb{R}^m$ é a solução dual associada à base B, com $c_B^T = [c_{B(1)} \; c_{B(2)} \; \ldots \; c_{B(m)}]$. No entanto, o cálculo dos valores c_j para todo $j \in J$ em geral é impraticável, visto que a cardinalidade do conjunto J é muito grande.

Assim, em vez de calcularmos $\bar{c}_j$ para todo índice $j \in J$, poderemos calcular apenas o menor desses valores, ou seja, lembrando que $a_j \in \mathcal{K}$ e $c_j = f(a_j)$ para cada $j \in J$, devemos resolver o seguinte problema de otimização, usualmente denominado *problema auxiliar* (Maculan e Fampa [48]), ou ainda *subproblema pricing*:

$$\begin{array}{lll} [SP] & \textit{minimizar} & z_{SP} = f(a_j) - u^T a_j \\ & \textit{sujeito a:} & a_j \in \mathcal{K}. \end{array}$$

Portanto, para testarmos a otimalidade do problema $[PL]$, basta conhecermos as colunas $a_j \in \mathcal{K}$ implicitamente e obtermos uma cujo custo reduzido é o menor. Se esse custo reduzido não for negativo, então as condições de otimalidade do problema $[PL]$ também foram satisfeitas e o problema mestre restrito $[PL]'$ foi resolvido sem a necessidade de especificar todas as suas colunas ou sem a necessidade de resolver diretamente a fomulção completa $[PL]$.

Caso contrário, a solução do subproblema *pricing* $[SP]$ fornece ao menos uma coluna que não faz parte da base B, digamos $a_q \in \mathcal{K}$, indicando inviabilidade dual, isto é:

$$z_{SP} = f(a_q) - u^T a_q < 0.$$

Se isto ocorrer, poderemos tomar a coluna a_q para entrar na base e

fazer o pivoteamento, como no método simplex, obtendo uma nova solução básica para $[PL]'$ não pior que a anterior (se esta solução não for degenerada, a nova solução será garantidamente melhor). Assim, à cada iteração de um algoritmo de geração de colunas usual, o problema mestre restrito terá seu número de colunas aumentado e o novo problema mestre poderá ser resolvido usando algum método para programação linear: o simplex, por exemplo.

É claro que, para se ter um desempenho aceitável, um algoritmo de geração de colunas exige que o subproblema seja resolvido eficientemente e que a solução do problema mestre linear seja obtida antes que um número excessivamente grande de colunas seja adicionado à formulação do problema restrito.

Já apresentamos, na seção 3.2, página 36, as idéias do algoritmo *branch-and-bound* para resolvermos um problema inteiro, cuja execução pode ser vista como a resolução de sucessivos subproblemas lineares nos nós de uma árvore de *branching*. Por outro lado, no Capítulo 2, página 27, vimos que um problema de programação inteira pode ser reformulado como um problema mestre que, geralmente, tem um número muito grande de variáveis. Diante disso, podemos pensar em combinar o algoritmo *branch-and-bound* para um problema inteiro com algum algoritmo de geração de colunas aplicado aos subproblemas lineares (com muitas variáveis) em cada nó da árvore de *branching*. Este é o nosso assunto da próxima seção.

3.4 O algoritmo *branch-and-price*

Nosso principal objetivo, aqui, é apresentar um algoritmo exato de geração de colunas para resolver problemas de Programação Inteira (PI) que envolvem um número muito grande de variáveis. Inicialmente, apresentaremos uma formulação de um problema de PI apropriada para os métodos de geração de colunas: o problema mestre. Em seguida, discorreremos sobre

os principais resultados teóricos envolvidos na abordagem de resolução que estamos utilizando na nossa implementação particular para o problema de particionamento.

O algoritmo de geração de colunas para programação inteira que descrevemos nesta tese é um método de otimização exato que integra geração de colunas com *branch-and-bound*. A abordagem adotada para a estimativa (*lower bound*) do valor ótimo do problema linear inteiro e, portanto, para a poda do corrente nó na árvore de *branch-and-bound*, está baseada na relaxação linear do problema mestre. Devido ao número excessivamente grande de variáveis no problema mestre, um algoritmo de geração de colunas é usado em cada nó da árvore para resolver um problema mestre formado apenas por certas colunas do problema mestre conceitual.

Como já apresentamos no capítulo anterior, um problema de programação inteira pode ser decomposto e reformulado como um problema mestre inteiro. Consideremos, novamente, o problema inteiro $[PI]$ que já apresentamos no capítulo anterior, a saber:

$$\begin{array}{lll} [PI] & \textit{minimizar} & z_{PI} = c^T x \\ & \textit{sujeito a:} & Ax = b \\ & & Dx \leq d \\ & & x \in \mathbb{Z}^n_+, \end{array}$$

onde as matrizes $A \in \mathbb{Q}^{m_1 \times n}$ e $D \in \mathbb{Q}^{m_2 \times n}$, e os vetores $b \in \mathbb{Q}^{m_1}$, $d \in \mathbb{Q}^{m_2}$ e $c \in \mathbb{Q}^n$ são dados. Além disso, sem perda de generalidade e para simplicidade de notação, suponhamos que o conjunto $\mathcal{X} = \{x \in \mathbb{Z}^n_+;\ Dx \leq d\}$ seja limitado. Recordemos, então, que o conjunto de direções extremas (raios extremos) de $\mathcal{X}$ é vazio e, assim, teremos:

$$\begin{aligned} \mathcal{X} &= \{x \in \mathbb{Z}^n_+;\ Dx \leq d\} = \\ &= \{x \in \mathbb{R}^n_+;\ x = \sum_{j=1}^{p} \lambda_j \hat{x}^j,\ \sum_{j=1}^{p} \lambda_j = 1,\ \lambda \in \{0,1\}^p\} = \\ &= \{\hat{x}^1,\ \hat{x}^2,\ ...,\ \hat{x}^p\}. \end{aligned}$$

Utilizando, agora, as notações

$$\hat{c}_j = c^T \hat{x}^j, \quad \hat{a}_j = \begin{bmatrix} A\hat{x}^j \\ 1 \end{bmatrix} \in \mathbb{R}^m \text{ e } \hat{b} = \begin{bmatrix} b \\ 1 \end{bmatrix} \in \mathbb{R}^m,$$

onde $m = m_1 + 1$ e $j \in J = \{1, 2, ..., p\}$, reescrevemos também o problema mestre associado a $[PI]$ como se segue:

$$[PM] \quad minimizar \quad z_{PM} = \sum_{j \in J} \lambda_j \hat{c}_j$$

$$sujeito\ a: \quad \sum_{j \in J} \lambda_j \hat{a}_j = \hat{b}$$

$$\lambda_j \in \{0, 1\}, \quad para\ j \in J.$$

Assim como no *branch-and-bound* tradicional, podemos pensar numa árvore associada à execução do algoritmo *branch-and-price*. Desta forma, a cada nó t da árvore de *branching* está associado um em problema mestre inteiro restrito $[PM]^t$, em que J é substituído por $J' \subseteq J$ na formulação $[PM]$ acima.

A relaxação linear de $[PM]^t$, em um nó t da árvore de *branching*, é

$$[\overline{PM}]^t \quad minimizar \quad \bar{z}^t_{PM} = \sum_{j \in J'} \lambda_j \hat{c}_j$$

$$sujeito\ a: \quad \sum_{j \in J'} \lambda_j \hat{a}_j = \hat{b}$$

$$\lambda_j \in \{0, 1\}, \quad para\ j \in J'.$$

Combinando geração de colunas com *branch-and-bound*, podemos resolver o problema $[PM]$ até a otimalidade. Em cada nó t da árvore de enumeração *branch-and-bound*, o algoritmo de geração de colunas é aplicado para resolver a relaxação linear $[\overline{PM}]^t$ do problema mestre $[PM]^t$. No nó t, a cada iteração do procedimento de geração de colunas, o problema mestre linear restrito é resolvido, fornecendo assim um *upper bound* $\bar{z}^t_{PM}$ no valor da relaxação linear. E, assim, esse processo continua, até que a otimalidade do

problema linear seja provada. Nesse ponto, teremos $\bar{z}^t_{PM} = \bar{z}_{PM}$, onde $\bar{z}_{PM}$ é o valor ótimo da relaxação linear de $[PM]$. Em seguida, cada nó t é podado por *bound*, pela otimalidade do problema inteiro original, por inviabilidade ou pela solução ótima da relaxação linear ser fracionária. Neste último caso, ramificamos (*branching*) a árvore de busca. Ou seja: separamos o conjunto de soluções viáveis do problema linear no nó t em subconjuntos mais restritos e cuja união não contenha a solução fracionária corrente, mas contenha todas as soluções inteiras.

Nas próximas seções, trataremos mais detalhadamente da resolução dos problemas mestre e dos subproblemas *pricing* correspondentes.

3.4.1 Resolução do subproblema *pricing*

Devido à decomposição do problema, a solução do problema inteiro original é obtida através da resolução de vários subproblemas *pricing*. Em um método de geração de colunas para programação inteira, muitas vezes, até os subproblemas são problemas de programação inteira difíceis. Se a decomposição tiver levado a um bom desempenho do algoritmo, então é porque a dificuldade de resolvermos o problema original foi substituída pela resolução de problemas menos difíceis.

Observemos que os algoritmos de geração de colunas não exigem, sempre, a resolução do subproblema *pricing* até a otimalidade. Em um problema de minimização, qualquer coluna associada a um custo reduzido negativo pode ser usada para passarmos à próxima iteração do procedimento de geração de colunas. Com efeito, estudos (Forrest e Goldfarb [27], por exemplo) de critérios de seleção para o pivoteamente no simplex, mostram que se escolhermos a coluna com custo reduzido mais negativo para entrar na base nem sempre obteremos melhor desempenho na resolução do problema mestre linear. Na próxima seção, trataremos da seleção de colunas que levam a um

bom desempenho do procedimento na sua convergência para a solução do problema. Agora, nosso foco estará na forma como geraremos novas colunas válidas.

A resolução de subproblemas *pricing* para aplicações bem específicas constitui, por si só, um importante tópico de pesquisa. Para muitas aplicações, entretanto, resultados relevantes já podem ser encontrados na literatura. A resolução do subproblema *pricing* é a parte computacionalmente mais intensiva do nosso algoritmo de geração de colunas.

3.4.2 Subproblemas *pricing* inteiros

Quando a formulação compacta $[PI]$ é um problema inteiro, então o subproblema *pricing* $[SP]$ também é inteiro. Logo, resolvê-lo pode ser tão difícil quanto resolver $[PI]$ diretamente. Este fato deixa claro que a eficiência da decomposição está condicionada à facilidade com que os subproblemas são repetidamente resolvidos, em comparação com a resolução da formulação compacta original, apenas.

Quando a envoltória convexa do conjunto $\mathcal{X}$, $conv(\mathcal{X})$, for um poliedro cujos pontos extremos (vértices) são todos inteiros, então a solução da relaxação linear do subproblema será inteira. Isto é chamado *propriedade de integralidade* do subproblema. Ela depende, claramente, da formulação do subproblema.

Para subproblemas cuja formulação resulta, sempre, em uma solução ótima inteira, o *lower bound* para o valor ótimo do problema original $[PI]$ obtido diretamente da sua relaxação linear não é pior que o obtido com a resolução direta da formulação compacta $[PI]$, conforme observado por Geoffrion [30]. Por outro lado, quando a propriedade de integralidade não vale, poderemos potencialmente melhorar o *lower bound*; mas, obter soluções inteiras para o subproblema pode se tornar uma tarefa bastante difícil.

Então, quando o *gap* de integralidade for grande, poderemos preferir um subproblema sem a propriedade de integralidade, conforme sugerido por Geoffrion [30] em relação à relaxação lagrangeana. Por outro lado, a presença da propriedade de integralidade nos subproblemas e o tempo computacional ganho por algoritmos combinatoriais rápidos e de fácil implementação, podem compensar as desvantagens de *gaps* grandes.

Quando o poliedro $conv(\mathcal{X})$ é bem estudado, como ocorre com o *knapsack* da nossa implementação (Capítulo 4), e um algoritmo eficiente estiver disponível, pode ser bastante vantajosa essa abordagem de resolução pela formulação extensiva e subproblemas.

3.4.3 Resolução do problema mestre

A solução do problema mestre linear, a menos que ela seja inteira, não fornece uma solução viável para o problema inteiro original. Isto significa que devemos continuar nossa busca na árvore de *branch-and-bound*, afim de encontrarmos uma solução para o problema original. Se aplicarmos o algoritmo de *branch-and-bound* padrão para o problema mestre restrito, não teremos garantia de que alguma solução ótima (ou até mesmo viável) para o problema inteiro virá a ser encontrada tão somente a partir das colunas já existentes. No entanto, uma solução ótima do problema mestre inteiro restrito fornece uma solução heurística: a melhor solução inteira que podemos obter a partir da combinação das colunas existentes.

Observemos que, quando o algoritmo padrão de *branch-and-bound* é aplicado ao problema mestre restrito, os bounds da relaxação linear são válidos tão somente para o mestre restrito. Em um nó particular, precisaremos provar a otimalidade do problema linear afim de obter um *lower bound* válido para o problema mestre restrito corrente. Então, deveremos resolver o subproblema *pricing*, visto que, após uma ramificação (*branching*)

ter sido feita, ao menos alguma nova coluna pode melhorar o valor da função objetivo, se a inserirmos no problema mestre linear restrito. Para sabermos se uma tal coluna existe, basta calcularmos o custo reduzido a ela associado. Assim, através de geração de colunas em cada nó, chegaremos a um *lower bound* válido para o problema mestre e teremos gerado colunas adicionais que podem ser parte da solução ótima inteira do problema mestre original.

Existem várias considerações a serem levadas em conta quando trabalhamos com técnicas de geração de colunas em programação linear inteira. Uma delas é que a abordagem adotada para a ramificação (*branching*) deve ser compatível com com a geraão de colunas, visto que, aplicar a abordagem convencional para programação inteira não é eficiente e nem mesmo recomendado nesse novo contexto. Outra consideração importante é o comportamento ineficiente do algoritmo na resolução do problema linear causado pelo fenômeno usualmente denominado por *tailing-off effect*: um número muito grande de iterações até a otimalidade. Mas, será que é realmente necessário resolvermos o problema linear até a otimalidade para obtermos um *bound* no nó corrente? Trataremos dessas considerações nas duas seções que se seguem.

3.4.4 A ramificação (*branching*)

A resolução de problemas de programação linear inteira através de algum algoritmo *branch-and-bound* baseado em relaxações lineares consideram o fato de que todas as soluções fracionárias podem ser eliminadas por sucessivas separações do espaço de soluções viáveis. Um esquema de *branching* é um conjunto de regras que nos permite excluir qualquer solução fracionária dada e, ao mesmo tempo, garantirmos que o resultado da separação do conjunto viável do problema inteiro é ainda válido. Formalmente: sendo $\mathcal{X}^{t_0}$ o conjunto viável da relaxação linear $[\overline{PM}]^{t_0}$ no nó t_0 e dado $\bar{x}$ a solução fracionária corrente de $[\overline{PM}]^{t_0}$, o esquema de *branching* deverá estabelecer uma

forma de separar $\mathcal{X}^{t_0}$ em $\mathcal{X}^{t_1}$, $\mathcal{X}^{t_2}$, ..., $\mathcal{X}^{t_T}$ de tal forma que $\bar{x} \notin (\cup_{i=1}^{p} \mathcal{X}^{t_i})$ e $\mathcal{X}^{t_0} \cap \mathbb{Z}_+^{|\mathcal{Q}|} \subseteq (\cup_{i=1}^{p} \mathcal{X}^{t_i})$. Além disso, para termos a garantia de que o algoritmo de *branch-and-bound* irá terminar, precisamos estarmos certos de que, após um número finito de separações, as soluções dos problemas mestre nos nós folha da árvore sejam inteiras. Nesta tese, trataremos apenas do caso mais usual: a árvore de *branching* é binária ($p = 2$). Então, após qualquer processo de separação, o nó corrente t_i terá dois nós sucessores: t_{i+1} e t_{i+2}.

Um "bom" esquema de *branching* é aquele que, além de ter as propriedades que citamos no parágrafo anterior, leva em consideração o desempenho do algoritmo de *branch-and-bound.* Esse é um conceito qualitativo. Sabemos que o *branch-and-bound* é uma forma de enumeração (implícita) de todas as soluções inteiras do problema. Nele, por um lado, tentamos construir uma solução ótima inteira e, por outro, tentamos produzir um *lower bound* cada vez melhor e que prove a otimalidade da solução inteira corrente. Então, dentre os diferentes esquemas de *branching* que levam a um separação válida, desejamos escolher aquele que maximiza o valor do menor *lower bound* sobre todos os nós sucessores e que aumente a possibilidade de gerarmos soluções intermediárias inteiras. Na literatura, podem ser encontradas muitas heurísticas que argumentam alguns esquemas de *branching* como "bons". Por exemplo: podemos fazer uma separação que leve a um particionamento do conjunto viável (ou seja: $\mathcal{X}_{i+1}^t \cap \mathcal{X}_{i+2}^t$), o que provavelmente é melhor que um recobrimento; podemos, ainda, tentar separar o conjunto de soluções em dois outros de tamanhos aproximadamente iguais; ou também tentar eliminar tantas soluções quanto possível a cada processo de separação.

No contexto de geração de colunas, precisamos, também, levar em conta outras considerações. Primeiro, temos de garantir que o esquema de *branching* que estamos adotando seja compatível com o algoritmo de geração de colunas. Ou seja, podemos continuar usando esse mesmo algoritmo para resolver as relaxações lineares dos problemas (mais restritos) definidos nos

nos nós sucessores. Além disso, ao usarmos um "bom" esquema de *branching*, desejaremos manter tratáveis o problema mestre e o subproblema em cada nó.

Na próxima subseção, mostraremos como podemos proceder a separação no contexto da geração de colunas. Daremos ênfase à regra de *branching* apresentada por Ryan e Foster [59] para o problema de particionamento $[PP]$ da página 15. Inicialmente, comentaremos porque o esquema de *branching* convencional em programação inteira mais geral não é apropriado para nosso problema em particular.

3.5 O *branching* para o problema de particionamento

Consideremos um caso mais particular, em que o problema mestre é dado pelo seguinte problema de particionamento:

$$
\begin{aligned}
[PP] \quad \textit{minimizar} \quad & z_{PP} = \sum_{j \in J} c_j \lambda_j \\
\textit{sujeito a}: \quad & \sum_{j \in J} a_j \lambda_j = 1 \\
& \lambda_j \in \{0,1\}, \quad \textit{para todo } j \in J.
\end{aligned}
$$

onde $J = \{1,\ 2,\ ...,\ n\}$, $a_j \in \mathcal{K} = \{a_1,\ a_2,\ ...,\ a_p\}$, um conjunto finito, e uma função $f : \mathcal{K} \rightarrow \mathbb{R}$ tal que $c_j = f(a_j)$ são dados.

Denotemos por $[\overline{PP}]$ a relaxação linear do problema de particionamento $[PP]$.

3.5.1 O *branching* convencional

Quando a solução ótima do problema mestre linear $[\overline{PP}]$ associado a determinado nó da árvore de *branch-and-bound* não é inteira, na estratégia

de *branching* convencional, devemos escolher alguma variável com valor fracionário, digamos $\lambda_k \notin \{0,1\}$, para procedermos a ramificação. Os novos problemas lineares serão obtidos a partir da imposição das restrições $\lambda_k = 1$, no 1-*branch* (Ryan e Foster [59]), e $\lambda_k = 0$, no 0-*branch*.

Ryan e Foster [59] observam que o 0-*branch* tem muito pouco efeito no valor da função objetivo do problema $[PP]$, visto que quando uma variável fracionária é forçada a assumir valor 0, alguma outra variável fracionária tende a assumir o valor 1.

Além disso, impor uma restrição $\lambda_k = 1$ implica em $\lambda_j = 0$, para todo $j \in J \setminus \{k\}$; enquanto a restrição $\lambda_k = 0$ leva a $\sum_{j \in J \setminus \{k\}} \lambda_j = 1$. Ou seja, esse esquema de *branching* resulta em uma separação desbalanceada. De um lado, o espaço viável é muito restrito; enquanto do outro, apenas um caso é excluído, o que resulta em um problema não muito restrito em relação ao nó predecessor. Observemos, ainda, que a restrição adicionada ao subproblema não é trivial e ela pode destruir a estrutura original do subproblema. Isto dificultaria a geração de colunas por meio de alguma heurística ou por programação dinâmica, fazendo do método um processo exato ineficiente.

Na prática, quando eliminamos uma variável (resultando em uma inviabilidade primal, mas continuando dual viável), uma outra, geralmente numa solução muito próxima, compensa essa "perda" e o valor da função objetivo continua praticamente o mesmo. Tanto que, tipicamente, uma ou duas iterações do algoritmo dual do simplex são suficientes para recuperarmos a viabilidade primal da solução. O 1-*branch*, pelo contrário, geralmente leva a uma mudança significativa na solução e no valor da função objetivo, e muitas vezes exige muitas iterações do dual do simplex.

Assim, a estratégia de *branching* convencional, quando aplicada à resolução do problema de particionamento $[PP]$, provavelmente resultará no

crescimento desbalanceado da árvore de *branch-and-bound.*

Diante disso, uma nova estratégia foi proposta por Ryan e Foster [59] ainda em 1981 e que, atualmente, tem grande utilidade no contexto da geração de colunas em problemas de otimização combinatória. Na próxima subseção, apresentamos, então, essa estratégia.

3.5.2 A estratégia de *branching* de Ryan e Foster

Para o problema de particionamento padrão $[PP]$, Ryan e Foster [59] sugerem uma estratégia de *branching* mais apropriada. Ela está baseada no fato de que, numa solução inteira, duas linhas quaisquer são cobertas pela mesma coluna ou são cobertas por colunas distintas. A estratégia está formalizada na seguinte proposição, cuja demonstração apresentamos seguida de seu enunciado. Antes, seja A a matriz formada pelas restrições $\sum_{j \in J} a_j \lambda_j = 1$ do problema $[PP]$.

Proposição 3 (Ryan e Foster, 1981) *Se λ é uma solução fracionária do problema de particionamento $[PP]$, então existe ao menos um par de linhas, digamos r e s, da matriz de restrições A, tal que*

$$0 < \sum_{q:\ a_{rq}=a_{sq}=1} \lambda_q < 1.$$

Demonstração: Dada uma solução λ do problema $[PP]$, consideremos, arbitrariamente, uma variável fracionária $\lambda_{q'}$. Seja r uma linha da coluna $A_{q'}$ da matriz de restrições A tal que $a_{rq'} = 1$. Pela viabilidade de λ, temos que

$$\sum_{q=1}^{n} a_{rq}\lambda_q = 1.$$

Além disso, como λ é fracionário, existe ao menos uma outra coluna na base, digamos q'', tal que $0 < \lambda_{q''} < 1$ e $a_{rq''} = 1$. Segue-se, pelo fato de não existirem colunas duplicadas em A, que existe uma linha s com $a_{sq'} = 1$ ou com $a_{sq''} = 1$, mas não ambas. Logo, a partir do somatório anterior e deste

fato, temos

$$1 = \sum_{q=1}^{n} a_{rq}\lambda_q = \sum_{q:\ a_{rq}=1} \lambda_q > \sum_{q:\ a_{rq}=a_{sq}=1} \lambda_q > 0,$$

como queríamos demonstrar. ∎

Assim, após identificarmos as linhas r e s nas condições dadas pela Proposição 3, podemos descartar a solução fracionária corrente através de uma estratégia de separação válida fornecida pela adição das seguintes restrições de *branching*:

$$\sum_{q:\ a_{rq}=a_{sq}=1} \lambda_q = 1 \tag{3.1}$$

para um dos nós filhos (o da esquerda, por exemplo), no 1-*branch*, e

$$\sum_{q:\ a_{rq}=a_{sq}=1} \lambda_q = 0 \tag{3.2}$$

para o outro nó (o da direita), no 0-*branch*. Como o número de pares r e s formados a partir das m linhas da matriz A é finito, o algoritmo irá terminar com uma solução inteira.

A restrição do 1-*branch* (3.1) faz com que as linhas r e s sejam cobertas pela mesma coluna e a do 0-*branch* necessariamente por colunas distintas.

Em geral, essa estratégia de *branching* é compatível com os algoritmos de geração de colunas. Ao procedermos com o 0-*branch*, devemos eliminar toda coluna q com $a_{rq} = a_{sq} = 1$ do problema mestre linear corrente e proibir que alguma tal coluna seja gerada. Para isto, toda nova coluna gerada deverá satisfazer a restrição $a_{rq} + a_{sq} \leq 1$. Portanto, adicionaremos a restrição $\alpha_r + \alpha_s \leq 1$ à formulação do subproblema *pricing*. Por outro lado, no 1-*branch* devemos tomar, exclusivamente, colunas que cobrem as linhas r e s ao mesmo tempo; ou seja, devemos excluir todas as colunas do problema mestre que cobrem apenas uma dessas linhas. Assim, eliminamos todas às

colunas com $a_{rq} \neq a_{sq}$ no problema mestre corrente e exigimos que todas as novas colunas geradas satisfaçam $a_{rq} = a_{sq}$, ou seja, adicionamos a restrição $\alpha_r = \alpha_s$ ao subproblema *pricing*.

Com o esquema de *branching* de Ryan e Foster [59], podemos separar o conjunto viável em subconjuntos de tamanhos aproximadamente iguais e, além disso, as restrições que teremos de adicionar ao subproblema *pricing* são bastante simples. Para agruparmos os elementos (linhas) r e s numa mesma partição (coluna) acrescentamos a restrição $\alpha_r = \alpha_s$ ou simplesmente combinamos esses dois elementos. E para fazê-los aparecer necessariamente em partições distintas, acrescentamos $\alpha_r + \alpha_s \leq 1$, que é uma restrição disjuntiva. Assim, uma seqüência de *branching* induz a formação de subclasses de partições representadas por colunas que cobrem determinados subconjuntos de linhas. Conseqüentemente, como devemos esperar de um eficiente esquema de *branching*, as soluções dos problemas mestre lineares intermediários (em cada nó) tendem, cada vez mais, a serem inteiras.

No próximo capítulo, apresentaremos nossa implementação de um algoritmo de geração de colunas para o problema de particionamento em que as colunas da matriz de restrições são soluções de um dado problema da mochila, ou seja, para o problema *bin packing*.

Capítulo 4

A Implementação

Neste capítulo, nosso objetivo é, finalmente, discorrer sobre as questões mais relevantes numa implementação do algoritmo *branch-and-price*, particularmente para o problema *bin packing*, e apresentar os alguns resultados computacionais obtidos a partir da nossa implementação.

A implementação de um algoritmo que combina apropriadamente geração de colunas com *branch-and-bound* para resolver um problema de otimização combinatória, ou seja, um algoritmo *branch-and-price*, envolve várias opções de abordagens em diversos pontos, que geralmente são cruciais para o seu desempenho. Exemplos dessas possibilidades envolvem: a forma como iremos iniciar o algoritmo de geração de colunas; se resolveremos o subproblema *pricing* de forma exata ou apenas aproximadamente; a estratégia de *branching* a ser adotada. O fato é que a eficiência do algoritmo irá depender fortemente das escolhas feitas durante sua implementação.

Iniciaremos nosso propósito reescrevendo o problema que, neste trabalho, estamos interessados em resolver.

4.1 O problema

Queremos resolver o problema de *bin packing*, que apresentamos no Capítulo 1, a saber:

$$[BP]\ \textit{minimizar} \quad \sum_{k=1}^{m} y^k$$

$$\textit{sujeito a}: \quad \sum_{k=1}^{m} x_i^k = 1 \qquad \textit{para todo } i = 1, 2, ..., m,$$

$$\sum_{i=1}^{m} \ell_i x_i^k \leq L y^k \qquad \textit{para todo } k = 1, 2, ..., m,$$

$$x_i^k,\ y^k \in \{0,\ 1\} \quad \textit{para todo } i = 1, 2, ..., m,\ k = 1, 2, ..., m,$$

onde $\ell_i \in \mathbb{Z}_+$, para cada $i \in I = \{1,\ 2,\ ...,\ m\}$, e $L \in \mathbb{Z}_+$ são dados.

Nossa implementação, no entanto, é para a sua formulação extensiva, apropriada para a aplicação do algoritmo *branching-and-price*. Definamos o conjunto $\mathcal{K} = \{a \in \{0,1\}^m;\ \sum_{i=1}^{m} \ell_i a_i \leq L\} = \{a_1,\ a_2,\ ...,\ a_p\}$ e uma função $f : \mathcal{K} \to \mathbb{R}$ tal que $a_j \mapsto f(a_j) = 1$, para cada $a_j \in \mathcal{K}$.

O problema de *bin packing* pode ser escrito, então, como o seguinte problema de particionamento:

$$[PP]\ \ \textit{minimizar} \quad \sum_{j=1}^{p} \lambda_j$$

$$\textit{sujeito a}: \quad \sum_{j=1}^{p} a_j \lambda_j = 1$$

$$\lambda_j \in \{0,\ 1\} \qquad \textit{para } j \in \{1,\ 2,\ ...,\ p\}.$$

Como já apresentamos no Capítulo 1, página 18, cada $a \in \{0,1\}^m$, dado (implicitamente) por $\sum_{i=1}^{m} \ell_i a_i \leq L$, pode ser visto como uma coluna da matriz de restrições do problema $[PP]$ e em sua notação matricial. Neste sentido, uma coluna $a \in \mathcal{K}$ fará parte da formulação do problema apenas

quando ela for potencialmente indispensável para a solção ótima do problema. Como sabemos, do Capítulo 3, considerando $[PP]$ e dado um vetor $u \in \mathbb{R}^m$, a escolha de uma tal coluna é feita a partir da resolução do subproblema *pricing* $[SP]$, a saber:

$$\begin{array}{lll} [SP] & \textit{minimizar} & z_{SP} = f(a) - u^T a \\ & \textit{sujeito a:} & a \in \mathcal{K}. \end{array}$$

Observemos que, na formulação do problema de particionamento para o *bin packing*, o custo associado a cada coluna $a \in \mathcal{K}$ é constante e igual a 1, ou seja, $f(a) = 1$. O problema de particionamento com um custo constante associado a cada coluna é denominado *problema de particionamento não ponderado*. Supondo, sem perda de generalidade, que o problema $[SP]$ é limitado, teremos as seguintes relações para qualquer $a \in \mathcal{K}$:

$$\min\{f(a) - u^T a\} = \min\{1 - u^T a\} = 1 + \min\{-u^T a\} = 1 - \max\{u^T a\}.$$

Logo, o subproblema *pricing*, na nossa implementação, é o seguinte problema da mochila 0-1:

$$\begin{array}{lll} [PK] & \textit{maximizar} & z_{PK} = \displaystyle\sum_{i=1}^{m} u_i \alpha_i \\ & \textit{sujeito a:} & \displaystyle\sum_{i=1}^{m} \ell_i \alpha_i \leq L \\ & & \alpha_i \in \{0,1\}, \quad \textit{para } i = 1,\ 2,\ \ldots,\ m. \end{array}$$

Assim, qualquer solução viável de $[PK]$, digamos $\alpha' \in \{0,1\}^m$, será uma coluna válida para o problema $[PP]$, pois $\alpha' \in \mathcal{K}$.

Dado algum $\mathcal{K}' \subseteq \mathcal{K}$, denotemos por $[PP]^t$ o problema mestre restrito inteiro definido pelas colunas em $\mathcal{K}'$ e associado a um nó t da árvore de *branching*, e por $[\overline{PP}]^t$ sua relaxação linear. Na próxima seção, descreveremos sobre a nossa abordagem na resolução do subproblema *pricing* $[PK]$, para a obtenção de novas colunas de $\mathcal{K}$ para o conjunto $\mathcal{K}'$.

4.2 A resolução do subproblema *pricing*

Para obtermos uma solução do subproblema *pricing* $[PK]$, usamos o resolvedor XPRESS-MP e não interferimos no processo de resolução do problema. Isto significa que o XPRESS-MP assume toda a parte de preprocessamento, em que o problema $[PK]$ tem sua estrutura convenientemente adapatada para uma melhor abordagem de resolução. Além disso, esse resolvedor usa heurísticas e geração de cortes durante a resolução desse problema, o que torna bastante eficiente a obtenção de uma nova coluna, cada vez que houver necessidade.

No entanto, nem sempre esse problema é resolvido até sua otimalidade, pois utilizamos os em bounds que já conhecemos. Após a resolução de $[PK]$, ou temos provado que o processo de geração de colunas está concluído, ou temos gerado uma nova coluna (com custo reduzido negativo) que será acrescentada ao problema mestre linear restrito do nó corrente.

Como sabemos, do Capítulo 3, um algoritmo que utiliza geração de colunas, como é o caso *branch-and-price*, precisa partir de um conjunto inicial de colunas válidas. É sobre a definição dessas colunas iniciais que iremos discorrer na próxima seção.

4.3 A definição das colunas iniciais

Em cada nó t da árvore de *branching*, o procedimento de geração de colunas precisa partir de uma base viável inicial para o problema linear mestre restrito $[\overline{PP}]^t$, se existir alguma. Para isto, devemos fornecer um conjunto de colunas iniciais para o problema mestre em cada nó da árvore. Adicionalmente, ao disponibilizarmos uma base viável, teremos também: um valor inicial $\bar{z}^t_{PP}$ para a função objetivo do problema $[\overline{PP}]^t$; e uma estimativa superior (*upper bound*) para o problema inteiro original $[PP]$, se a solução

for inteira.

Uma solução viável inicial para o problema mestre linear pode ser obtida através, por exemplo, da resolução do problema Fase 1, uma abordagem já clássica no método simplex. Ou seja, adicionando apropriadamente, à formulação do problema, variáveis artificiais associadas a uma função objetivo que penaliza a presença dessas variáveis artificiais na solução. A abordagem que estamos usando é a combinação da Fase 1 com a Fase 2, acrescentando a função objetivo artificial à função objetivo original do problema mestre.

No nó raiz da árvore de *branch-and-bound*, o problema linear mestre restrito corrente pode ser inviável. Isto ocorre porque o problema inteiro original é inviável ou simplesmente porque as colunas conhecidas *a priori* ainda não são suficientes para formarem uma base viável. Nos nós sucessores, a inviabilidade dos problemas mestre lineares restritos pode ser causada por restrições de *branching* ou também pela falta de colunas relevantes. Como apresentamos anteriormente, na regra de Ryan e Foster, as restrições de *branching* são impostas através da remoção explícita de colunas na formulação do problema mestre do nó antecessor. Então, inviabilidades devido ao *branching* podem ocorrer quando as colunas disponíveis não formam uma base viável.

Em qualquer caso, com o acréscimo das variáveis artificiais, o problema mestre linear restrito sempre terá alguma solução básica viável. Se o problema mestre restrito é inviável, então ao menos alguma variável artificial permanecerá na solução do problema mestre linear restrito e, neste caso, iremos podar o nó por inviabilidade.

Na implementação, o que estamos fazendo é adicionar variáveis artificiais no momento do processamento de um nó apenas quando as colunas já disponíveis não forem suficientes para formarem uma base viável. Na prática, observamos que raramente precisamos adicionar variáveis artificiais à formulação.

Para evitarmos inviabilidades triviais causadas pelas restrições de *branching*, antes, nos certificamos de que elas não são inconsistentes. Consideremos, por exemplo, que certas restrições de *branching* impõem que as linhas i e j sejam cobertas pela mesma coluna, que as linhas j e k também sejam cobertas pela mesma coluna, mas que as linhas i e k sejam cobertas por colunas distintas. Temos, neste caso, uma inconsistência e, portanto, o problema é inviável. Outro caso possível, em que ocorre inconsistência, é quando i e j são forçadas a serem cobertas pela mesma coluna, mas a soma de seus pesos ultrapassar o peso máximo permitido. Se uma inviabilidade é detectada, então o nó é podado.

Assim, no nó raiz, as colunas iniciais que escolhemos para o problema mestre restrito formam a matriz identidade, a base canônica. Nos nós sucessores, definimos como o conjunto inicial de colunas para o problema mestre restrito, todas aquelas que foram previamente geradas (nos nós antecessores), mas que não violam as restrições de *branching* no nó corrente. Se ainda for necessário, acrescentamos colunas elementares, garantindo que existe ao menos uma base viável.

Vejamos, agora, como estamos escolhendo uma nova coluna para o problema mestre restrito.

4.4 A escolha de uma nova coluna

Uma questão natural que fazemos quando pensamos numa estratégia de seleção de colunas é: o que é uma "boa" coluna? Em outras palavras: gostaríamos de poder reconhecer as colunas válidas que são melhores para minimizarmos o valor da função objetivo. Relembremos que nosso interesse é encontrar uma solução viável primal para o problema mestre inteiro e, ao mesmo tempo, obtermos um *bound* melhor, indicando que essa é uma boa solução. Então, quando dizemos que temos uma "boa" coluna, significa que

essa coluna é necessária para a construção de uma solução inteira ou que ela nos leva mais rapidamente à solução da relaxação linear do problema mestre restrito.

Uma estratégia de seleção de colunas pode ser direcionada na geração de boas colunas ou, também, na seleção das melhores colunas dentre várias que tenham sido geradas em um dada iteração.

Nas próximas duas subseções, discorreremos sobre a estratégia de escolha de uma nova coluna a ser utilizada na implementação de um algoritmo *branch-and-price.*

4.4.1 Novas colunas para o problema mestre linear

Em uma dada iteração do algoritmo de geração de colunas para resolver o problema mestre linear $[\overline{PP}]^t$, em um nó t, a coluna associada ao custo reduzido mais negativo representa o corte mais violado na formulação dual do problema. A motivação para selecionarmos a coluna com custo reduzido mais negativo é que ela pode nos levar a uma maior redução no valor da função objetivo. No entanto, não temos garantia de que isto sempre irá ocorrer. Pode acontecer, por exemplo, que o valor corrente da função objetivo seja o ótimo e que, devido a uma presença de degeneração, algumas colunas ainda estejam associadas a custos reduzidos negativos e, assim, acrescentar tais colunas à formulação não irá melhorar o valor da função objetivo.

Uma alternativa, então, é o critério de "corte profundo", que está baseado na idéia geométrica. Nesse critério, o progresso alcançado ao acrescentarmos uma nova coluna à formulação não é medido em termos da melhora do valor da função objetivo, mas em termos das restrições do conjunto (poliedral) viável do dual do problema mestre linear. Agora, uma boa coluna é aquela que exclui uma parte maior desse conjunto poliedral. Um "corte profundo", que é aquele que tem a maior distância euclidiana até a solução

dual corrente, potencialmente leva a um espaço dual viável mais restrito. A distância euclidiana entre um corte e a solução dual corrente $u \in \mathbb{R}^m$ é a distância entre este último ponto e sua projeção no hiperplano definido pelo corte.

Para entendermos o que significa adicionar colunas ao problema mestre com o sentido de acelerarmos a convergência da otimização do problema linear, podemos consultar Forrest e Goldfarb [27] e Balas *et al.* [4], por exemplo. A primeira referência apresenta um estudo da seleção do pivô no método simplex, comparando a escolha do melhor custo reduzido com o critério de "corte profundo". Por outro lado, Balas *et al.* [4] apresenta um estudo do critério de seleção de um corte baseado na maior violação em comparação com uma abordagem de "corte profundo". Esses dois estudos são equivalentes, no sentido de que adicionar colunas no problema primal corresponde a adicionar cortes na sua formulação dual.

Na nossa implementação, optamos tão somente por escolher a nova coluna baseados no custo reduzido a ela associado. Isto se deve ao fato de que os subproblemas que tratamos são do tipo mochila 0-1 e, portanto, são resolvidos eficientemente pelo XPRESS-MP. Adicionalmente, existem desvantagens quando selecionamos as novas colunas com a abordagem de "corte profundo": a geração de colunas que maximizem a distância entre a solução dual corrente e sua projeção em alguma face do conjunto viável tem um custo computacional muito alto; essa medida se torna totalmente irrelevante quando o ponto de projeção não é dual viável; além disso, essa distância é bastante dependente da solução dual corrente para o problema mestre restrito (e geralmente temos várias alternativas para essa solução dual).

E, finalmente, como parte das estratégias de geração de colunas, precisamos decidir entre resolver, ou não, o subproblema *pricing* até a otimalidade, e gerar múltiplas colunas ou apenas uma a cada iteração do algoritmo.

Na nossa implementação, optamos por resolver cada subproblema até a sua otimalidade e gerarmos apenas uma coluna a cada resolução do problema restrito. O motivo é que a estratégia de resolução do subproblema até sua otimalidade e da geração de uma única coluna a cada iteração é geralmente superior para os casos em que podemos obter uma solução ótima para o subproblema de forma eficiente e rápida, conforme verificado por Degraeve [21] e Vanderbeck [63]), por exemplo. E este é o caso do problema da nossa implementação. A resolução dos subproblemas apenas aproximadamente e a geração múltiplas colunas a cada iteração é mais indicado para os casos em que os subproblemas exigem um esforço computacional muito grande.

Podemos usar, também, diferentes estratégias para selecionarmos colunas que melhorem nossas chances de encontrar boas soluções para o problema mestre inteiro. Este é o assunto da próxima seção.

4.4.2 Novas colunas para o problema mestre inteiro

Conforme considerações e resultados apresentados por Vanderbeck [63], colunas que são "boas" para nos fornecer uma solução inteira, geralmente não nos ajudam muito na obtenção de uma solução para o problema mestre linear. Quando implementamos um algoritmo de geração de colunas, procuramos lançar mão das melhores estratégias para obtermos as colunas mais apropriadas para a resolução do problema mestre linear restrito. Assim, ao gerarmos colunas extras que supostamente são "boas" para a otimização do problema mestre inteiro, perturbamos as estratégias para o problema linear que já estamos usando no algoritmo, visto que isto influecia nos valores das variáveis duais. Outra desvantagem, nessa abordagem, é que aumentamos desnecessariamente o tamanho do problema mestre linear restrito. Além disso, o desempenho da nossa implementação como um todo depende principalmente da sua eficiência na resolução do problema mestre linear, pois isto requer a resolução de vários subproblemas inteiros (difíceis) para gerarmos

as novas colunas. Uma solção inteira nos ajuda apenas fornecendo uma estimativa superior (*upper bound*) para o problema inteiro, que servirá para podarmos a árvore de *branching* em certos nós.

Comentaremos, agora, a implementação da regra de *branching*.

4.5 A operação de *branching*

Se a solução encontrada para o problema linear mestre não for inteira, devemos efetuar o *branching*. No nosso caso, em vez da estratégia de *branching* convencional (página 51), implementamos a regra de Ryan e Foster [59] (página 53).

Sejam r e s os índices de linhas definidos de acordo com a Proposição 3 (página 53). A cada ramificação da árvore (binária) de *branching*, estamos considerando que o ramo (nó) à esqueda corresponde ao problema no qual as colunas contêm (cobrem) as linhas r e s e que à esquerda está associado ao problema no qual as colunas não cobrem, ao mesmo tempo, as linhas r e s. Ou seja, a ramificação à esquerda, o 1-*branch*, corresponde ao caso em que r e s aparecem ambas na mesma coluna ou ambas estão fora; e a ramificação à direita, o 0-*branch*, corresponde ao caso em que apenas r, ou apenas s aparece numa coluna, ou ambas as linhas r e s estão ausentes numa determinada coluna.

Consideremos uma iteração arbitrária do algoritmo *branch-and-price*. A Proposição 3 nos garante tão somente a existência das referidas linhas r e s, sempre que alguma variável na solução corrente do problema mestre restrito linear não for inteira. Na implementação, então, precisamos fazer uma busca para identificarmos um tal par de linhas r e s. Nossa estratégia está baseada no fato de que variáveis associadas a colunas com linhas distintas, como em $[1\ 0]^T$, são mais propícias a assumirem valores inteiros que aquelas associadas

a colunas com linhas idênticas, como em $[1\ 1]^T$. Assim, tentaremos evitar que variáveis inicialmente fracionárias continuem associadas a colunas com linhas muito dependentes uma da outra.

Para isto, nossa abordagem consiste em, inicialmente, tomarmos r como sendo a linha presente no maior número de colunas, digamos $\mathcal{K}' \subset \mathcal{K}$, associadas a variáveis fracionárias; e, em seguida, ao contrário, tomarmos s como sendo a linha presente no menor número de colunas em $\mathcal{K}'$. Isto não garante que o número de colunas associadas a variáveis fracionárias e contendo ambas as linhas r e s sejam o menor possível (como gostaríamos), mas provavelmente resultará numa situação ao menos próxima da desejada.

Suponhamos que a matriz de restrições A do problema mestre restrito associado a determinado nó da árvore de *branching* tenha p linhas e q colunas, e que a solução corrente para a relaxação linear seja ótima. Inicialmente, o XPRESS-MP nos fornece o vetor de soluções λ, que tem q componentes. Em seguida, identificamos cada componente fracionária em λ. Para cada $j \in \{1,\ 2,\ ...,\ q\}$, se λ_j não é inteiro, tomamos a coluna j de A e, para cada uma de suas p linhas, incrementamos um contador que, no final da busca por variáveis fracionárias, nos fornecerá a quantidade de colunas associadas a essas variáveis. Logo, a complexidade dessa operação é $O(pq)$. E, finalmente, com a informação dado pelo contador associado a cada uma das p linhas, obtemos as linhas r e s da estratégia de *branching* de Ryan e Foster. Portanto, na nossa implementação dessa estratégia, a busca por r e s tem complexidade $O(pq) + O(p)$, ou simplesmente $O(pq)$.

Na prática, devido a esparsidade da matriz de restrições e devido à estrutura de dados que adotamos, descrita mais adiante, na seção 4.7, a escolha das linhas r e s ocorre em um tempo computacional bastante razoável. Além disso, lembremos que essa escolha é feita no máximo uma vez para cada nó da árvore de *branching* e o gargalo no algoritmo *branch-and-price* continua

sendo a resolução do problema mestre linear associado a cada nó.

Uma vez que temos determinado as linhas r e s, no ramo direito da árvore, resultante de um 0-*branch*, resolvemos o seguinte problema da mochila 0-1:

$$[PK]' \quad maximizar \quad z_{PK} = \sum_{i=1}^{m} u_i \alpha_i$$

$$sujeito\ a: \quad \sum_{i=1}^{m} \ell_i \alpha_i \leq L$$

$$\alpha_r + \alpha_s \leq 1,$$

$$\alpha_i \in \{0,1\}, \quad para\ j = 1,\ 2,\ ...,\ m.$$

E, no ramo esquerdo, resolvemos o seguinte problema resultante do 1-*branch*:

$$[PK]'' \quad maximizar \quad z_{PK} = \sum_{i=1}^{m} u_i \alpha_i$$

$$\text{sujeito a:} \quad \sum_{i=1}^{m} \ell_i \alpha_i \leq L$$

$$\alpha_r = \alpha_s,$$

$$\alpha_i \in \{0,1\}, \quad para\ j = 1,\ 2,\ ...,\ m.$$

Na prática, em vez de simplesmente acrescentarmos a restrição $\alpha_r = \alpha_s$ em $[PK]''$, o que fazemos é eliminar a linha s do problema mestre restrito e, no subproblema *pricing*, alterar o peso ℓ_r para $\ell_r + \ell_s$ e eliminar a coluna s de suas restrições. O que fazemos no problema mestre restrito é equivalente a combinarmos as linhas r e s de suas restrições. Portanto, o problema mestre restrito tem uma linha a menos e o subproblema *pricing* uma coluna a menos a cada operação 1-*branch* realizada.

Observemos, então, que esta estratégia de *branching* não irá modificar a estrutura dos subproblemas; eles continuarão sendo problemas da mochila 0-1, eficientemente solucionáveis.

Observemos, ainda, que associado a qualquer nó da árvore de *branching*, o subproblema *pricing* correspondente será formado por uma combinação de restrições idênticas à última linha de cada problema acima. Ou seja, devemos levar em conta todas as operações de *branching* realizadas do nó raiz até o nó corrente. Na nossa implementação, essa combinação dos problemas $[PK]'$ e $[PK]''$ é resolvida exclusivamente pelo XPRESS-MP. A partir da solução obtida, acrescentamos uma nova coluna ao problema mestre restrito, ou nos certificamos de que a solução corrente é ótima.

Na próxima seção, apresentaremos uma visão geral do algoritmo *branch-and-price* para o problema de particionamento com colunas do tipo "mochina".

4.6 Uma visão geral do algoritmo

Aqui, descreveremos suscintamente sobre os principais pontos da nossa implementação: o início, a seleção de um nó, o processamento de um nó, a melhora e atualização de *bounds*.

4.6.1 O início

O programa inicia sua execução lendo os dados do problema, que podem ser gerados randomicamente a partir de certos parâmetros (número de itens, peso mínimo e peso máximo de um intem, capacidade máxima de um empacotamento) ou serem lidos a partir de algum arquivo de dados. Em seguida, gera as colunas iniciais para o problema mestre restrito $[\overline{PP}]^0$; define a árvore de *branching* com o problema $[\overline{PP}]^0$ associado ao nó raiz; define o subproblema *pricing* e inicia o resolvedor XPRESS-MP. Adicionalmente, o nó raiz é atribuído à lista de nós ativos.

4.6.2 A seleção de um nó

Como já sabemos, cada iteração do algoritmo *branch-and-price* consiste na seleção de um nó da árvore a ser processado, resolvendo a relaxação linear do problema mestre restrito correspondente por geração de colunas, afim de obter um *lower bound* para esse nó e, ao mesmo tempo, tentar melhorar a solução inteira corrente. A ordem em que os nós da árvore de *branching* são processados tem uma significante influência sobre o desempenho geral do algoritmo.

Implementamos a combinação de dois esquemas de seleção de nós: busca em profundidade (*depth-first search*) e busca do melhor limitante (*best-first search*). Na busca em profundidade, o nó selecionado é sempre algum descendente do nó processado na interação imediatamente anterior. Como é mais provável que encontremos soluções inteiras mais abaixo na árvore de *branching*, quando várias restrições de *branching* já foram adicionadas, aplicamos a busca em profundidade apenas em um primeiro momento, até obtermos uma solução inteira. Esta solução, geralmente, irá nos ajudar a podar a ávore em determinados nós, visto que estaremos interessados em processar um número menor quanto possível deles. De fato, em todos os casos o nó com o melhor limitante inferior (*lower bound*) precisa ser processado e, portanto, procuraremos selecioná-lo o mais cêdo possível. Por outro lado, nós que tiverem um limitante inferior muito alto podem não ter de serem completamente processados se já conhecermos um limitante inferior melhor que nos leve a podá-los. Além disso, o nó com melhor limitante inferior é também o mais promissor a nos fornecer uma solução inteira melhor.

As estratégias de seleção de um nó e de seleção de uma variável de *branching* é que determinam a seqüência de todo o processo de resolução do problema inteiro. O uso de determinada estratégia pode ser melhor para determinada instância do problema e pior para outra. Portanto, não podemos

fazer uma conclusão sobre o desempenho de uma em relação à outra.

4.6.3 O processamento de um nó

Uma vez que temos selecionado determinado nó, tomamos as restrições de *branching*; detectamos certas inconsistências óbvias; geramos o conjunto inicial de colunas para esse nó específico; e definimos a formulação do problema e do subproblema a ele associado. Em seguida, procedemos a resolução do problema mestre linear.

Em cada iteração do algoritmo de geração de colunas, resolvemos o problema mestre restrito linear. Em seguida, verificamos se a solução é inteira e, caso seja, atualizamos convenientemente o valor do limitante inferior. Então, tomamos os valores da solução dual e resolvemos o subproblema *pricing*. Assim, temos nos certificado de que já obtemos uma solução ótima ou temos gerado uma nova coluna para o problema mestre.

Se o nó corrente não pode ser eliminado a partir de argumentos óbvios, então procedemos a estratégia de *branching* de Ryan e Foster, descrita na seção anterior. Assim, definimos dois nós sucessores, cujos limitantes são, inicialmente, iguais aos do nó antecessor, e os incluímos na lista de nós ativos.

4.6.4 A obtenção de soluções inteiras

O algoritmo de geração de colunas é um método de aproximação primal, no sentido em que as soluções sucessivas dos problemas mestre lineares são primais viáveis. São elas que fornecem limites superiores (*upper bounds*) na solução ótima do problema mestre linear e, tamém, na solução ótima do problema inteiro, se existirem soluções inteiras para os problemas intermediários. Em geral, é muito difícil encontrar uma solução inteira ótima, ou seja, eliminar o *gap* entre o *lower bound* em cada nó e o valor da me-

lhor solução inteira. O esforço computacional envolvido na sua construção é muito grande.

4.6.5 O critério de parada

Depois que um nó tiver sido processado, se uma nova solução inteira foi encontrada, verificamos, na lista de nós ativos, se certos nós podem ser eliminados por *bound*. O algoritmo termina quando essa lista de nós for vazia. Neste caso, exibimos os resultados obtidos: a melhor solução inteira encontrada e, também, estatísticas sobre número de interações e números de colunas geradas, por exemplo.

Antes de apresentarmos alguns resultados computacionais obtidos, faremos um comentário sobre a estrutura de dados que adotamos na implementação do *branch-and-price* para o problema de *bin packing*.

4.7 A estrutura de dados

A matriz de restrições de um problema de particionamento, em geral, é bastante esparsa, ou seja, poucos elementos, em relação ao total, são valores diferentes de zero. No caso do problema da nossa aplicação do *branch-and-price*, a matriz de restrições do problema mestre restrito é binária. Logo, em todas as operações com esta matriz, como busca e adição de elementos, precisamos realmente conhecer tão somente a localizações dos valores iguais a 1.

Consideremos uma matriz A formada por elementos $a_{ij} \in \{0, 1\}$, para cada $i = 1, 2, ..., p$ e $j = 1, 2, ..., p$, e seja N o número de elementos iguais a 1 (os outros $pq - N$ elementos são iguais a zero). Para armazenarmos essa matriz, na nossa implementação, estamos utilizando três vetores: o primeiro, digamos `ind[]`, de tamanho N, é formado pelos índices das linhas não-nulas

da primeira coluna, seguido dos índices das linhas não-nulas da segunda coluna, e assim sucessivamente, até o índice da última linha não nula na última coluna da matriz A; o segundo vetor, digamos `qnz[]`, de tamanho p, é formado pela quantidade de elementos não-nulos, sucessivamente, em cada coluna de A; e, finalmente, o terceiro vetor, digamos `inic[]`, também de tamanho p, sucessivamente formado pela posição em `ind[]` associada ao primeiro elemento não-nulo de cada coluna da matriz A. Esta abordagem é similar à que é usada pelo Xpress Optimizer [14].

Finalmente, apresentaremos alguns resultados computacionais na próxima seção.

4.8 Os resultados computacionais

Todos as execuções foram feitas em um computador com processador `AMD Athlon XP 1800`, `1.53GHz` e com `1GB` de memória RAM. Todo o código da nossa implementação foi desenvolvido em `C++`, com chamadas à biblioteca `Optimizer` do XPRESS-MP tão somente para a resolução dos problemas mestre lineares e dos subproblemas *pricing* (*knapsack 0-1*). Para a compilação, utilizamos o `GNU gcc`, versão 3.2, para Linux.

Inicialmente, apresentaremos os pontos mais relevantes no decorrer da execução do programa obtido a partir da nossa implementação do *branch-and-price* para o problema de particionamento com colunas *knapsack*, ou simplesmente *bin packing*.

Consideremos o problema de *bin packing* [BP], da página 57. Como dissemos anteriormente, uma das opções para a entrada dos dados do problema no início da nossa implementação é que eles sejam gerados aleatoriamente a partir de quatro parâmetros fornecidos diretamente pelo usuário: a quantidade de itens m, o menor e o maior valor para ℓ_i, e a capacidade L.

Instâncias	Itens m	Capacidade L	Pesos dos itens			Valor ótimo inteiro z^*_{BP}
			min	*med*	*max*	
bp01	5	120	24	50,00	80	3
bp02	10	120	2	48,90	80	5
bp03	15	120	2	46,67	96	6
bp04	20	120	2	46,47	96	8
bp05	25	120	2	47,58	99	10
bp06	30	120	2	47,07	99	12
bp07	32	120	2	45,75	99	13
bp08	34	120	2	44,79	99	13
bp09	36	120	2	47,08	99	15
bp10	38	120	2	47,95	99	16
bp11	40	120	2	47,00	99	16
bp12	44	120	2	48,32	99	18
bp13	46	120	2	49,59	99	20
bp14	47	120	2	49,81	99	20
bp15	48	120	2	49,21	99	20
bp16	50	120	2	49,95	99	21
bp17	52	120	2	48,77	99	22
bp18	55	120	2	48,76	99	23
bp19	57	120	2	49,23	99	24
bp20	60	120	2	48,38	99	25

Tabela 4.1: Instâncias dos problemas teste. *As instâncias do problema de* bin packing *são geradas aleatoriamente a partir de quatro parâmetros: a quantidade de itens (*m*), a capacidade (*L*), o menor peso (*min*) e o maior peso (*max*). Adicionalmente, conhecemos a média dos pesos (*med*) e o valor ótimo inteiro para cada instância.*

A título de ilustração, no Anexo A, apresentamos as saídas fornecidas (em Inglês) durante a execução da nossa implementação para uma instância aleatória com apenas 10 itens no *bin packing*. Apesar de ser um problema tão modesto, é possível perceber, inclusive, a ocorrência do *tailing off effect*, em que soluções ótimas consecutivas iguais são obtidas para a relaxação linear de problemas mestre restritos consecutivos num mesmo nó da árvore de *branching*.

Até o final da resolução de um problema, o programa exibe várias informações sobre as operações realizadas, dentre elas: o número de linhas e o número de colunas na formulação compacta do *bin packing*; o número

de linhas no problema de particionamento da sua formulação extensiva; o número de itens considerados; o menor, o médio e o maior peso associados ao subproblema *pricing* e sua capacidade; o valor da solução ótima inteira; o total de colunas geradas, incluindo as da base inicial; o número máximo de colunas geradas; o número máximo de colunas em algum problema restrito; o número de nós na árvore de *branching* gerados durante a resolução do problema e daqueles efetivamente processados; o número de nós podados por inviabilidade, por otimalidade e por *bound*; o número de nós ativos, que ainda não foram conferidos.

Instâncias	Formulações				
	compacta $[BP]$			extensiva $[PP]$	
	$\bar{z}^*_{BP}$	linhas	colunas	$\bar{z}^*_{PP}$	linhas
bp01	2,08	10	30	3,00	5
bp02	4,07	20	110	5,00	10
bp03	5,83	30	240	6,00	15
bp04	7,36	40	420	8,00	20
bp05	9,51	50	650	10,00	25
bp06	11,76	60	930	12,00	30
bp07	12,20	64	1.056	12,96	32
bp08	12,69	68	1.190	12,99	34
bp09	14,12	72	1.332	14,95	36
bp10	15,18	76	1.482	15,99	38
bp11	15,66	80	1.640	16,00	40
bp12	17,71	88	1.980	18,00	44
bp13	19,01	92	2.162	19,96	46
bp14	19,50	94	2.256	20,00	47
bp15	19,68	96	2.352	20,00	48
bp16	20,47	100	2.550	20,99	50
bp17	21,13	104	2.756	21,94	52
bp18	22,35	110	3.080	22,94	55
bp19	23,38	114	3.306	24,00	57
bp20	24,19	120	3.660	24,95	60

Tabela 4.2: As duas formulações para o *bin packing*. *As formulações compacta, $[BP]$, e extensiva, $[PP]$, se referem ao mesmo problema; no entanto, elas se diferem no número de linhas e de colunas e, o mais importante: a relaxação linear de $[PP]$ fornece* bounds *melhores para o valor ótimo inteiro do problema.*

Os resultados que apresentaremos agora, referem-se a problemas teste similares a uma das classes usadas, em 1997, por Scholl *et al.* [60] e também

por Alvim *et al.* [1], em 2004. Scholl *et al.* propõem um método exato (BISON) que faz uso de vários *bounds*, procedimentos de redução, heurísticas, e procedimentos de *branch-and-bound* com um novo esquema de *branching*. O trabalho de Alvim *et al.* apresenta uma heurística que consiste em um procedimento híbrido com o uso de estratégias de obtenção de *lower bounds*, geração de soluções iniciais a partir do dual do *bin packing* e outras técnicas. Nosso objetivo, aqui, não é fazer uma comparação com outros trabalhos, mas levantar alguns pontos importantes dos resultados obtidos para uma das classes de instâncias que implementamos.

Já sabemos que os modelos $[BP]$ e $[PP]$, da página 57 deste capítulo, se referem a formulações distintas para o mesmo problema: o *bin packing*. Assim, podemos resolvê-lo a partir de uma ou outra formulação.

Os dados das instâncias que apresentamos nesta seção são os da Tabela 4.1. Inicialmente, fornecemos a quantidade de itens, m, a capacidade máxima do *packing*, L, o menor peso, min, e o maior peso, max, que, em seguida são gerados para formar uma instância do problema. Na tabela, apresentamos, ainda, a média aritmética med dos m pesos gerados pelo gerador `rand()` do e o valor ótimo inteiro para cada instância

Na Tabela 4.2 apresentamos alguns dados sobre as formulações compacta, $[BP]$, e extensiva, $[PP]$, do *bin packing* dados pelas instâncias representadas na Tabela 4.1. Observemos que, se o número de itens do problema é m, então sua formulação compacta tem $2m$ linhas e $m^2 + m$ colunas. Mais adiante, na Tabela 4.3, observaremos que para a resolução do problema pelo *branch-and-pricing*, em todos os casos testados, o número de colunas geradas foi bastante menhor que o presente na formulação compacta. Além disso, o número de linhas sempre cai para a metade na formulação extensiva.

Observamos, ainda, na Tabela 4.2, que a o valor ótimo $\bar{z}^*_{PP}$ da relaxação linear do problema reformulado $[PP]$ é um *bound* nunca pior para o

Instâncias	*Branch-and-price*			XPRESS-MP		
	Iterações do simplex	Nós	Colunas geradas	Iterações do simplex	Iterações total	Nós
bp01	1	1	6	12	173	1
bp02	32	1	23	35	130	1
bp03	174	6	59	54	2.973	262
bp04	206	7	65	96	2.239	73
bp05	358	7	87	132	3.810	309
bp06	652	5	110	219	9.469	1.223
bp07	1.713	51	291	214	2.919	117
bp08	1.741	34	239	217	3.904	245
bp09	1.655	26	205	236	3.130	129
bp10	1.928	34	247	281	3.823	157
bp11	1.172	1	135	342	6.004	413
bp12	1.763	16	200	332	127.583	15.878
bp13	2.789	58	315	257	6.464	401
bp14	2.263	19	225	335	7.357	604
bp15	2.888	32	277	352	33.778	3.855
bp16	2.755	25	261	285	6.896	2.154
bp17	3.011	38	278	451	5.474	268
bp18	3.160	20	265	273	7.261	450
bp19	2.272	7	221	301	13.891	1.126
bp20	3.561	16	265	283	7.802	262

Tabela 4.3: Resultados computacionais da implementação do *branch-and-price* e do XPRESS-MP para o *bin packing. Algumas estatísticas sobre a nossa implementação e sobre o XPRESS-MP aplicados às instâncias da Tabela 4.1: o total de iterações do simplex, o número de nós processados na árvore de* branching*, o número de colunas geradas até a otimalidade do problema e, no caso do XPRESS-MP, o número total de iterações (incluindo heurísticas e cortes adicionados à matriz de restrições).*

valor ótimo inteiro, se comparado com o ótimo $\bar{z}^*_{BP}$ da relaxação linear do problema original $[BP]$.

E, finalmente, na Tabela 4.3, apresentamos alguns resultados obtidos com a aplicação pura do resolvedor XPRESS-MP para as instâncias da Tabela 4.1 e também da nossa implementação do *branch-and-price*. Todos os problemas foram resolvidos até a otimalidade, tanto pelo XPRESS-MP, quanto pela nossa implementação.

A segunda coluna da Tabela 4.3, mostra o número total de iterações

do simplex em toda a execução do *branch-and-price*. Ou seja, esse é o número de iterações que o simplex, já presente no resolver XPRESS-MP, levou para resolver todas as relaxações lineares dos problemas mestre associados a cada nó da árvore de *branching*. Os dados dessa coluna podem ser comparados com os da quinta coluna, que traz a mesma informação para o caso do uso tão somente do XPRESS-MP. Observemos que esse número de iterações é bem menor que a grande parte dos correspondentes nos resultados da nossa implementação.

A terceira coluna apresenta o número de nós da árvore de *branching* processados no *branch-and-price*. Ele é, em geral, bem menor que o correspondente na resolução pelo XPRESS-MP, dado na última coluna da Tabela 4.3. Este resultado, possivelmente, sugere que a regra de *branching* proposta por Ryan e Foster [59] para o problema de particionamento com dados binários torna menor a árvore de *branching*, ao eliminar várias colunas em uma única vez.

Como sugerimos anteriormente, o número de colunas geradas pelo *branch-and-price* pode ser comparado com a quantidade de colunas presente (explicitamente) na formulação compacta do *bin packing* que, em todos os casos que temos testado, é consideravelmente maior, conforme a Tabela 4.2.

O total de iterações, fornecido pelo XPRESS-MP, tem se mostrado bastante maior para a maioria dos testes que temos realizado, como podemos observar pelos dados da penúltima coluna da Tabela 4.3. Devemos levar em conta que, durante a resolução do problema, o XPRESS-MP lança mão de várias técnicas, dentre elas: abordagens eficientes de preprocessamento, heurísticas e métodos de geração de cortes.

Todas as instâncias da Tabela 4.1 foram resolvidas em um tempo computacional bastante pequeno, tanto pelo XPRESS-MP, quanto pelo *branch-and-price*. Algumas vezes, esse tempo foi menor no caso do XPRESS-MP e,

em outras, ele foi melhor na execução do *branch-and-price.* O maior tempo ocorreu para o *bp*12, com 44 itens, que foi de 17 segundos para o *branch-and-price*, contra 44 segundos para o XPRESS-MP. Para a instância *bp*06, com 30 itens, também tivemos uma situação parecida: 7 segundos para a nossa implementação, contra 15 segundos para o XPRESS-MP. No entanto, houve casos em que o contrário ocorreu: 25 segundos para o *branch-and-price* e 15 segundos para o XPRESS-MP. A maior parte dos tempos computacionais ficou abaixo de 1 segundo para as duas abordagens de resolução.

A seguir, faremos as nossas considerações finais.

Considerações Finais

Este trabalho resultou na discussão básica de algoritmos de geração de colunas em problemas de otimização combinatória linear e, em particular, de otimização combinatória. De forma mais específica, estivemos interessados na resolução do problema *bin-packing* e na implementação de um algoritmo *branch-and-price* com a aplicação da regra de *branching* de Ryan e Foster para problemas de particionamento 0-1.

Pudemos observar que a decomposição de um problema inteiro pode resultar em uma reformulação cuja relaxação linear forneça melhores estimativas (*bounds*) para o valor ótimo inteiro. Isto nos permite resolver problemas inteiros difíceis até a otimalidade. No entanto, vimos que em geral teremos uma formulação com um número excessivamente grande de variáveis, o que requer que o problema seja resolvido através do uso de técnicas de geração de colunas, combinado com um algoritmo exato. As técnicas clássicas de programação inteira precisam, então, serem adaptadas com o objetivo de se tornarem compatíveis com o procedimento de geração de colunas utilizado.

Neste sentido, apresentamos um esquema de *branching* proposto por Ryan e Foster que é provavelmente mais vantajoso que o esquema clássico no contexto da geração de colunas. Além disso, sugerimos algumas condições para a poda de um nó na árvore de *branching* antes do problema associado ser resolvido até sua otimalidade. Nos comentários sobre a implementação, sugerimos algumas decisões a serem tomadas em determinados pontos do algoritmo que podem ter uma influência significativa no desempenho final do

algoritmo. Os resultados computacionais que obtivemos mostram as potenciais capacidades e limitações de tais decisões.

Como já observamos, o emprego de um algoritmo de geração de colunas é fortemente aconselhável para problemas em que a presença de determinadas restrições em sua formulação original dificulta em muito a sua resolução direta e, por outro lado, essas restrições podem ser tratadas por meio de um problema auxiliar que pode ser resolvido eficientemente. Particularmente, apresentamos o caso em que a reformulação do problema original, obtida a partir de algum método de decomposição, é um problema de particionamento e o subproblema *pricing* é um problema da mochila (*knapsack* 0-1). A portabilidade do método para outras aplicações é bastante natural em muitos casos, bastando que procuremos usar uma forma tão eficiente quanto possível para obtermos a solução do subproblema.

Por outro lado, o método de geração de colunas pode ser bastante ineficiente quando o subproblema for intratável, no sentido de que ele não pode ser resolvido dentro de um tempo computacional razoável. Em muitas aplicações, o subproblema é difícil e não pode ser resolvido com técnicas clássicas. Assim, é necessário desenvolvermos um algoritmo especial para cada subproblema particular. Na nossa implementação, por exemplo, tivemos que adicionar cortes ao subproblema *pricing*, um *knapsack* 0-1, à medida que nos aprofundamos nos nós da árvore de *branching*. Isto acaba exigindo mais tempo de processamento na resolução desses subproblemas.

Outra dificuldade que verificamos para um algoritmo de geração de colunas em problemas de programação linear inteira ocorre quando as soluções do problema mestre se torna intratável. No nossa aplicação, por exemplo, muitas vezes é evidente a dificuldade de se encontrar uma solução ótima para o problema mestre inteiro, visto que podem existir muitas combinações de colunas que levam a um mesmo valor ótimo para a função objetivo do problema

linear, fenômeno denominado *tailing off effect*. Ele pode ser observado até mesmo para instâncias pequenas, como é o caso que apresentamos no Anexo A.

Muitos trabalhos têm sido desenvolvidos com o objetivo de se obter algoritmos de geração de colunas cada vez mais robustos para diferentes classes de problemas de otimização combinatória. Na nossa implementação, não exploramos completamente as abordagens atualmente usadas em programação inteira. Muitas técnicas clássicas de programação inteira têm suas equivalentes no contexto da geração de colunas. Uma dessas técnicas é a fixação de variáveis de acordo com o seu custo reduzido associado; uma outra técnica clássica se refere à construção de uma soluão inteira para o problema mestre a partir do arredondamento de alguma solução do problema relaxado, que também pode ser adaptada para o contexto da geração de colunas.

As pesquisas atuais têm se concentrado também na procura por maneiras de se reduzir o *tailing off effect*, através dos métodos de *estabilização de colunas*. Geralmente, essas pesquisas estão baseadas em soluções duais que são pontos centrais da face ótima, obtidos a partir de algum método de ponto interior, em vez de soluções associadas a algum ponto extremo do conjunto de soluções ótimas.

Temos observado, ainda, que se uma boa estimativa (*bound*) para a solução ótima inteira é conhecida *a priori*, então podemos reduzir significativamente o esforço computacional na busca por uma solução inteira. Conseqüentemente, parece bastante razoável que inicialmente gastemos um certo tempo na tentativa de encontrar uma boa solução para o problema mestre inteiro. Neste sentido, boas heurísticas podem ser desenvolvidas para o problema mestre inteiro de cada aplicação em particular. Alvim *et al.* [1] propõem um procedimento híbrido para o caso particular do problema de *bin packing* que apresenta bons resultados em relação a várias outras heurísticas.

Devido à correspondência entre a adição de colunas na formulação primal e a adição de cortes na formulação dual do problema mestre linear, poderíamos pensar também em usar técnicas desenvolvidas para *branch-and-cut*, como os métodos de planos de cortes, por exemplo. No entanto, em programação não-linear, outras técnicas com melhor convergência teórica já são conhecidas e estão sendo testadas atualmente como uma alternativa aos métodos clássicos. Uma dessas técnicas é denominada *método bundle* é apresentada por Lemaréchal [43, 44], Kiwiel [39] e Hiriart-Urruty e Lemaréchal [37], por exemplo. Briant *et al.* [11], em janeiro de 2005, apresenta uma comparação entre diversos métodos de estabilização de colunas sob o ponto de vista primal e também dual, com algumas aplicações práticas, inclusive.

Portanto, a implementação de um algoritmo *branch-and-price*, juntamente com um bom resolvedor para as relaxações lineares e para os subproblemas *pricing*, pode se tornar bastante eficiente e robusta se incorporarmos métodos de estabilização de colunas e também heurísticas, que não foram tratados nos trabalhos desta tese de mestrado.

Referências Bibliográficas

[1] Alvim, A. C. F., Ribeiro, C. C., Glover, D. J., Aloise, D. J. "A hybrid improvement heuristic for the one-dimensional bin packing problem". *Journal of Heuristics*, v. 10, pp. 205-229, 2004.

[2] Balas, E., Padberg, M. W. "Set partitioning: a survey". *SIAM REVIEW*, v. 18, n. 4, pp. 710-760, 1976.

[3] Balas, E., Padberg, M. W. "Set partitioning: a survey". *Combinatorial Optimization*, John Wiley & Sons, edt., pp. 151-210, 1979.

[4] Balas, E., Ceria, S., Cornuéjols, G. "A lift-and-project cutting plane algorithm for mixed 0-1 program". *Mathematical Programming*, v. 58, pp. 295-324, 1993.

[5] Balas, E., Glover, F., Zionts, S. "An additive algorithm for solving linear programs with zero-one variables". *Operations Research*, v. 13, pp. 517-546, 1965.

[6] Barnhart, C., Johnson, E. L., Nemhauser, G. L., Savelsbergh, M. W. P., Vance, P. H. "Branch-and-price: column generation for solving huge integer programs". *Operations Research*, v. 46, n. 3, pp. 316-329, 1998.

[7] Beasley, J. E., Cao, B. "A tree search algorithm for the crew scheduling problem". *European Journal of Operations Research*, v. 94, pp. 517-526, 1996.

[8] Benders, J. F. "Partitioning procedures for solving mixed-variables programming problems". *Numerical Methods*, v. 4, pp. 239-252, 1962.

[9] Bodin, L. D. "Twenty years of routing and scheduling". *Operations Research*, v. 30 (4), pp. 571-579, 1990.

[10] Borndörfer, R. Grötschel, M., Löbel, A. "Duty scheduling in public transit". *In* W. Jäger e H. J. Krebs, ed., *Mathematics - Key Technology for the Future*, pp. 653-674, Spring-Verlag, Berlin, 2003.

[11] Briant, O., Lemaréchal, C., Meurdesoif, Michel, S., Perrot, N., Vanderbeck, F. "Comparation of bundle and classical column generation". *Rapport de recherche*, n. 5453, 2005.

[12] Cooper, M.W., Farhangian, K. "Multicriteria optimization for nonlinear integer-variable problems". *Large Scale Systems*, v. 9, pp. 73-78, 1985.

[13] Cullen, F. J., Jarvis, J., Ratliff, D. "Set partitioning based heuristics for interactive routing problem". *Networks*, v. 11, pp. 125-144, 1981.

[14] Dash Optimization Ltd. *Xpress-Optimizer Reference Manual.* Release 13, 2001.

[15] Dantzig, G. B. "Maximization of a linear function of variables subject to linear inequalities". In *Activity Analysis of Production and Allocation* (Chap. XXI), ed. T. C. Koopmans, John Wiley, New York, pp. 339-347, 1951.

[16] Dantzig, G. B. "On the significance of solving linear programming problems with some integer variables". *Econometrica*, v. 28, pp. 30-44, 1960.

[17] Dantzig, G. B. "Programming in a linear structure". *Comptroller*, USAF, Washington D. C., 1948.

[18] Dantzig, G. B. Fulkerson, D. R., Johnson, S. "Solution of a large scale traveling salesman problem". *Operations Research*, v. 2, pp. 393-410, 1954.

[19] Dantzig, G. B., Thapa, M. K. *Linear programming 1: introduction*, Springer Series in Operations Research, Springer-Verlag, New York, 1997.

[20] Dantzig, G. B., Wolfe, P. "Decomposition principle for linear programs". *Operations Research*, v. 8, pp. 101-111, 1960.

[21] Degraeve, Z. *Scheduling joint product operations with proposal generation methods.* Dissertation for the degree of Doctor in Philosophy,

Graduate School of Business, University of Chicago, Chicago, Illinois, 1996.

[22] Desrosiers, J., Soumis, F., Desrochers, M. "Routing with time windows by column generation". *Networks*, v. 14, pp. 545-565, 1984.

[23] Dudziński, K., Walukiewicz, S. "Exact methods for the knapsack problem and its generalizations". *European Journal of Operational Research*, v. 28, pp. 3-21, 1987.

[24] Easingwood, C. "A heuristic approach to selecting sales regions and territories". *Operations Research*, v. 24 (4), 1980.

[25] Fleuren, H. A. *A computational study of the set partitioning approach for vehicle routing and scheduling problems.* Dissertation, University of Twente, The Netherlands. 1988.

[26] Ford, L. R., Fulkerson, D. R. "A suggested computation for multicommodity flows". *Management Science*, v. 5, n. 1, pp. 97-101, 1958.

[27] Forrest, J. J., Goldfarb, D. "Steepest-edge simplex algorithms for linear programming". *Mathematical Programming*, v. 57, pp. 341-374, 1992.

[28] Garcia, S. M. A. *Heurística polinomial para seqüência minmax disjuntiva de matchings perfeitos em* K_{nn}. Dissertação de mestrado, Instituto Militar de Engenharia (IME), Rio de Janeiro, 1990.

[29] Garey, M., Johnson, D. S. *Computer and intractibility: a guide to the theory of NPcompleteness.* Freeman, San Francisco, 1979.

[30] Geoffrion, A. M. "Lagrangean relaxation for integer programming". *Mathematical Programming Study*, v. 6, pp. 62-88, 1974.

[31] Gilmore, P. C., Gomory, R. E. "A linear programming approach to the cutting stock problem". *Operations Research*, v. 9, pp. 849-859, 1961.

[32] Gilmore, P. C., Gomory, R. E. "A linear programming approach to the cutting stock problem, Part II". *Operations Research*, v. 11, pp. 863-888, 1963.

[33] Gilmore, P. C., Gomory, R. E. "The theory and computation of kanapsack functions". *Operations Research*, v. 14, pp. 1045-1074, 1966.

[34] Goldbarg, M. C., Luna, H. P. L. *Otimização combinatória e programação linear: modelos e algoritmos*, Campus-SP, 2000.

[35] Gomory, R. E. "An algorithm for integer solutions to linear programs". *In: Recent advances in mathematical programming*, pp. 269-302, McGraw-Hill, 1963.

[36] Gomory, R. E. "Outline of an algorithm for integer solutions to linear programs". *Bulletim of the American Mathematics Society*, v. 64, pp. 275-278, 1958.

[37] Hiriart-Urruty, J. B., Lemaréchal, C. *Convexis analysis and minimization algorithms*. Spring Verlag, Heidelberg, 1993.

[38] Hoffman, K. L., Padberg, M. "Solving airline crew scheduling problems by branch-and-cut". *Management Science*, v. 39, pp. 657-682, 1993.

[39] Kiwiel, K. C. "An aggregate subgradient method for nonsmooth convex minimization". *Mathematical Programming*, v. 27, pp. 320-341, 1983.

[40] Krieken, M. G. C., Fleuren, H. A., Peeters, M. J. P. "A lagrangean relaxation based algorithm for solving set partitioning problems". *CentER Discussion Paper*, n. 2004-44. Disponível em: `http://ssm.com/abstract=557848`. Acesso em: 22 fev. 2005.

[41] Klee, V., Minty, G. "How good is the simplex algorithm?". *Inequalities III*, O. Sisha, Academic Press, New York, 1972.

[42] Land, A. H., Doig, A. G. "An automatic method for solving dicrete programming problems". *Econometrica*, v. 28, pp. 497-520, 1960.

[43] Lemaréchal, C. "An algorithm for minimizing convex functions". *In* Rosenfeld, J. L., editor, *Information Processing*, v. 74, pp. 552-556, North Holland, 1974.

[44] Lemaréchal, c. "Nonsmooth optimization and descent methods". *Research Report 78-4*, IIASA, 1978.

[45] Lübbecke, M. E., Desrosiers, J. "Selected topics in column generation". *Technical Report 008-2004*, Technische Universität Berlin, 2004.

[46] Maculan, N. "Linear 0-1 programming". *In* Pardalos, P. M., Resende, M. G. C. *Handbook of applied combinatorial optimization*. Oxford University Press, 2002.

[47] Maculan, N. "Relaxation Langrangienne: le problème du Knapsack 0-1". *INFOR*, v. 21, pp. 315-327, 1983.

[48] Maculan, N., Fampa, M. *Otimização Linear*, Notas - PESC/COPPE/UFRJ, 2002.

[49] Maculan, N., Passini, M. M., Brito, J. A. M., Loiseau, I. "Column-generation in integer linear programming". *RAIRO Operational Research*, v. 37, pp. 67-83, 2003.

[50] Martello, S. "A new algorithm for the 0-1 kanapsack problems". *Management Science*, v. 34, pp. 633-645, 1190.

[51] Martello, S., Toth, P. *Knapsack problems: algorithms and computer implementations.* John Willey & Sons, New York, 1990.

[52] Minkowski, H. *Gesammelte Abhanlungen.* Teubener, Leipzig, 1911.

[53] Nemhauser, G. L., Wolsey, L. *Integer and Combinatorial Optimization.* Wiley & Sons, 1988.

[54] Nisijo, K. Maruoka, A. "On optimum decomposition of files". *Trans. Inst. Electron. and Commun. Eng. Jp. Sect. E.*. E62, v. 8, pp. 579-580, 1979.

[55] Ralphs, T., Galati, M. "Decomposition in integer programming". For book chapter in *Integer Programming: theory and practice*, CRC Press 2005. Disponível em: `http://www.optimization-online.org/DB_HTML/2004/12/1029.html`. Acesso em: 14 fev. 2005.

[56] Reinoso, H., Maculan, N. "Lagrangean decomposition for integer programming: a new scheme". *INFOR*, v. 52, n. 2, pp. 147-167, 1995.

[57] Revelle, C. S., Marks, D., Liebman, J. C. "An analysis and public setor location models". *Man. Sci.*, v. 16 (12), pp. 692-707, 1970.

[58] Ronen, E. "NP-Complete stable matching problems". *Journal of Alg.*, v. 11, pp. 285-304, 1990.

[59] Ryan, D. M., Foster, B. A. "An integer programming approach to scheduling". *In* A. Wren (editor), *Computer Scheduling of Public Transport Urban Passenger Vehicle and Crew Scheduling*, North-Holland, Amsterdan, pp. 269-280, 1981.

[60] Scholl, A., Klein, R, Jürgens, C. "BISON: a fast hybrid procedure for exactly solving the one dimensional bin packing problem". *Computers and Operations Research*, v. 24, pp. 627-645, 1997.

[61] Vance, P. H., Barnhart, C., Johnson, E. L., Nemhauser, G. L. "Airline crew scheduling: a new formulation and decomposition algorithm". *Operations Research*, v. 45(2), pp. 188-200, 1987.

[62] Vanderbeck, F. "Computational study of a column generation algorithm for bin packing and cutting stock problems". *Mathematical Programming*, v. 86, n. 3, pp. 565-594, 1999.

[63] Vanderbeck, F. *Decomposition and Column Generation for Integer Programs.* Thèse présentée en vue de l'obtention du Grade de Docteur en Sciences Appliquées, Université Catholique de Louvain, 1994.

[64] Vanderbeck, F. "On Dantzig-Wolfe decomposition in integer programming and ways to perform branching in a branch-and-price algorithm". *Operations Research*, v. 48, n. 1, pp. 111-128, 2000.

[65] Zanakis, S. H., Evans, J. "Heuristic optimization: why, when and how to use it". *Inferfaces*, v. 11, pp. 84-90, 1981.

[66] Wolsey, L. A. *Integer Programming*, New York, John Wiley & Sons, 1998.

Anexo A

Resolução de um problema de *bin packing*: saída gerada durante a execução da nossa implementação do *branch-and-price* para o *bin packing* dado por $m = 10$, $\ell_1 = 1$, $\ell_1 = 1$, $\ell_2 = 1$, $\ell_3 = 2$, $\ell_4 = 2$, $\ell_5 = 2$, $\ell_6 = 3$, $\ell_7 = 3$, $\ell_8 = 4$, $\ell_9 = 4$, $\ell_{10} = 5$ e $L = 7$.

```
[ariel]:/home3/efbueno/mestrado/implem> myprog

------------------------------------------------------------------
UFRJ / COPPE / PESC
Rio de Janeiro, Tue Mar  8 04:29:52 2005

Nelson Maculan Filho (maculan@cos.ufrj.br)
Elivelton Ferreira Bueno (efbueno@cos.ufrj.br)
------------------------------------------------------------------
Method: Branch-and-Price
Problem: 0-1 Set Partitioning with Knapsack columns
------------------------------------------------------------------

(IP) minimize cx
     subject to:  Ax = 1
                   x is binary

where: A'l <= L
------------------------------------------------------------------

Enter number of rows: 10

Creating the initial Restricted Master Problem...

Creating the initial subproblem (knapsack)...

XPRESS-MP: Reading Problem "knapsack"
Problem Statistics
          1 (      0 spare) rows
```

```
          10 (      0 spare) structural columns
          10 (     10 spare) non-zero elements
Global Statistics
          10 entities        0 sets        0 set members

XPRESS-MP: Reading Problem "binpacking"
Problem Statistics
          10 (      0 spare) rows
          10 (      0 spare) structural columns
          10 (      0 spare) non-zero elements
Global Statistics
           0 entities        0 sets        0 set members

About the branch-and-price search tree...

   ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
   Number of active nodes:                          1
   Best upper bound:                                10
   Best lower bound:                                4
   ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

New node selected

###################### Active node 0 ##############################


   Node number:                                     0
   Number of 1-branchings:                          0
   Number of 0-branchings:                          0

Definning the local node LP problem...

   Problem name:                                    probtest/binpacking
   Number of rows:                                  10
   Number of columns:                               10
   Number of nonzeros:                              10

Definning the local node pricing subproblem...

   Problem name:                                    probtest/knapsack
   Number of rows:                                  1
   Number of columns:                               10
   Number of nonzeros:                              10

Solving the local node LP problem...

   Number of generated columns:                     17
   Number of total columns:                         27
```

```
   LP status:                                        optimal
   LP objective values:
        10.00      7.00      7.00      7.00      7.00      7.00
         6.25      5.00      5.00      5.00      4.62      4.50
         4.50      4.50      4.29      4.17      4.08      4.00

Processing the branching...

   Selected branching pairs of rows:                 0 and 6
   Number of eliminated variables on 1-branching:    10
   Number of eliminated variables on 0-branching:    2

New node selected

About the branch-and-price search tree...

   ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
   Number of active nodes:                           1
   Best upper bound:                                 5
   Best lower bound:                                 4
   ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

###################### Active node 1 ################################


   Node number:                                      2
   Number of 1-branchings:                           0
   Number of 0-branchings:                           1

Definning the local node LP problem...

   Problem name:                                     probtest/binpacking2
   Number of rows:                                   10
   Number of columns:                                25
   Number of nonzeros:                               54

Definning the local node pricing subproblem...

   Problem name:                                     probtest/knapsack2
   Number of rows:                                   2
   Number of columns:                                10
   Number of nonzeros:                               12

Solving the local node LP problem...

   Number of generated columns:                      1
   Number of total columns:                          26
   LP status:                                        o
```

```
   LP objective values:
         4.57      4.00

Processing the branching...

   Selected branching pairs of rows:                5 and 7
   Number of eliminated variables on 1-branching:   4
   Number of eliminated variables on 0-branching:   1

New node selected

About the branch-and-price search tree...

   ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
   Number of active nodes:                          2
   Best upper bound:                                5
   Best lower bound:                                4
   ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

###################### Active node 2 ################################


   Node number:                                     4
   Number of 1-branchings:                          0
   Number of 0-branchings:                          2

Definning the local node LP problem...

   Problem name:                                    probtest/binpacking3
   Number of rows:                                  10
   Number of columns:                               25
   Number of nonzeros:                              55

Definning the local node pricing subproblem...

   Problem name:                                    probtest/knapsack3
   Number of rows:                                  3
   Number of columns:                               10
   Number of nonzeros:                              14

Solving the local node LP problem...

   Number of generated columns:                     1
   Number of total columns:                         26
   LP status:                                       o
   LP objective values:
         4.50      4.00
```

```
Node 4 pruned by optimality.

New node selected

About the branch-and-price search tree...

   ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
   Number of active nodes:                          1
   Best upper bound:                                4
   Best lower bound:                                4
   ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

   ------------------------------------------------------------
  |                                                            |
  |              F I N A L    S T A T I S T I C S              |
  |____________________________________________________________|

   Bin-Packing Compact
      - rows:                       20
      - columns:                    110

   Set Partitioning
      - rows:                       10

   Knapsack Subproblem
      - number of items:            10
      - smallest item weight:       1
      - average item weight:        2.7
      - largest item weight:        5
      - bin capacity:               7

   Optimal integer solution value: 4
   Total of generated columns:     29
   Maximum generated columns:      17
   Maxinum columns of a problem:   27
   Number of nodes
      - generated:                  5
      - processed:                  4
      - pruned by infeasiblity:     0
      - pruned by optimality:       1
      - pruned nodes by bound:      0
      - active:                     1
   ------------------------------------------------------------


------------------------------------------------------------------------
The end.
[ariel]:/home3/efbueno/mestrado/implem>
```