Universidade Estadual de Campinas
Instituto de Computação

Mauro Roberto Costa da Silva

Problemas de Empacotamento com Número Fixo de Recipientes

CAMPINAS
2025

Mauro Roberto Costa da Silva

Problemas de Empacotamento com Número Fixo de Recipientes

Tese apresentada ao Instituto de Computação da Universidade Estadual de Campinas como parte dos requisitos para a obtenção do título de Doutor em Ciência da Computação.

Orientador: Prof. Dr. Rafael Crivellari Saliba Schouery
Coorientador: Prof. Dr. Lehilton Lelis Chaves Pedrosa

Este exemplar corresponde à versão final da Tese defendida por Mauro Roberto Costa da Silva e orientada pelo Prof. Dr. Rafael Crivellari Saliba Schouery.

CAMPINAS
2025

Ficha catalográfica
Universidade Estadual de Campinas (UNICAMP)
Biblioteca do Instituto de Matemática, Estatística e Computação Científica
Ana Regina Machado - CRB 8/5467

Si38p

Silva, Mauro Roberto Costa da, 1995-
Problemas de empacotamento com número fixo de recipientes / Mauro Roberto Costa da Silva. – Campinas, SP : [s.n.], 2025.

Orientador: Rafael Crivellari Saliba Schouery.
Coorientador: Lehilton Lelis Chaves Pedrosa.
Tese (doutorado) – Universidade Estadual de Campinas (UNICAMP), Instituto de Computação.

1. Problemas de empacotamento. 2. Algoritmos de aproximação. 3. Formulação arc-flow. 4. Algoritmos branch-and-price. 5. Algoritmos exatos. I. Schouery, Rafael Crivellari Saliba, 1986-. II. Pedrosa, Lehilton Lelis Chaves, 1985-. III. Universidade Estadual de Campinas (UNICAMP). Instituto de Computação. IV. Título.

Informações complementares

Título em outro idioma: Packing problems with fixed number of bins
Palavras-chave em inglês:
Packing problems
Approximation algorithms
Arc-flow formulation
Branch-and-price algorithms
Exact algorithms
Área de concentração: Ciência da Computação
Titulação: Doutor em Ciência da Computação
Banca examinadora:
Rafael Crivellari Saliba Schouery [Orientador]
Santiago Valdés Ravelo
Anand Subramanian
Edna Ayako Hoshino
André Luís Vignatti
Data de defesa: 27-01-2025
Programa de Pós-Graduação: Ciência da Computação

Identificação e informações acadêmicas do(a) aluno(a)
- ORCID do autor: https://orcid.org/0000-0002-0691-4554
- Currículo Lattes do autor: http://lattes.cnpq.br/4787353202383977

Universidade Estadual de Campinas
Instituto de Computação

Mauro Roberto Costa da Silva

Problemas de Empacotamento com Número Fixo de Recipientes

Banca Examinadora:

- Prof. Dr. Rafael Crivellari Saliba Schouery
  Instituto de Computação - Unicamp
- Prof. Dr. Santiago Valdes Ravelo
  Instituto de Computação - Unicamp
- Prof. Dr. Anand Subramanian
  Universidade Federal da Paraíba
- Profa. Dra. Edna Ayako Hoshino
  Universidade Federal de Mato Grosso do Sul
- Prof. Dr. André Luís Vignatti
  Universidade Federal do Paraná

A ata da defesa, assinada pelos membros da Comissão Examinadora, consta no SIGA/Sistema de Fluxo de Dissertação/Tese e na Secretaria do Programa da Unidade.

Campinas, 27 de janeiro de 2025

Agradecimentos

Aos Meus Pais, Maria José e Jusmaro Correia, e à minha noiva Bruna Barros.

À toda minha família e amigos.

Aos Professores Rafael Schouery e Lehilton Pedrosa, pela excelente orientação, disponibilidade e paciência.

Aos membros da Banca Examinadora, Anand Subramanian, Edna Ayako Hoshino, André Luís Vignatti e Santiago Valdes Ravelo pelo tempo, pelas valiosas colaborações e sugestões.

Aos membros do Laboratório de Otimização e Combinatória (LOCo) pelas excelentes contribuições e companhias.

O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Código de Financiamento 001 e da Fundação de Amparo à Pesquisa do Estado de São Paulo (FAPESP) por meio dos projetos #2020/13162-2 e #2015/11937-9.

Resumo

Problemas de empacotamento com número fixo de recipientes consistem em, dado um conjunto de itens e um número de recipientes de uma determinada capacidade, empacotar um subconjunto dos itens nos recipientes, maximizando ou minimizando uma certa função objetivo, tal como a soma dos valores dos itens empacotados. O Binary Knapsack Problem (KP), o Generalized Assignment Problem (GAP), o Multiple Knapsack Problem (MKP) e o Bin Packing Problem (BPP) são exemplos clássicos de problemas de empacotamento, mas apenas o KP, o GAP e o MKP possuem número fixo de recipientes, já que o BPP permite que novos recipientes sejam criados. Outros problemas com essa característica são o Maxspace, o Positional Knapsack Problem e o Extensible Bin Packing. O Maxspace é um problema de disposição de propagandas, no qual desejamos exibir propagandas em um *banner*, maximizando a área ocupada pelas propagandas exibidas. O Positional Knapsack Problem é uma variante do Binary Knapsack Problem em que desejamos colocar itens em um recipiente e o valor do item pode variar de acordo com a posição que é empacotado. Já o Extensible Bin Packing (EBP) é um problema de empacotamento em que desejamos adicionar uma quantidade de itens em um número fixo de recipientes de mesmo tamanho, mas, diferentemente dos Maxspace e do PKP, o EBP exige que todos os itens sejam empacotados e permite que os recipientes sejam estendidos, aumentando o custo da solução proporcionalmente ao valor total de extensão. Neste trabalho, apresentamos um *Polynomial-Time Approximation Scheme* (PTAS) para uma variante do Maxspace, um *Fully Polynomial-Time Approximation Scheme* (FPTAS) para uma função de variação específica do PKP e um PTAS para o PKP com um conjunto de funções mais geral. Além disso, apresentamos algoritmos utilizando a formulação *Arc-Flow* e a técnica de *Branch-Cut-and-Price* para o EBP.

Abstract

Packing problems with a fixed number of bins consist of, given a set of items and several bins of a given capacity, packing a subset of the items in the bins, maximizing or minimizing a certain objective function, such as the sum of the values of the packed items. The Binary Knapsack Problem (KP), the Generalized Assignment Problem (GAP), the Multiple Knapsack Problem (MKP), and the Bin Packing Problem (BPP) are classic examples of packing problems, but only KP, GAP and MKP have a fixed number of bins since BPP allows new bins to be created. Other problems with this characteristic are Maxspace, Positional Knapsack Problem, and Extensible Bin Packing. The Maxspace is an advertising placement problem in which we want to display advertisements on a banner, maximizing the area occupied by the advertisements displayed. The Positional Knapsack Problem is a variant of Binary Knapsack Problem in which we want to place items in a knapsack, and the item's value can vary according to the position in which it is packed. And the Extensible Bin Packing (EBP) is a packing problem in which we want to place several items in a fixed number of bins of the same size, but, unlike Maxspace and PKP, EBP requires that all items be packed, and allows the bins to be extended, increasing the cost of the solution proportionally to the total value of the extension. The main objective of this work is to develop new approximation and exact algorithms for packing problems with a fixed number of bins, more specifically to Maxspace, Positional Knapsack Problem, and Extensible Bin Packing, taking into account variants found in practice for these problems. In this work, we present a Polynomial-Time Approximation Scheme (PTAS) for a variant of Maxspace, a Fully Polynomial-Time Approximation Scheme (FPTAS) for a specific variation function of PKP, and a PTAS for PKP with a more general set of functions. In addition, we present algorithms using the Arc-Flow formulation and the Branch-Cut-and-Price technique for EBP.

Lista de Figuras

Lista de Tabelas

Sumário

Capítulo 1

Introdução

Problemas de empacotamento possuem muitas aplicações práticas, principalmente na logística dos setores de indústrias e serviços. A logística é responsável por uma parcela considerável do custo dos serviços e dos produtos. O custo total dessa área representou 26% do PIB brasileiro e 9,5% do PIB estadunidense em 2008 [73]. Uma otimização nesses processos pode reduzir os custos e tornar os produtos mais baratos e competitivos.

Esses problemas também despertam um grande interesse da literatura. Delorme [27] apresentou um levantamento do número de artigos científicos sobre problemas de empacotamento ou problemas de corte de estoque em diferentes fontes bibliográficas entre os anos de 1991 e 2016. Podemos verificar que houve um aumento de interesse nesses problemas na Figura 1.1 (extraída de [27]). Dessa forma, é importante encontrar bons resultados e validá-los em veículos de divulgação internacional. Além disso, os problemas abordados representam um grande desafio teórico, pois são NP-difíceis.

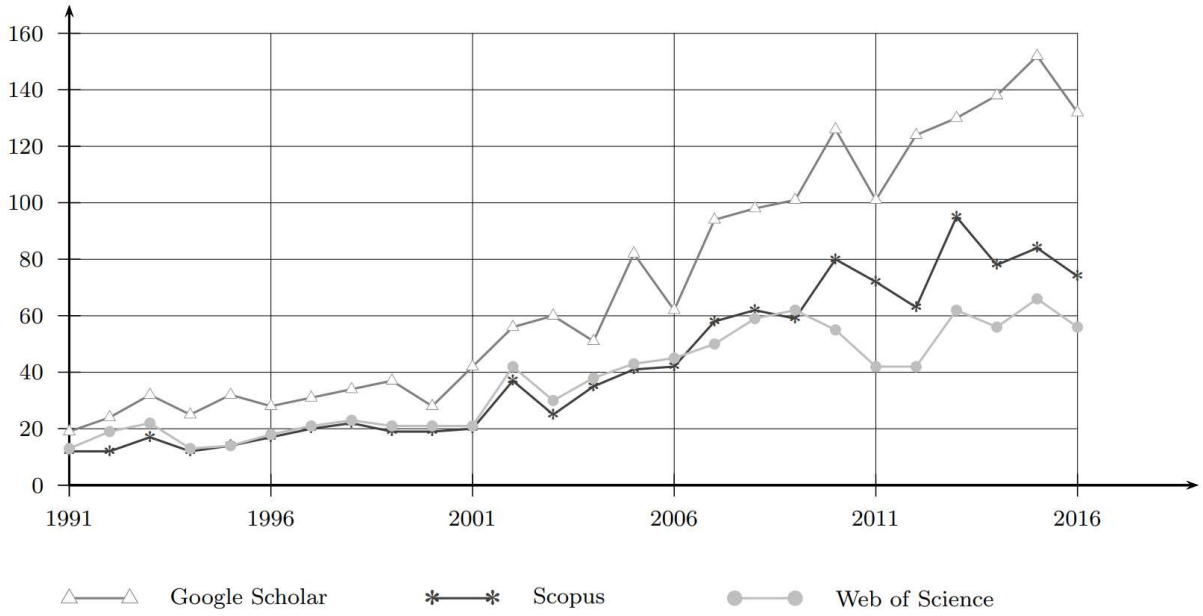

Figura 1.1: Número de artigos sobre problemas de empacotamento ou problemas de corte de estoque entre 1991 e 2016 [27].

Alguns problemas clássicos de empacotamento são o Binary Knapsack Problem, o Multiple Knapsack Problem e o Bin Packing Problem. No Binary Knapsack

Problem (KP), a tarefa é empacotar itens em uma mochila de determinada capacidade. Como pode não ser possível adicionar todos os itens à mochila, escolhe-se um subconjunto para ser empacotado que maximize o valor total. Formalmente, uma instância do KP consiste em uma mochila de capacidade L, um conjunto de itens $I = \{i_1, i_2, \ldots, i_n\}$ e, para cada item i, um valor $v_i > 0$ e um tamanho $s_i > 0$. O objetivo é encontrar um subconjunto $S \subseteq I$ que maximize $\sum_{i \in S} v_i$ e que não exceda a capacidade da mochila, ou seja, $\sum_{i \in S} s_i \leq L$ [55]. Este é um dos problemas mais conhecidos da Computação, possuindo muitas variantes e generalizações [10, 11], como o Multiple Knapsack Problem e o Bin Packing Problem.

O Generalized Assignment Problem (GAP) é um problema em que, dado um conjunto de itens $I = \{i_1, i_2, \ldots, i_k\}$ e um conjunto de m recipientes $M = \{M_1, \ldots, M_m\}$, tem como objetivo empacotar um subconjunto dos itens nos recipientes, maximizando o valor dos itens empacotados e respeitando a capacidade dos recipientes. Cada item i tem um valor v_{ij} e um tamanho s_{ij} ao ser empacotado em um recipiente j, e cada recipiente j possui uma capacidade limitada L_j. O objetivo do GAP é encontrar $S = \{I_1, I_2, \ldots, I_m\}$ com m subconjuntos de itens e $\bigcap_{I_j \in S} I_j = \emptyset$, tal que S maximize o valor dos itens, dado por $\sum_{I_j \in S} \sum_{i \in I_j} v_{ij}$, e que cada subconjunto de itens $I_j \in S$ possa ser empacotado na mochila $m_j \in M$ para todo j, ou seja, $\sum_{i \in I_j} s_{ij} \leq L_j$ para todo j [13, 33].

O Multiple Knapsack Problem (MKP) é uma generalização do KP e um caso particular do GAP em que cada item i possui o mesmo valor e tamanho para todos os recipientes. No MKP, são dados um conjunto de itens $I = \{i_1, i_2, \ldots, i_k\}$, em que cada item i possui valor v_i e tamanho s_i, e um conjunto de m recipientes $M = \{M_1, M_2, \ldots, M_m\}$, em que cada recipiente j possui uma capacidade L_j. Desejamos encontrar $S = \{I_1, I_2, \ldots, I_m\}$ com m subconjuntos de itens e $\bigcap_{I_j \in S} I_j = \emptyset$, tal que S maximize o valor dos itens, dado por $\sum_{I_j \in S} \sum_{i \in I_j} v_i$, e que cada subconjunto de itens $I_j \in S$ possa ser empacotado na mochila $m_j \in M$ para todo j, ou seja, $\sum_{i \in I_j} s_i \leq L_j$ para todo j [13].

Já o Bin Packing Problem (BPP) é um problema de empacotamento em que o número de recipientes (neste caso, também chamados de *bins*) é ilimitado e desejamos empacotar todos os itens, minimizando o número de recipientes usados. Formalmente, esse problema pode ser definido da seguinte maneira: dado um conjunto de itens $I = \{i_1, i_2, \ldots, i_n\}$, em que cada item i possui tamanho s_i, e um conjunto ilimitado de recipientes idênticos de capacidade L; encontre $S = \{I_1, I_2, \ldots, I_m\}$ com m subconjuntos de itens, tal que $\bigcup_{I_j \in S} I_j = I$, m seja mínimo e todo subconjunto $I_j \in S$ possa ser empacotado em uma mochila de capacidade L, ou seja, $\sum_{i \in I_j} s_i \leq L$ para todo $I_j \in S$ [53].

Problemas de empacotamento com número fixo de recipientes são aqueles que possuem um dado número de recipientes dado na entrada ou na definição do problema, não permitindo a criação de novos recipientes para o empacotamento dos itens. O KP, o GAP e o MKP são, portanto, exemplos de problemas dessa classe (o BPP não, já que permite a criação de novos recipientes).

Essa classe de problemas com um número fixo de recipientes introduz um novo desafio: a seleção dos itens a serem empacotados. Ao contrário dos problemas com recipientes ilimitados, em que todos os itens precisam ser empacotados, nos problemas com número fixo de recipientes pode ser necessário escolher quais itens serão empacotados e como organizá-los dentro dos recipientes disponíveis. Neste trabalho, abordaremos essa classe

de problemas, mais especificamente os problemas MAXSPACE, POSITIONAL KNAPSACK PROBLEM e EXTENSIBLE BIN PACKING.

O MAXSPACE é um problema de disposição de propagandas, no qual desejamos exibir propagandas em um *banner*, maximizando a área ocupada pelas propagandas exibidas. Cada propaganda possui uma quantidade de cópias e as propagandas exibidas no *banner* são alteradas a cada intervalo de tempo. A quantidade de intervalos de tempo é fixa, e podemos considerar que cada intervalo de tempo é um recipiente de capacidade igual à do *banner*. Problemas de disposição de propagandas são comuns em sites que oferecem serviços de forma gratuita enquanto exibem propagandas aos seus usuários, como Google e Facebook [59]. Anualmente, a receita de publicidade na web movimenta centenas de bilhões de dólares [48], o que faz com que, mesmo pequenas melhorias na forma de exibi-las, tenham um grande impacto na receita.

Introduzimos uma variante do BINARY KNAPSACK PROBLEM, que denominamos POSITIONAL KNAPSACK PROBLEM (PKP), em que o ganho do item varia de acordo com a posição em que é adicionado. O PKP também foi motivado pela exibição de anúncios em *layouts* de *sites*. Mas, diferentemente dos problemas de disposição de propagandas apresentados anteriormente, no PKP a receita esperada para a exibição de um anúncio varia de acordo com sua proximidade do topo.

Já o EXTENSIBLE BIN PACKING (EBP) é um problema de empacotamento em que desejamos adicionar uma quantidade de itens em um número fixo de recipientes de mesmo tamanho. Diferentemente do MKP, no EBP todos os itens devem ser adicionados e as capacidades dos recipientes podem ser excedidas. O custo de um recipiente é dado pela sua capacidade mais o espaço excedido naquele recipiente. O objetivo do EBP é minimizar os custos dos recipientes. Embora o EBP exija que todos os itens sejam empacotados, a necessidade de escolher quais itens adicionar permanece. Isso porque podemos resolver o problema decidindo quais itens tem pelo menos uma parte dentro de algum recipiente e como empacotá-los. Já os itens restantes, que estão totalmente fora dos recipientes, podem ser empacotados em qualquer recipiente sem alterar o valor da solução. O EBP surge em uma variedade de problemas de escalonamento e alocação de armazenamento (*storage allocation*). Considere, por exemplo, o caso de um conjunto de tarefas que devem ser atribuídas a um conjunto de recursos idênticos (por exemplo, trabalhadores) que estão disponíveis a um custo fixo por um determinado período (o tempo regular) e podem ser adquiridos a um custo adicional, proporcional ao tempo (horas extras), em caso de necessidade. O problema de atribuir as tarefas aos trabalhadores com o objetivo de minimizar o custo total é equivalente ao EBP. [26].

No Capítulo 2, apresentamos os principais conceitos utilizados durante todo o trabalho. No Capítulo 3, definimos formalmente o MAXSPACE e algumas variantes, apresentamos a literatura do problema e um *Polynomial-Time Approximation Scheme* (PTAS) para uma variante do problema, que publicamos no periódico "*Theory of Computing Systems* - TOCS" [71]. Esse é o melhor fator de aproximação possível para esse problema, já que generaliza o MULTIPLE KNAPSACK PROBLEM, que é fortemente NP-difícil até mesmo com dois recipientes. No Capítulo 4, definimos formalmente o PKP, mostramos que é NP-difícil até mesmo para uma função de variação específica e apresentamos um *Fully Polynomial-Time Approximation Scheme* (FPTAS) para o problema com essa função, e um PTAS

para um conjunto de funções mais geral. Apresentamos uma versão preliminar do FPTAS para o Positional Knapsack Problem no "*XII Latin-American Algorithms, Graphs and Optimization Symposium* - LAGOS" [70] e submetemos a versão completa do FPTAS e do PTAS para o PKP para um periódico internacional. No Capítulo 5, definimos formalmente o EBP e apresentamos algoritmos exatos, utilizando as técnicas de *Branch-Cut-and-Price* e *Arc-flow*. Também apresentamos e analisamos os resultados obtidos pelos algoritmos desenvolvidos. No Capítulo 6, fazemos nossas considerações finais e indicamos quais serão os próximos passos.

Capítulo 2

Preliminares

Na Teoria da Computação, um algoritmo é chamado de eficiente se resolve um determinado problema em tempo polinomial no tamanho da representação da entrada. Um problema é chamado de *Problema de Decisão* quando a solução consiste em responder "sim" ou "não". A classe de todos os problemas de decisão que podem ser resolvidos em tempo polinomial é chamada de P e chamamos de NP a classe de problemas de decisão que podemos, dado um certificado, verificar se a resposta é "sim" para qualquer instância em tempo polinomial. A grande questão em aberto da Teoria da Computação é determinar se P = NP.

Chamamos de NP-difíceis os problemas que são tão difíceis quanto qualquer problema em NP. Como o MAXSPACE, o EXTENSIBLE BIN PACKING, e o POSITIONAL KNAPSACK PROBLEM são NP-difíceis, não existem algoritmos para resolvê-los de forma eficiente, a menos que P = NP [68].

Neste trabalho, criamos novos algoritmos exatos e algoritmos de aproximação para alguns problemas de empacotamento com número fixo de recipientes. A seguir, apresentamos essas abordagens de pesquisa.

2.1 Algoritmos Exatos

No contexto de algoritmos exatos, procuramos desenvolver algoritmos para encontrar uma solução ótima. Em problemas de otimização combinatória existe, em geral, um número muito grande de soluções viáveis e enumerá-las pode tomar muito tempo. Por isso, é importante projetar um algoritmo que encontre soluções ótimas rapidamente. Nesse contexto, algoritmos baseados em Programação Linear, Geração de Colunas, *Branch-and-Price* e *Arc-flow* são bastante aplicados.

2.1.1 Programação Linear

A Programação Linear (PL) é um modelo matemático que descreve um problema de otimização e possui uma função objetivo linear de várias variáveis e um conjunto de restrições que são desigualdades lineares sobre essas variáveis. As restrições lineares definem um poliedro convexo, chamado de conjunto dos pontos viáveis. Como a função objetivo também é linear, todo ótimo local também é um ótimo global. A linearidade da função objetivo

também implica que uma solução ótima só pode ocorrer em um ponto da fronteira do poliedro convexo. Essas propriedades permitem que um modelo de PL seja resolvido em tempo polinomial [6].

Quando existem restrições que se contradizem dizemos que o modelo é inviável (o poliedro é vazio) e quando a função objetivo é ilimitada dizemos que o modelo é ilimitado. Em ambos os casos, não é possível obter solução para o modelo [6].

Seja x um vetor de variáveis e sejam A, b e c vetores de constantes para um problema de minimização. Um programa linear para esse problema pode ser escrito como:

$$\text{minimizar } c^T x \tag{2.1}$$
$$\text{sujeito a: } Ax \geq b \tag{2.2}$$
$$x \geq 0 \tag{2.3}$$

Em (2.1) temos a função objetivo do modelo e em (2.2) e (2.3) temos as restrições do modelo. Em um modelo de PL, todas as variáveis são lineares. Também existem modelos de Programação Linear que possuem variáveis inteiras, é a chamada Programação Linear Inteira (PLI), uma das principais abordagens para projetar algoritmos exatos. Modelos em que existem variáveis lineares e inteiras também podem ser chamados de modelos de Programação Linear Mista (PLM).

Ao contrário da PL, que pode ser resolvida em tempo polinomial, muitos modelos de PLI não podem, a menos que P = NP. Por isso, quando usamos PLI para modelar problemas NP-difíceis, não conseguimos um algoritmo polinomial. Usamos o termo *relaxação linear* ou modelo *relaxado* para nos referir a um modelo de PL obtido a partir de um modelo de PLI com suas variáveis inteiras modificadas para racionais. A seguir, apresentamos um modelo usando Programação Linear Inteira proposto Martello e Toth [66] para o Bin Packing Problem.

$$(T) \text{ minimizar } \sum_{j=1}^{K} y_j \tag{2.4}$$
$$\text{sujeito a: } \sum_{j=1}^{K} x_{i,j} \geq 1 \qquad \forall i \in I \tag{2.5}$$
$$\sum_{i \in I} s_i x_{i,j} \leq L y_j \qquad j = 1, 2, \ldots, K \tag{2.6}$$
$$x_{i,j} \in \{0, 1\} \qquad \forall i \in I, j = 1, 2, \ldots, K \tag{2.7}$$
$$y_j \in \{0, 1\} \qquad j = 1, 2, \ldots, K \tag{2.8}$$

Seja K um limitante superior conhecido para o número de *bins* necessários para uma solução (por exemplo o número de itens), o modelo considera que existem K *bins* e a função objetivo (2.4) minimiza quantos desses *bins* serão de fato utilizados na solução. A variável binária y_j indica se um *bin* j foi utilizado ou não e a variável binária $x_{i,j}$ indica se um item i foi adicionado ao *bin* j. As restrições (2.5) garantem que cada item foi

adicionado pelo menos uma vez e as restrições (2.6) garantem que o preenchimento de cada *bin* j é no máximo L quando o *bin* foi utilizado.

O Teorema da Dualidade [20] em Programação Linear relaciona um problema de maximização com um problema de minimização, um chamado de Primal e outro de Dual, mostrando que os dois estão relacionados, de forma que toda solução viável de um é um limitante para o outro e, se um dos problemas tem uma solução ótima, o outro também tem e os valores coincidem. O Problema Dual é comumente utilizado para obter limitantes para o Problema Primal. A formulação a seguir corresponde ao modelo dual da relaxação linear da formulação (T) apresentada anteriormente para o BPP.

$$\text{maximizar} \sum_{i \in I} \gamma_i \tag{2.9}$$

$$\text{sujeito a: } \gamma_i - \lambda_j s_i \geq 0 \qquad \forall i \in I, \forall j = 1, 2, \ldots, K \tag{2.10}$$

$$L\lambda_j \geq 1 \qquad j = 1, 2, \ldots, K \tag{2.11}$$

$$\gamma_i \geq 0 \qquad \forall i \in I \tag{2.12}$$

$$\lambda_j \geq 0 \qquad j = 1, 2, \ldots, K \tag{2.13}$$

O modelo dual troca os papéis das variáveis e das restrições do primal, maximizando uma função linear sujeita a restrições lineares, em contraste com o primal que minimiza uma função linear sujeita a restrições lineares. Nesse modelo, as variáveis γ estão relacionadas às restrições (2.5) e as variáveis λ estão relacionadas às restrições (2.6). Já as restrições (2.10) estão relacionadas às variáveis x do primal e as restrições (2.11) estão relacionadas às variáveis y do primal.

2.1.2 Geração de Colunas

A ideia geral da geração de colunas é que muitos programas lineares são grandes demais para considerar todas as variáveis explicitamente. A premissa é que a maioria das variáveis assumirão valor zero na solução ótima, por isso, apenas um subconjunto de variáveis precisa ser considerado ao resolver o problema. A geração de colunas aproveita essa ideia para gerar apenas as variáveis que têm o potencial de melhorar a função objetivo. O problema a ser resolvido é dividido em três problemas: o Problema Principal (ou *Master Problem*-(MP)), o Problema Principal Restrito (ou *Restricted Master Problem*-(RMP)) e o Subproblema. O Problema Principal é o modelo do problema original, o Problema Principal Restrito é o Problema Principal com apenas um subconjunto de variáveis sendo considerado e o Subproblema é um novo problema utilizado para identificar uma nova variável, que posteriormente será adicionada ao Problema Principal Restrito [64].

No Bin Packing Problem, um padrão p é uma maneira de empacotar itens em um *bin* respeitando a condição $\sum_{i \in I} a_{ip} s_i \leq L$, em que a_{ip} indica se um item i está no padrão p. Chamaremos de P o conjunto com todos os padrões de uma instância. Nesse contexto, a seguinte formulação para o BPP pode ser obtida [37]:

$$\text{(MP) minimizar} \sum_{p \in P} \lambda_p \tag{2.14}$$

$$\text{sujeito a:} \sum_{p \in P} a_{ip} \lambda_p \geq 1 \qquad \forall i \in I \tag{2.15}$$

$$\lambda_p \in \{0, 1\} \qquad \forall p \in P \tag{2.16}$$

As variáveis binárias λ_p indicam se um padrão p foi utilizado na solução, a função objetivo minimiza a quantidade de padrões utilizados e as restrições (2.15) garantem que cada item foi adicionado pelo menos uma vez.

Note que o número de padrões pode ser exponencial na quantidade de itens, o que torna inviável resolver esse modelo rapidamente, a menos que P = NP. Nesse contexto, podemos aplicar a técnica de geração de colunas, criando um problema em que apenas um subconjunto de variáveis será considerado. Assim, utilizamos a formulação (MP) considerando apenas um subconjunto $\bar{P} \subseteq P$ de padrões para ser o Problema Principal Restrito (RMP). A ideia é que o RMP possa ser resolvido em tempo hábil para mais instâncias do que o modelo original, mas ainda ficamos com o problema de decidir quais padrões devem ser considerados. Para isso, utilizamos o modelo *dual* do RMP, apresentado a seguir:

$$\text{(DRMP) maximizar} \sum_{i \in I} \pi_i \tag{2.17}$$

$$\text{sujeito a:} \sum_{i \in P} a_{ip} \pi_p \leq 1 \qquad \forall p \in \bar{P} \tag{2.18}$$

$$\pi_i \in \mathbb{R}^+ \qquad \forall i \in I \tag{2.19}$$

Definimos um conjunto de padrões inicial para $\bar{P}$ de forma que exista uma solução viável e adicionamos novos padrões a partir da restrição (2.18).

O *custo reduzido* de uma variável na programação linear representa o quanto o coeficiente da função objetivo precisaria mudar para que essa variável entrasse na solução ótima com um valor positivo. No modelo (DRMP), o custo reduzido de um padrão é dado por $1 - \sum_{i \in P} a_{ip} \pi_p$. Estamos interessados em padrões que minimizem o custo reduzido.

Dizemos que um padrão p está *violado* quando ele viola a restrição (2.18). Nem sempre estamos interessados em todos os padrões violados, mas sim em decidir se existe algum padrão violado. Nesse modelo, o Subproblema consiste em encontrar um padrão violado p que maximize

$$\sum_{i \in P} a_{ip} \pi_p.$$

Note que isso é equivalente a encontrar um padrão de custo reduzido mínimo e que resolver o subproblema pode ser NP-difícil.

Sempre que novos padrões são adicionados ao RMP, o modelo deve ser reotimizado, atualizando os valores das variáveis do modelo *dual* (DRMP) e podendo identificar assim

novos padrões violados.

2.1.3 Branch-and-Price

O *Branch-and-Price* é um híbrido entre o método *Branch-and-Bound* e a técnica de geração de colunas. O método *Branch-and-Bound* consiste em uma enumeração sistemática de soluções candidatas no espaço de busca. O conjunto de soluções candidatas pode ser pensado como formando uma árvore enraizada com o conjunto completo na raiz. O algoritmo explora ramos desta árvore, que representam subconjuntos do conjunto de soluções. Antes de enumerar as soluções candidatas de uma ramificação, ela é verificada em relação aos limites superior e/ou inferior conhecidos da solução ótima e é descartada se não puder produzir uma solução melhor do que a melhor encontrada pelo algoritmo até o momento [63].

Um *limitante primal* corresponde a um limite do valor da função objetivo do problema primal, de forma que nenhuma solução ultrapasse esse limite dentro da região viável. Um *limitante dual* é um valor que corresponde a um limite do valor da função objetivo do problema dual, estabelecendo um limite superior ou inferior para o valor ótimo da função objetivo do problema primal, conforme o teorema da dualidade [20] da programação linear. Em geral, o limitante dual é obtido resolvendo-se a relaxação linear do problema associado ao nó da árvore de enumeração do *Branch-and-Bound*. Desta forma, o *Branch-and-Price* consiste no *Branch-and-Bound* no qual a relaxação linear em cada nó é resolvido por geração de colunas. Ele é usado quando o problema de PLI sendo resolvido é modelado com uma quantidade muito grande de variáveis.

Um algoritmo *Branch-and-Price* começa a partir do problema principal e, para realizar a otimização, o subproblema da geração de colunas (aqui também denominado de problema de precificação) é resolvido a fim de encontrar uma coluna que melhora a função objetivo. Cada vez que uma coluna é encontrada, ela é adicionada ao problema principal e a relaxação é reotimizada. Se nenhuma coluna puder entrar no problema e a solução para a relaxação não for inteira, então ocorre a ramificação [4].

Na ramificação ou *branching*, estamos interessados em particionar o espaço de soluções do modelo, fixando valores de variáveis e resolvendo o subproblema gerado. No exemplo do modelo (2.14)-(2.16) de geração de colunas apresentado na Seção 2.1.2, podemos fixar os valores das variáveis x_p em 0 ou 1 de acordo com alguma estratégia, como arredondar padrões em que x_p está próximo 0.5 na solução atual para 0 ou para 1. Estratégias de *branching* impactam diretamente no desempenho do algoritmo, pois fazem com que a árvore aumente ou diminua de tamanho, além de possibilitarem que o algoritmo chegue mais cedo em soluções melhores.

Após fixar variáveis em um nó, o algoritmo cria uma ramificação para o subproblema gerado. Em cada ramificação, um limitante para o valor do novo modelo pode ser comparado com a melhor solução encontrada, podando o nó atual quando esse limitante não puder melhorar a solução.

Nos nós que não foram podados, o processo de geração de colunas se repete: o modelo é resolvido e novos padrões são gerados. Uma solução é encontrada sempre que nenhum padrão pode ser adicionado ao modelo e a solução para relaxação for inteira. Nenhuma

ramificação é criada a partir de um nó que encontra uma solução inteira. O algoritmo para quando todas as ramificações encontram soluções inteiras ou são podadas por algum limitante, e retorna a melhor solução encontrada. A Figura 2.1 apresenta um diagrama resumindo o funcionamento de um nó do *Branch-and-Price*.

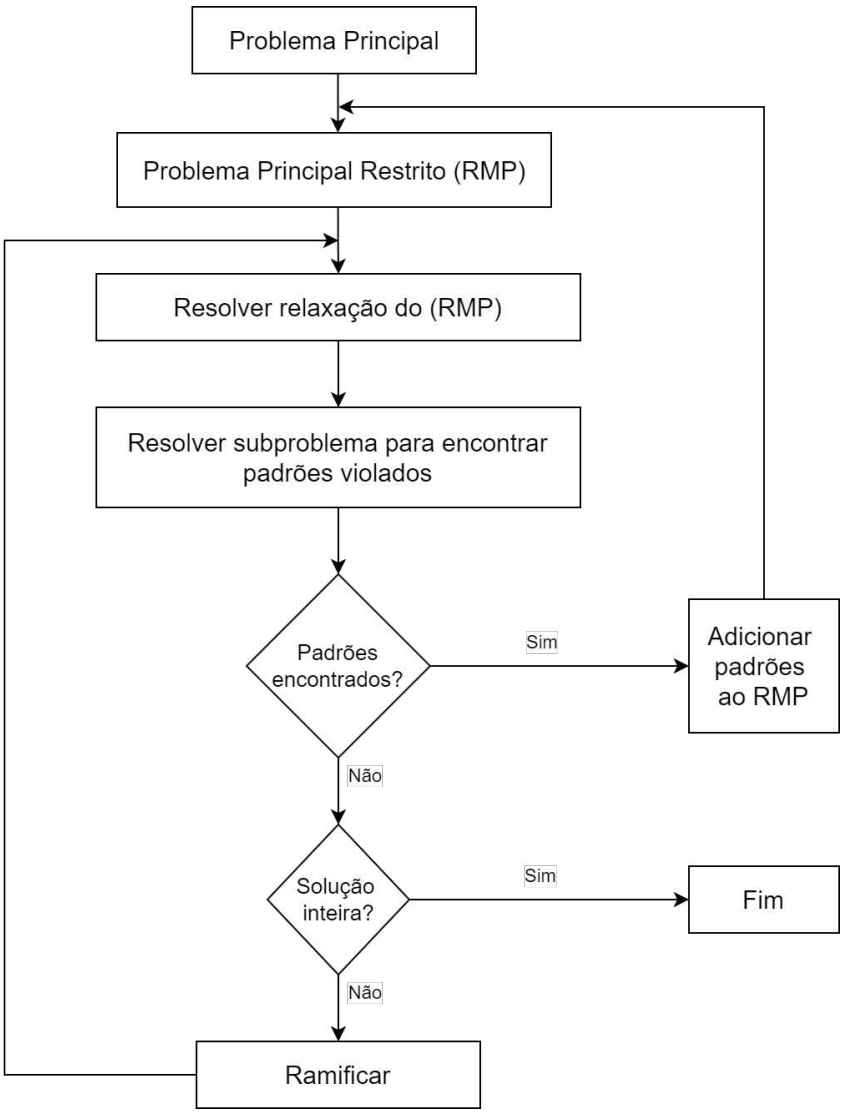

Figura 2.1: Diagrama do funcionamento de um nó do *branch-and-price*.

Branch-Cut-and-Price

O método de Plano de Corte foi introduzido por Gomory [38] e consiste em resolver a relaxação linear de um modelo PLI para um problema e derivar de sua solução ótima uma ou mais desigualdades (chamadas de *cortes*) violadas pela própria solução, mas respeitadas por qualquer solução inteira viável do problema. Essas desigualdades são adicionadas ao problema, que é reotimizado. Então, o método é aplicado novamente à nova solução, prosseguindo enquanto a solução ótima se torna inteira e, portanto, viável e ótima também para o problema PLI original [38].

O *Branch-Cut-and-Price* é uma variante do *Branch-and-Price* em que adicionamos cortes ao modelo em cada nó da ramificação, usando o método de Plano de Cortes.

O objetivo com essa adição é fazer com que o limitante dado pela solução relaxada se aproxime do valor da solução ótima inteira.

2.1.4 Arc-Flow

O *Arc-Flow* [22] é uma formulação que já foi usada para modelar problemas como o Binary Knapsack Problem, o problema de determinar o caminho mais longo em um grafo direcionado e problemas de empacotamento como o Bin Packing Problem.

A seguir apresentamos uma formulação *Arc-Flow* para o Bin Packing Problem. Dados *bins* de capacidade inteira L e um conjunto de n itens com tamanhos diferentes $\{s_1, s_2, \cdots, s_m\}$, determinar o empacotamento de um *bin* pode ser modelado como o problema de encontrar um caminho em um grafo acíclico direcionado com $L+1$ vértices. Para isso, considere um grafo $G = (V, E)$ com $V = \{0, 1, 2, \ldots, L\}$ e $E = \{(i, j) : 0 \leq i < j < L$ e $j - i = s_d$ para algum $d \leq m\}$, o que significa que existe um arco direcionado entre dois vértices se há um item de tamanho correspondente à distância entre os vértices. Considere também arcos adicionais, chamados de arcos de perda, entre $(k, k+1)$, $k = 0, 1, \ldots, L-1$ correspondendo a partes desocupadas do *bin*. Há um empacotamento para um conjunto de itens se e somente se houver um caminho entre os vértices 0 e L em G que passa apenas por arcos de tamanhos correspondentes aos tamanhos dos itens ou arcos de perda. O comprimento dos arcos que constituem o caminho definem os tamanhos dos itens a serem empacotados [22]. Considere uma instância com 3 itens de tamanhos $\{3, 2, 1\}$ e *bins* de capacidade 6, um grafo para essa instância pode ser visto na Figura 2.2.

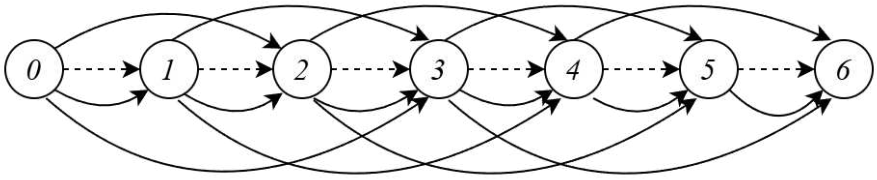

Figura 2.2: Exemplo de grafo do *arc-flow* considerando uma instância com os itens $\{3, 2, 1\}$ e *bins* de tamanho 6.

Para facilitar a visualização, a Figura 2.3 apresenta uma versão simplificada do grafo da Figura 2.2. Essa versão do grafo pode ser obtida aplicando as compressões apresentadas por De Carvalho [22]. Essas compressões também são úteis para reduzir o número de variáveis da formulação *arc-flow*, melhorando o desempenho de modelos que usam essa técnica.

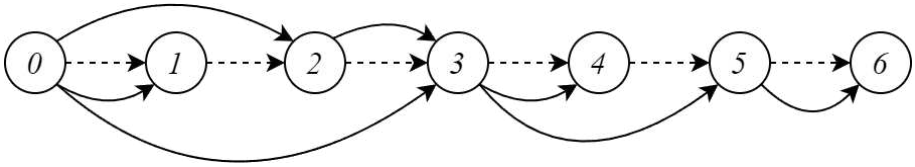

Figura 2.3: Exemplo de grafo do *arc-flow* considerando uma instância com os itens $\{3, 2, 1\}$, *bins* de tamanho 6 e com compressão de De Carvalho [22].

Nos grafos das Figuras 2.2 e 2.3, os vértices representam o preenchimento do *bin*, as arestas tracejadas são arcos de perda e as demais arestas representam um item da

instância de tamanho equivalente. Observe que, para cada possível combinação de itens em um *bin*, desconsiderando a ordem, existe um caminho no grafo, que vai do vértice 0 ao vértice $L = 6$, formado apenas por arcos de tamanho equivalente a esses itens e por arcos de perda. Dessa forma, podemos visualizar que o Bin Packing Problem é equivalente a encontrar o menor número de caminhos de 0 até L tal que o arco que representa cada item apareça em pelo menos um dos caminhos. Podemos generalizar e considerar que um arco de um determinado tamanho s representa todos os itens desse tamanho, e, nesse caso, precisamos que o número de arcos de tamanho s nos caminhos seja pelo menos d, em que d é a quantidade de itens de tamanho s na instância.

2.2 Algoritmos de Aproximação

Muitas vezes, na prática, é suficiente encontrar soluções com valor próximo ao valor de uma solução ótima e que sejam calculadas de forma eficiente. Um *Algoritmo de Aproximação* encontra, em tempo polinomial, uma solução viável que possui valor garantidamente a no máximo um fator do valor de uma solução ótima [81].

Considere um problema de otimização Π com função objetivo f_Π. Quando Π é um problema de maximização, dizemos que um algoritmo H é uma α-aproximação para Π se, para qualquer instância I de Π, ele consome tempo polinomial no tamanho da representação binária de I e produz uma solução S tal que $f_\Pi(S) \geq \alpha \cdot \text{OPT}$, com $\alpha \leq 1$ e em que OPT é o valor de uma solução ótima de I. Quando temos um problema Π de minimização, uma α-aproximação H é tal que H é um algoritmo polinomial no tamanho da representação binária da entrada I e $f_\Pi(S) \leq \alpha \cdot \text{OPT}$, para $\alpha \geq 1$.

Dizemos que H é um esquema de aproximação para Π se, para qualquer instância I de Π, e dado um parâmetro de erro $\varepsilon > 0$, H devolve uma solução S tal que $f_\Pi(S) \leq (1+\varepsilon)\text{OPT}$ se Π é um problema de minimização e $f_\Pi(S) \geq (1-\varepsilon)\text{OPT}$ se Π é um problema de maximização, em que OPT é o valor de uma solução ótima de I [81].

Um *Polynomial-Time Approximation Scheme* (PTAS) é um esquema de aproximação que executa em tempo polinomial no tamanho da instância, mas não necessariamente em $1/\varepsilon$. Um *Fully Polynomial-Time Approximation Scheme* (FPTAS) é um esquema de aproximação que executa em tempo polinomial no tamanho da instância e em $1/\varepsilon$. Já um *Asymptotic Polynomial-Time Approximation Scheme* (APTAS) é uma família de algoritmos para um problema de otimização Π que, para alguma constante k e qualquer instância I de Π, executam em tempo polinomial no tamanho de I, de modo que $f(I) \leq (1+\varepsilon)\text{OPT} + k$, em que OPT é o valor de uma solução ótima de I [81].

Existem diferentes estratégias para se obter um algoritmo de aproximação. Em particular, destacamos métodos combinatórios e métodos usando Programação Linear. Métodos combinatórios são aqueles que utilizam estratégias como combinação, permutação ou enumeração de conjuntos finitos para encontrar uma solução de fator desejado. Já métodos usando Programação Linear são aqueles que utilizam algum modelo de Programação Linear para resolver parte do problema, normalmente aplicando transformações, como arredondamentos de variáveis, nas soluções fracionárias obtidas através desses métodos. A seguir, apresentamos uma 1/2-aproximação combinatória para o caso do Binary

Knapsack Problem em que todos os itens possuem $v_i = s_i$.

Algoritmo 1 1/2-aproximação para o KP com $v_i = s_i$ para todo $i \in I$.

```
1: procedimento 1/2-APROX-KP(I, L)
2:     S' ← ∅
3:     para cada i ∈ I em ordem não-crescente de tamanho faça
4:         se s_i + Σ_{i∈S'} s_i ≤ L então
5:             S' ← S' ∪ {i}
6:     devolve S'
```

O Algoritmo 1 recebe um conjunto de itens I e o tamanho da mochila L, e devolve uma solução que adiciona os itens em ordem de tamanho (do maior para o menor) sempre que for possível, ignorando os itens que não couberem. No Lema 1, demonstramos que esse algoritmo é uma 1/2-aproximação para o Binary Knapsack Problem quando todos os itens possuem tamanho igual ao valor, ou seja, $v_i = s_i$ para todo item $i \in I$.

Lema 1. *O Algoritmo 1 é uma* 1/2*-aproximação para o Binary Knapsack Problem quando* $v_i = s_i$ *para todo item* $i \in I$.

Demonstração. Seja OPT uma solução ótima para uma dada instância com os itens I e tamanho de mochila L. Seja a função $f(X) = \sum_{i \in X} s_i$, o valor de OPT é $f(\text{OPT})$. Como todo item tem tamanho igual ao valor, uma solução ótima é aquela que maximiza a área ocupada pelos itens. Com isso, sabemos que $f(\text{OPT}) \leq L$.

Seja S a solução devolvida pelo algoritmo e seja $s_{max} \leftarrow \max_{i \in I} s_i$ o tamanho do maior item da instância. Note que o primeiro item está na solução, logo $f(S) \geq s_{max}$. Se $s_{max} \geq L/2$, então,

$$f(S) \geq L/2 \geq f(\text{OPT})/2.$$

Agora, considere que $s_{max} < L/2$. Se o algoritmo entra na condição da Linha 4 para todos os itens, então $S = I$ e, portanto, a solução é ótima. Assim, considere o caso em que existe algum item tal que o algoritmo não entra na condição da Linha 4 e seja i o primeiro item para qual essa condição foi rejeitada. Sabemos que nesse momento do algoritmo

$$f(S) > L - s_i > L - s_{max},$$

como $s_{max} < L/2$ e nenhum item é removido de S, então $f(S) > L/2$. E novamente temos que

$$f(S) > L/2 \geq f(\text{OPT})/2.$$

Dessa forma, mostramos que o algoritmo sempre devolve uma solução com valor pelo menos 1/2 do valor da solução ótima. Como o algoritmo possui tempo polinomial, temos uma 1/2-aproximação para o KP quando $v_i = s_i$ para todo item $i \in I$. □

Em problemas de empacotamento, geralmente usamos limitantes baseados na área ocupada pelos itens para mostrar o fator de aproximação, mas esses limitantes nem sempre são úteis, principalmente quando o valor do item não tem relação com a sua área, como ocorre no Maxspace com valor diferente do tamanho [17] e no caso geral do KP. Também

é comum arredondar o tamanho dos itens, permitindo que uma parte seja enumerada e a outra parte seja resolvida com erro pequeno. Essa é a estratégia utilizada no APTAS para o Bin Packing Problem apresentado por De La Vega e Lueker [23] e no PTAS para o Maxspace com *release dates*, *deadlines* e número constante de *bins* apresentado por Da Silva et al. [17].

Capítulo 3

Maxspace

Muitos sites (como Google, Yahoo!, Facebook e outros) oferecem serviços gratuitos enquanto exibem propagandas aos usuários. Cada site geralmente tem uma única faixa de altura fixa, reservada para exibir propagandas, e o conjunto de propagandas exibidas muda com base no tempo. Para esses sites, a propaganda é a principal fonte de receita. Portanto, é essencial encontrar a melhor maneira de exibi-las no tempo e espaço disponíveis, maximizando a receita [59]. Com receitas de publicidade na web na casa das centenas de bilhões de dólares [48], mesmo pequenos ganhos percentuais têm um grande impacto.

Sites como Facebook e Mercado Livre usam banners para exibir propagandas enquanto os usuários navegam. O Google exibe propagandas vendidas pelo *Google Ad Words* em seus resultados de busca em uma área limitada, na qual as propagandas estão em formato de texto e têm tamanhos que variam conforme o preço.

Consideramos a classe de problemas de disposição de propagandas introduzida por Adler et al. [1], em que, dado um conjunto $\mathcal{A} = \{A_1, A_2, \ldots, A_n\}$ de n propagandas, o objetivo é dispor um subconjunto $\mathcal{A}' \subseteq \mathcal{A}$ em um *banner* em K intervalos de tempo iguais, chamados *slots*. O conjunto de propagandas dispostas em um determinado *slot* j, $1 \leq j \leq K$, é representado por $B_j \subseteq \mathcal{A}'$. Cada propaganda A_i tem um *tamanho* s_i e uma *frequência* $w_i \leq K$. O tamanho s_i representa o espaço que A_i ocupa em um *slot* e a frequência w_i representa o número de *slots* que devem conter uma cópia de A_i. Uma propaganda A_i pode aparecer no máximo uma vez por *slot* e A_i é dita *disposta* se w_i cópias de A_i aparecem em *slots* com no máximo uma cópia por *slot* [1, 21].

Os principais problemas dessa classe são o Minspace e o Maxspace. No Minspace, todas as propagandas devem ser dispostas nos *slots* e o objetivo é minimizar a altura do *slot* mais alto, em que a altura de cada *slot* j é dada por $\sum_{A_i \in B_j} s_i$. No Maxspace, um limite superior L é especificado para a altura de cada *slot*. Uma solução viável para o Maxspace consiste em um subconjunto $\mathcal{A}' \subseteq \mathcal{A}$ de forma que toda propaganda $A_i \in \mathcal{A}'$ esteja disposta e nenhum *slot* tenha excedido a altura máxima L, isto é, para todo *slot* B_j, $\sum_{A_i \in B_j} s_i \leq L$. O objetivo do Maxspace é maximizar a receita, dada por $\sum_{A_i \in \mathcal{A}'} s_i \cdot w_i$, ou seja, o valor associado à propaganda é a sua altura multiplicada pela sua frequência [1, 21].

O Minspace e o Maxspace são fortemente NP-difíceis [1, 21]. O Maxspace não admite um FPTAS mesmo com $K = 2$, já que generaliza o Multiple Subset Sum Problem com capacidades idênticas (MSSP-I), que também não admite um FPTAS

mesmo para $K = 2$ [55]. Isso implica que um PTAS é a melhor aproximação possível para esse problema e suas generalizações, a menos que $P = NP$.

Considere as propagandas descritas na Tabela 3.1. Nas Figuras 3.1 e 3.2 são apresentados exemplos de soluções para o MINSPACE e o MAXSPACE, respectivamente, usando as propagandas da Tabela 3.1.

A_i	A_1	A_2	A_3	A_4	A_5	A_6	A_7
s_i	6	4	2	3	1	1	4
w_i	3	2	1	2	1	1	1

Tabela 3.1: Exemplos de Propagandas.

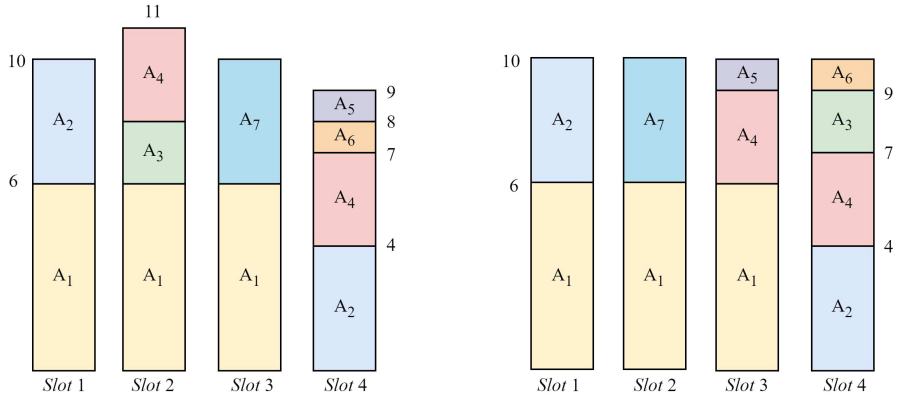

Figura 3.1: Exemplos de soluções para o MINSPACE usando as propagandas da Tabela 3.1 e considerando $K = 4$. Na esquerda temos uma solução viável com valor 11 e na direita temos uma solução ótima com valor 10 [21].

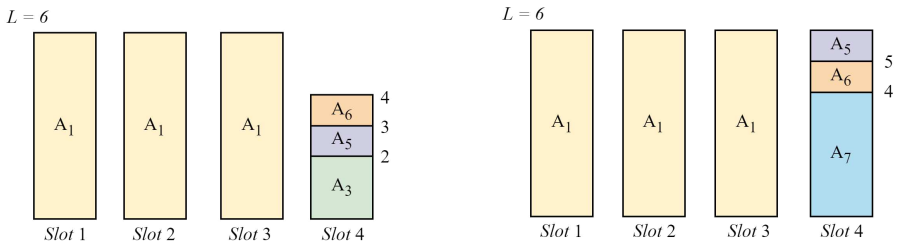

Figura 3.2: Exemplos de soluções para o MAXSPACE usando as propagandas da Tabela 3.1 com $L = 6$ e $K = 4$. Na esquerda temos uma solução viável de valor 22 e na direita temos uma solução ótima de valor 24 [21].

Adler et al. [1] apresentaram uma 1/2-aproximação para o MAXSPACE quando os tamanhos das propagandas formam uma sequência $s_1 \geq s_2 \geq \cdots \geq s_n$, tal que s_i é um múltiplo de s_{i+1} para todo i.

Posteriormente, Dawande et al. [21] definiram três casos especiais do problema MAXSPACE: MAX_w, $\text{MAX}_{K|w}$ e MAX_s. Em MAX_w, cada propaganda tem a mesma frequência w. Em $\text{MAX}_{K|w}$, cada propaganda tem a mesma frequência w e o número de *slots* K é um múltiplo de w. E em MAX_s, cada propaganda tem o mesmo tamanho s. Eles apresentaram três algoritmos de aproximação, uma $(1/4 + 1/4K)$-aproximação para o MAXSPACE, uma 1/3-aproximação para MAX_w e uma 1/2-aproximação para $\text{MAX}_{K|w}$. De forma análoga, também definiram três casos especiais para o MINSPACE: MIN_w, $\text{MIN}_{K|w}$ e MIN_s.

Freund e Naor [32] propuseram uma $(1/3-\varepsilon)$-aproximação para o MAXSPACE e uma $(1/2-\varepsilon)$-aproximação para o caso especial em que os tamanhos das propagandas estão no intervalo $[L/2, L]$.

Wendan e Dingwei [84] apresentaram a heurística *Largest-Size Most-Full* (LSMF) para o MAXSPACE e Kumar et al. [60] a utilizaram combinada a um algoritmo genético para desenvolver um algoritmo genético híbrido para o mesmo problema. Amiri e Menon [3] apresentam um modelo de Programação Linear Inteira para uma variante do MAXSPACE em que existe um conjunto de valores possíveis para a frequência de cada propaganda.

Em Boskamp et al. [8], o problema de adicionar propagandas em formato quadrado a um *banner* retangular é abordado. Embora também aborde propagandas, este problema é mais semelhante ao problema da mochila bidimensional do que ao MAXSPACE.

Kim e Moon [56] abordaram o MAXSPACE considerando quatro características: taxa de cliques (do inglês, *Clickthrough Rate*-CTR), competição entre propagandas, uma função objetivo baseada na CTR e frequência variável (como em Amiri e Menon[3]). Eles fornecem um modelo de programação inteira com uma função objetivo não linear e dois algoritmos, um heurístico e outro meta-heurístico, como metodologias de solução. Os testes foram realizados com instâncias com número de tipos de propagandas entre 4 e 100. Os algoritmos obtiveram bons resultados, mas não conseguiram resolver instâncias de número de tipos de propagandas entre 30 e 100 no tempo limite de 3600s.

Adler et al. [1] apresentaram a heurística *Largest-Size Least-Full* (LSLF), que é uma 2-aproximação para o MINSPACE. O algoritmo LSLF também é uma $(4/3 - w/(3K))$-aproximação para $\text{MIN}_{K|w}$ [21]. Dawande et al. [21] apresentaram uma 2-aproximação para o MINSPACE usando *LP Rouding* e Dean e Goemans [25] apresentaram uma 4/3-aproximação para o MINSPACE usando o algoritmo de Graham [41] para o Problema do Escalonamento.

3.1 Variantes

O MAXSPACE considera que o valor de uma propaganda é dada pela altura multiplicada pelo número de *slots* (ou unidades de tempo) em que ela aparece. Na prática, o valor de uma propaganda pode ser influenciado por outros fatores, como o número de cliques (ou a esperança do número de cliques) que a propaganda gera para o anunciante [9]. Definimos MAXSPACE-V como a variante do MAXSPACE em que cada propaganda A_i possui um valor v_i que pode não estar relacionado com $s_i w_i$.

A unidade de tempo relativa a cada *slot* pode representar segundos, minutos ou períodos maiores, como dias ou semanas. Muitas vezes, podemos considerar a ideia de *deadlines* e *release dates*. Uma propaganda pode possuir um *deadline*, que indica até qual período ela pode aparecer. Por exemplo, propagandas para o Natal devem ser exibidas até dia 24 de dezembro. De forma análoga, uma propaganda também pode possuir um *release date*, que indica a data a partir da qual a propaganda pode ser exibida.

Chamamos de MAXSPACE-D e MAXSPACE-R as variantes que, respectivamente, adicionam *deadlines* e *release dates*. No MAXSPACE-D, cada propaganda A_i possui um *deadline* $d_i \leq K$ que indica até qual *slot* A_i pode aparecer. De forma análoga, no MAXSPACE-

R, cada propaganda A_i possui um *release date* $r_i \geq 1$, que indica a partir de qual *slot* A_i pode aparecer.

Outra variante que pode ser interessante na prática é a que considera que cada propaganda possui um orçamento (ao invés de uma frequência) e que esse orçamento é reduzido quando a propaganda é exibida (considerando a probabilidade de clique, por exemplo). Para formalizar essa ideia de frequência variável propusemos a variante MAXSPACE-W, em que cada propaganda A_i possui um intervalo para a frequência que varia entre w_i^{min} e w_i^{max}.

As variantes MAXSPACE-RD e MAXSPACE-RDWV são obtidas a partir da junção de algumas variantes apresentadas anteriormente. O MAXSPACE-RD é uma variante que possui *release dates* e *deadlines* e o MAXSPACE-RDWV é uma variante que possui *release dates*, *deadlines*, frequência variável e valor generalizado.

Em Da Silva e Schouery [18], foram apresentados algoritmos baseados nas metaheurísticas *Greedy Randomized Adaptive Search Procedure* (GRASP), *Variable Neighborhood Search* (VNS), *Variable Neighborhood Descent* (VND), Busca Local e Busca Tabu para o MAXSPACE e para o MAXSPACE-RDWV. Os resultados obtidos nos testes computacionais foram comparados com o algoritmo genético híbrido Hybrid-GA proposto por Kumar et al. [60].

Em Da Silva et al. [17] foi apresentado um PTAS para o problema MAXSPACE-RD com número de *slots* constantes. Quando o número de *slots* é dado na entrada, o MAXSPACE é fortemente NP-difícil e não admite um esquema de aproximação totalmente em tempo polinomial (FPTAS) [21].

A seguir, apresentamos um PTAS para o problema MAXSPACE-RDV, uma generalização do MAXSPACE-RD em que o valor das propagandas é dado na entrada e não possui relação com o tamanho. No MAXSPACE-RDV, cada propaganda A_i possui valor v_i e esse valor é contabilizado para cada cópia adicionada de A_i, assim, o valor total de uma propaganda A_i é $v_i w_i$. Além de generalizar o MAXSPACE, essa variante é uma generalização do MULTIPLE KNAPSACK PROBLEM, que é fortemente NP-difícil até mesmo com $K = 2$.

3.2 Um PTAS para o MAXSPACE-RDV com um número constante de *slots*

A seguir, suponha que o número de *slots* K seja uma constante, $L = 1$, e $0 < s_i \leq 1$ para cada $A_i \in \mathcal{A}$. No MAXSPACE-RDV, definimos $f(B_j) = \sum_{A_i \in B_j} v_i$ como o valor de um *slot* B_j e $f(S) = \sum_{B_j \in S} f(B_j)$ como o valor de uma solução S.

Seja S uma solução viável com as propagandas $\mathcal{A}' \subseteq \mathcal{A}$ dispostas em *slots* $B_1, B_2, \ldots, B_K$. O *tipo* t de uma propaganda $A_i \in \mathcal{A}'$ em relação a S é o subconjunto de *slots* ao qual A_i é atribuído, ou seja, $A_i \in B_j$ se e somente se $j \in t$. Observe que duas propagandas do mesmo tipo possuem a mesma frequência, e assim pode-se pensar em todas as propagandas em $\mathcal{A}'$ do mesmo tipo como uma única propaganda. Seja $\mathcal{T}$ o conjunto de todos os subconjuntos de *slots*, então $\mathcal{T}$ contém todos os tipos possíveis e $|\mathcal{T}| = 2^K$.

Dada uma constante $\varepsilon > 0$ de forma que $1/\varepsilon$ seja um número inteiro, e seja $q = \min\{|\mathcal{A}|, 2^{2^K} 2^K/\varepsilon\}$. Nosso algoritmo "adivinha" um conjunto V com no máximo q pro-

pagandas com os maiores valores em uma solução ótima. Para cada $V \subseteq \mathcal{A}$ tal que $|V| \leq q$, definimos U como o conjunto de cada propaganda $A_i \in \mathcal{A} \setminus V$ tal que $v_i w_i \leq V_{min}$, onde $V_{min} = \min\{v_i w_i : A_i \in V\}$. Então, para cada configuração viável de $V \subseteq \mathcal{A}$, preenchemos os espaços restantes com propagandas de U usando programação linear. Em seguida, modificamos a solução ótima da programação linear para obter uma solução de mesmo valor e com no máximo uma propaganda alocada de forma fracionária por tipo. Removemos as propagandas alocadas de forma fracionária e mostramos que o valor perdido é pequeno o suficiente para se obter o fator de aproximação desejado.

Uma *configuração* para um subconjunto de propagandas $\mathcal{A}' \subseteq \mathcal{A}$ é uma solução viável que adiciona todas as propagandas em $\mathcal{A}'$. O Lema 2 afirma que se K for constante, então o número de configurações possíveis contendo apenas propagandas em V é polinomial no número de propagandas em V e pode ser enumerado por um algoritmo de força bruta.

Lema 2. *Se K for constante, então as configurações para todos os subconjuntos de V podem ser listadas em tempo polinomial.*

Demonstração. Existem $O(|\mathcal{A}|^q)$ escolhas possíveis para V, e existem $O(q^{2^K})$ soluções possíveis para cada conjunto, já que o número de tipos é 2^K. Assim, podemos enumerar V e todas as suas configurações no tempo $O(q^{2^K}|\mathcal{A}|^q)$, que é polinomial, pois K é constante e q é limitado por uma constante. □

Seja OPT_V uma solução ótima para as propagandas de V. Como, pelo Lema 2, todas as configurações candidatas podem ser listadas em tempo polinomial, podemos presumir que a configuração S_V encontrada pelo algoritmo é equivalente à solução induzida pelas propagandas de OPT_V. Ficamos com o problema residual de colocar propagandas de U na solução. Para cada *slot* j em S_V, $1 \leq j \leq K$, o espaço que não é utilizado pelas propagandas mais valiosas V é

$$u_j = 1 - \sum_{A_i \in B_j} s_i.$$

Definimos Residual-Maxspace-rdv como o problema de, dado um conjunto de propagandas U, onde $v_i w_i \leq V_{min}$ para todos $A_i \in U$, encontrar um subconjunto $\mathcal{A}'_t \subseteq U$ para cada $t \in \mathcal{T}$ tal que a ocupação de cada *slot* j seja no máximo u_j, e que maximize o valor

$$\sum_{t \in \mathcal{T}} \sum_{A_i \in \mathcal{A}'_t} |t| v_i.$$

Seja $\mathcal{T}(A_i)$ o subconjunto de tipos *compatíveis* com uma propaganda A_i, ou seja, o conjunto de tipos $t \in \mathcal{T}$ com $|t| = w_i$, e de forma que os *slots* em t respeitem o *release date* r_i e o *deadline* d_i. Resolvemos o programa linear (P) para atribuir propagandas de U

aos tipos.

$$\text{(P) Maximizar} \sum_{A_i \in U} \sum_{t \in \mathcal{T}(A_i)} v_i w_i F_{A_i,t} \tag{3.1}$$

$$\text{Sujeito a:} \sum_{t \in \mathcal{T}(A_i)} F_{A_i,t} \leq 1 \qquad \forall A_i \in U \tag{3.2}$$

$$\sum_{\substack{t \in \mathcal{T}: \\ j \in t}} \sum_{A_i \in U} s_i F_{A_i,t} \leq u_j \qquad j = 1, 2, \ldots, K \tag{3.3}$$

$$F_{A_i,t} \geq 0 \qquad \forall A_i \in U, \forall t \in \mathcal{T}(A_i) \tag{3.4}$$

As variáveis $F_{A_i,t}$ indicam se propaganda A_i está atribuída ao tipo t, as restrições (3.2) garantem que uma propaganda não pode ser atribuída a mais de um tipo e as restrições (3.3) garantem que a capacidade residual de qualquer *slot* não será violada.

Considere uma solução F para (P), que pode ser obtida em tempo polinomial [52], e observe que F induz uma atribuição fracionária de propagandas a tipos. Se a solução for tal que $F_{A_i,t} < 1$ unidades da propaganda A_i são atribuídas ao tipo t, então dizemos que a propaganda A_i é atribuída fracionadamente a t por um valor de $F_{A_i,t}$. O conjunto de todos os tipos t para os quais $F_{A_i,t} > 0$ é chamado de *suporte* de A_i e é denotado por $Sup(A_i)$.

Para eliminar atribuições fracionárias, agrupamos propagandas com o mesmo suporte. Seja W um subconjunto de tipos e P_W o conjunto de propagandas A_i com $Sup(A_i) = W$. Em particular, cada propaganda $A_i \in P_W$ é compatível com qualquer tipo $t \in W$. Para cada tipo $t \in W$, definimos o preenchimento total recebido por t de P_W como

$$z_t = \sum_{A_i \in P_W} s_i F_{A_i,t}.$$

Pelo fato de que cada propaganda em P_W é atribuída fracionadamente aos tipos em W, sabemos que

$$\sum_{A_i \in P_W} s_i \geq \sum_{A_i \in P_W} \sum_{t \in W} s_i F_{A_i,t} = \sum_{t \in W} z_t.$$

Em outras palavras, o tamanho total de P_W, dado por $\sum_{A_i \in P_W} s_i$, não é menor que o tamanho recebido por tipos W de P_W. Portanto, removemos a atribuição fracionária de todas as propagandas em P_W e reatribuímos integralmente cada propaganda em P_W aos tipos em W, descartando qualquer propaganda restante.

O processo de arredondamento da atribuição fracionária é resumido no Algoritmo 3, que recebe como entrada uma atribuição fracionária F de propagandas aos tipos W e retorna uma atribuição inteira F'.

Como parte do arredondamento, usamos um procedimento chamado Reassign, que pega um suporte W e as propagandas atribuídas a esse suporte P_W e retorna uma nova alocação dessas propagandas em W. Este procedimento remove todas as propagandas de P_W da solução fracionária e preenche de forma gulosa seu espaço com propagandas de P_W em ordem de eficiência (v_i/s_i). Note que este novo empacotamento não tem um valor pior já que todo o espaço está preenchido, as propagandas são escolhidas por

Algoritmo 2 Algoritmo para reatribuir uma solução fracionária.

procedimento REASSIGN(W, P_W)
  para cada $A_i \in P_W$ e $t \in W$ **faça**
    $F'_{A_i,t} \leftarrow 0$
  para cada $t \in W$ **faça**
    $z_t \leftarrow \sum_{A_i \in P_W} s_i F_{A_i,t}$
  $U'_W \leftarrow \emptyset$
  $z_W \leftarrow \sum_{t \in W} z_t$ ▷ Área total das propagandas com suporte W
  para cada $A_i \in P_W$ em ordem não-crescente de v_i/s_i **faça**
    se $\sum_{A_j \in U'_W} s_j < z_W$ **então** ▷ A última propaganda pode não caber integralmente
      $U'_W \leftarrow U'_W \cup \{A_i\}$
  seja $\{t_1, t_2, \ldots, t_{|W|}\}$ os tipos em W
  $k \leftarrow 1$
  para cada $A_i \in U'_W$ **faça**
    $s'_i \leftarrow s_i$
    enquanto $s'_i > 0$ **faça**
      $m \leftarrow \min\{s'_i, z_{t_k} - s'_i\}$
      $F'_{A_i,t_k} \leftarrow m/s_i$
      $s'_i \leftarrow s'_i - m$
      $z_{t_k} \leftarrow z_{t_k} - m$
      se $z_{t_k} = 0$ **então**
        $k \leftarrow k + 1$
  devolve F', U'_W

ordem de eficiência, e esta nova solução é fracionária. Sejam $U'_W \subseteq P_W$ as propagandas recém-empacotadas; observe que todas as propagandas U'_W, exceto talvez a última, estão totalmente empacotadas nos tipos W. O pseudocódigo REASSIGN é apresentado em Algoritmo 2.

No Lema 3, observamos que o Algoritmo 2 é polinomial no tamanho da instância. O Lema 4 mostra que REASSIGN não piora a solução.

Lema 3. *O Algoritmo 2 é executado em tempo polinomial.*

Demonstração. O tamanho de P_W é $O(n)$ e o tamanho de W é $O(2^K)$; portanto, o tempo de execução do Algoritmo 2 é $O(n2^K)$, que é polinomial, pois K é constante. □

Lema 4. *O Algoritmo 2 retorna uma solução com o mesmo valor da atribuição de programação linear (P).*

Demonstração. O algoritmo preenche o espaço das propagandas P_W nos tipos W de forma gulosa por eficiência, adicionando de forma integral cada propaganda, com exceção talvez da última, que preencherá o espaço restante. Seja F' o nome da alocação do algoritmo.

Suponha que F' não seja uma alocação ótima. Então existe uma alocação F^* que não foi feita por ordem de eficiência e que possui valor melhor que F'. Se F' aloca exatamente as mesmas propagandas de F^* na mesma proporção, então possuem o mesmo valor. Dessa

forma, as propagandas das alocações são diferentes ou a proporção das propagandas são diferentes.

No primeiro caso, existe uma propaganda j que está em F^* e não está em F'. Como j não foi usada em F', significa que existem propagandas com eficiência maior que a de j que foram usadas antes em F' e que não foram usadas totalmente em F^*. Podemos substituir todo o espaço de j em F^* por espaços dessas propagandas sem piorar a solução. Podemos repetir esse processo para toda propaganda que está em F^* e não está em F', de forma a obter a mesma alocação e mostrando que o valor da solução não piora. Logo, F' também é ótima.

Agora considere que as duas alocações possuem as mesmas propagandas, mas em proporções diferentes. Se todas as propagandas que aparecem em proporções diferentes nas alocações possuem a mesma eficiência, os valores delas para a solução são os mesmos e, portanto, as alocações possuem o mesmo valor. Assim, suponha que i e j são propagandas que aparecem em proporções diferentes em F' e F^* e a eficiência de i é maior que a de j. Como F' foi criada de forma gulosa em eficiência, a propaganda j será alocada em F' apenas quando toda a propaganda i estiver alocada, assim, sabemos que a propaganda i está completamente alocada em F' e está parcialmente alocada em F^*. Podemos remover espaço da propaganda j de F^* para adicionar mais de i. Pela eficiência das propagandas, essa mudança não piora o valor de F^* e podemos repeti-la para todas as propagandas com proporções diferentes, encontrando uma alocação igual a F' sem piorar o valor. E, novamente, mostrando que F' é ótima.

Desta forma, o algoritmo obtém um valor ótimo para a alocação fracionária das propagandas P_W no espaço considerado. O algoritmo de programação linear (P) também obtém uma solução ótima fracionária para as mesmas propagandas e considera o mesmo espaço. Portanto, ambas as soluções têm o mesmo valor. □

Algoritmo 3 Algoritmo para arredondamento de atribuição de propagandas.

1: **procedimento** ROUNDING(F)
2: **para cada** $A_i \in U$ e $t \in \mathcal{T}$ **faça**
3: $F'_{A_i,t} \leftarrow 0$
4: **para cada** $W \subseteq \mathcal{T}$ **faça**
5: $P_W \leftarrow$ todas as propagandas A_i com $Sup(A_i) = W$
6: $F', U'_W \leftarrow$ REASSIGN(W, P_W)
7: descarta de U'_W qualquer propaganda que não esteja totalmente atribuída ao mesmo tipo em W
8: **devolve** F'

O Lema 5 mostra que o Algoritmo 3 é polinomial no tamanho da instância. O Corolário 1 é obtido do Lema 6 e limita o valor total das propagandas descartadas pelo Algoritmo 3 em cada execução da Linha 7. E o Corolário 2 é obtido do Lema 6 e do Corolário 1.

Lema 5. *O Algoritmo 3 é executado em tempo polinomial no tamanho da instância.*

Demonstração. O laço da Linha 4 executa um número constante de iterações, já que $|\mathcal{T}| = 2^K$ e o número de subconjuntos de $\mathcal{T}$ é 2^{2^K}. O algoritmo REASSIGN também é polinomial, pelo Lema 3. Então, o algoritmo é executado em tempo polinomial. □

Lema 6. *Seja $W \subseteq \mathcal{T}$ e seja U'_W o conjunto de propagandas com suporte W após o algoritmo de reatribuição. O número de propagandas descartadas de U'_W na linha 7 do ROUNDING é no máximo $|W|$.*

Demonstração. O algoritmo REASSIGN adiciona uma propaganda fracionadamente a um tipo de duas maneiras: começando em um tipo t e continuando em um tipo $t+1$, e completando o preenchimento do último tipo de W. No primeiro caso, o algoritmo pode adicionar no máximo $|W| - 1$ propagandas fracionadas, sendo que no segundo caso é possível adicionar no máximo uma propaganda fracionada a um tipo. Assim, no máximo, $|W|$ propagandas são adicionadas fracionadamente aos tipos de W, e o resultado segue. □

Corolário 1. *Seja $W \subseteq \mathcal{T}$ e seja U'_W o conjunto de propagandas atribuídas a W após a reatribuição do algoritmo. Então o valor total das propagandas selecionadas em U'_W após a execução do arredondamento é*

$$\sum_{A_i \in U'_W} \sum_{t \in W} v_i w_i F'_{A_i,t} \geq \sum_{A_i \in U'_W} \sum_{t \in W} v_i w_i F_{A_i,t} - |W| V_{min}.$$

Demonstração. Seja U''_W o conjunto de propagandas descartadas de U'_W,

$$\begin{aligned}\sum_{A_i \in U'_W} \sum_{t \in W} v_i w_i F'_{A_i,t} &\geq \sum_{A_i \in U'_W} \sum_{t \in W} v_i w_i F_{A_i,t} - \sum_{A_i \in U''_W} v_i w_i \\ &\geq \sum_{A_i \in U'_W} \sum_{t \in W} v_i w_i F_{A_i,t} - |W| V_{min}.\end{aligned}$$

A segunda desigualdade é válida porque o número de propagandas descartadas em P_W na linha 7 é no máximo $|W|$ (Lema 6), e todas as propagandas $A_i \in U'_W$ tem valor $v_i w_i \leq V_{min}$, pela definição de U. □

Corolário 2. *A diferença entre os valores máximos da solução fracionária e modificada não é maior que $2^{2^K} 2^K V_{min}$. Isso é,*

$$\sum_{A_i \in \mathcal{A}} \sum_{t \in \mathcal{T}} v_i w_i F'_{A_i,t} \geq \sum_{A_i \in \mathcal{A}} \sum_{t \in \mathcal{T}} v_i w_i F_{A_i,t} - 2^{2^K} 2^K V_{min}.$$

Demonstração. Considere o valor das variáveis W e U'_W do Algoritmo 3. Usando o Coro-

lário 1, temos que

$$\begin{aligned}\sum_{A_i\in\mathcal{A}}\sum_{t\in\mathcal{T}} v_i w_i F'_{A_i,t} &= \sum_{W\subseteq\mathcal{T}}\sum_{A_i\in U'_W}\sum_{t\in W} v_i w_i F'_{A_i,t} \\ &\geq \sum_{W\subseteq\mathcal{T}}\left(\sum_{A_i\in U'_W}\sum_{t\in W} v_i w_i F_{A_i,t} - |W|V_{min}\right) \\ &\geq \sum_{A_i\in\mathcal{A}}\sum_{t\in\mathcal{T}} v_i w_i F_{A_i,t} - \sum_{W\subseteq\mathcal{T}} 2^K V_{min} \\ &= \sum_{A_i\in\mathcal{A}}\sum_{t\in\mathcal{T}} v_i w_i F_{A_i,t} - 2^{2^K} 2^K V_{min},\end{aligned}$$

onde a última desigualdade é válida porque $|W| \leq 2^K$, e a última igualdade é válida porque existem $2^{|\mathcal{T}|} = 2^{2^K}$ escolhas distintas para W. □

O algoritmo completo para MAXSPACE-RDV é apresentado em Algoritmo 4. Dado o parâmetro $\varepsilon > 0$, este algoritmo recebe um conjunto de propagandas $\mathcal{A}$ como entrada. O algoritmo tenta adivinhar quais propagandas q são mais valiosas para uma solução ótima. Ele explora todas as combinações possíveis de subconjuntos $V \subseteq \mathcal{A}$ com no máximo q propagandas, e para cada empacotamento viável para V, tenta preencher os espaços restantes com um conjunto de propagandas menos valiosas que as de V, chamado de U. Nesta etapa, o algoritmo associa propagandas de U a tipos utilizando o programa linear (P). O Algoritmo ROUNDING transforma a atribuição fracionária F em uma atribuição inteira F'. Observe que esta atribuição pode ser facilmente convertida em uma atribuição de propagandas de U em uma solução S'. O algoritmo retorna a solução de valor máximo dentre as consideradas.

Algoritmo 4 Algoritmo para o MAXSPACE-RDV com K constante..

procedimento ALGRDV$_\varepsilon(\mathcal{A})$
  $q \leftarrow \min\{|\mathcal{A}|, 2^{2^K}2^K/\varepsilon\}$
  $S \leftarrow \emptyset$
  para cada $V \subseteq \mathcal{A}$ **tal que** $|V| \leq q$ **faça**
    para cada empacotamento viável S_V de V **faça**
      $V_{min} \leftarrow \min\{v_i w_i : A_i \in V\}$
      $U \leftarrow \{A_i \in \mathcal{A} \setminus V \mid v_i w_i \leq V_{min}\}$
      $F \leftarrow$ resolve LP (P) com as propagandas de U
      $F' \leftarrow$ ROUDING(F)
      Adiciona propagandas de U a S_U de acordo com F'
      $S' \leftarrow S_V \cup S_U$
      se $f(S') \geq f(S)$ **então**
        $S \leftarrow S'$
  devolve S

No Lema 7 e 8, provamos que o Algoritmo 4 é polinomial no tamanho da instância e retorna uma solução viável. No Teorema 1, provamos que o Algoritmo 4 é um PTAS para

MAXSPACE-RDV .

Lema 7. *O Algoritmo 4 é executado em tempo polinomial.*

Demonstração. O programa linear é resolvido em tempo polinomial no tamanho do modelo [52], e o modelo é polinomial no tamanho da instância, pois possui $O(|U| + K)$ restrições e $O(|U|2^{2^K})$ variáveis. O algoritmo ROUDING também é polinomial, pelo Lema 5. Os laços nas linhas 4 e 5 são polinomiais, pelo Lema 2. Então, Algoritmo 4 é polinomial no tamanho da instância. □

Lema 8. *O Algoritmo 4 retorna uma solução viável.*

Demonstração. Como cada configuração de propaganda em V é viável, S_V respeita as restrições de *release date* e *deadline*. A solução S_U também respeita as restrições de *release date* e *deadline*, e as restrições (3.3) garantem que esta solução respeite as capacidades dos *slots*. Assim, o algoritmo retorna uma solução viável. □

Teorema 1. *O Algoritmo 4 é um PTAS para MAXSPACE-RDV.*

Demonstração. Tentamos todas as atribuições para V com $|V| \leq q$. Assim, considere o momento em que a disposição de V é o mesmo das propagandas mais valiosas de $|V|$ em uma solução ótima OPT. Seja S_V a atribuição de propagandas de V na solução retornada S. Assim, $f(S_V) = f(\text{OPT}_V)$, onde OPT_V é a atribuição de V em OPT. Observe que, se $q = |\mathcal{A}|$ ou $|\text{OPT}| \leq q$, então $f(S) = f(S_V) = f(\text{OPT}_V) = f(\text{OPT})$ e o resultado segue. Agora, considere que $q = 2^{2^K}2^K/\varepsilon < |\mathcal{A}|$ e $|\text{OPT}| > q$.

Seja F a solução do programa linear (P) e F' a saída de ROUNDING. Defina

$$f(F) = \sum_{A_i \in \mathcal{A}} \sum_{t \in \mathcal{T}} v_i w_i F_{A_i,t} \quad \text{e} \quad f(F') = \sum_{A_i \in \mathcal{A}} \sum_{t \in \mathcal{T}} v_i w_i F'_{A_i,t}.$$

Seja OPT_U uma solução ótima para as propagandas em U nos espaços restantes de OPT_V. Observe que OPT_U induz uma solução viável com valor $f(\text{OPT}_U)$. Isso implica que $f(F) \geq f(\text{OPT}_U)$, já que F é uma solução fracionária ótima nos espaços restantes de S_V, que tem o mesmo preenchimento de OPT_V. Além disso, observe que $f(S) = f(F') + f(S_V)$, então, usando o Corolário 2, temos que

$$\begin{aligned}
f(S) &= f(F') + f(S_V) \\
&= f(F') + f(\text{OPT}_V) \\
&\geq f(F) - 2^{2^K}2^K V_{min} + f(\text{OPT}_V) \\
&\geq f(\text{OPT}_U) - 2^{2^K}2^K \frac{f(S_V)}{q} + f(\text{OPT}_V) \\
&= f(\text{OPT}_U) - 2^{2^K}2^K \frac{f(S_V)}{\frac{2^{2^K}2^K}{\varepsilon}} + f(\text{OPT}_V) \\
&= f(\text{OPT}_U) - \varepsilon f(S_V) + f(\text{OPT}_V) \\
&\geq f(\text{OPT}) - \varepsilon f(\text{OPT}).
\end{aligned}$$

onde a primeira desigualdade é válida pelo Corolário 2, a segunda desigualdade é válida porque $V_{min} \leq f(S_V)/q$ e a última desigualdade é válida porque $f(\text{OPT}) \geq f(S_V)$. Como o algoritmo retorna a melhor solução e considera a solução S, o resultado segue. □

Capítulo 4

Positional Knapsack Problem

Problemas de mochila são recorrentes no *layout* publicitário de *sites*, como Google e Mercado Livre, onde os anúncios são exibidos em um *banner*. Nesses casos, a probabilidade de receber cliques e, portanto, a receita esperada para a exibição de um anúncio varia de acordo com sua proximidade do topo [86, 87]. Isto leva ao problema de organizar esses anúncios no *banner* de forma semelhante ao Binary Knapsack Problem (KP), com a diferença de que o ganho de um anúncio varia de acordo com sua posição. Para modelar esta tarefa, introduzimos uma nova variante do KP, que denominamos Positional Knapsack Problem (PKP), em que o valor de um item varia com a posição em que ele é colocado.

Formalmente, uma instância do PKP é composta por uma mochila de capacidade L e um conjunto de itens $I = \{1, 2, \ldots, n\}$, onde cada item i tem valor $v_i > 0$, tamanho $s_i > 0$ e uma função $g_i(h_i)$, onde h_i é a posição em que o item i é colocado na mochila e corresponde à soma dos tamanhos dos itens empacotados antes de i na mochila. O ganho de um item i é dado por $G_i = g_i(h_i)v_i$. O objetivo é encontrar um subconjunto $S \subseteq I$ e as posições para empacotar S que maximize o ganho total e não exceda a capacidade da mochila. Veja um exemplo na Figura 4.1, e observe que o topo da mochila corresponde à posição 0. Além disso, observe que, quando g_i é uma função constante para cada item i, temos o KP.

Ibarra e Kim [49] e Lawler [61] propuseram FPTASes para o KP. Várias outras abordagens também foram consideradas para lidar com o KP, que incluem programação dinâmica [2, 47, 80] e outros algoritmos exatos [5, 42, 47, 57, 65, 69, 88]. Referenciamos ao leitor a pesquisa em duas partes [10, 11] recentemente apresentada por Cacchiani et al. sobre o KP e suas variantes, onde algoritmos exatos e de aproximação são discutidos.

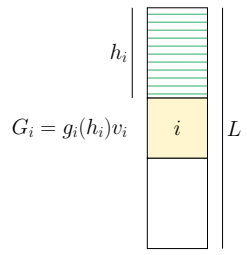

Figura 4.1: Ganho G_i de um item i quando empacotado na posição h_i de uma mochila.

Semelhante ao PKP, Gawiejnowicz et al. [36] consideram problemas de mochila com pesos ou valores de itens variáveis. Nesses problemas, o peso ou o valor de um item depende do índice do seu índice na sequência de itens empacotados na mochila. Observe que esses problemas diferem do PKP pois, no PKP, os ganhos variam de acordo com a posição na mochila, e não com o índice na sequência de itens. Gawiejnowicz et al. [36] apresentaram FPTASes para os problemas propostos considerando funções monotônicas.

O PKP também está relacionado a problemas de escalonamento. Um desses problemas é o Minimum Weighted Completion Time (MWCT), onde queremos agendar um conjunto J de n tarefas com pesos w_j em m máquinas, que só podem processar uma tarefa por vez. O tempo para processar uma tarefa $j \in J$ na máquina i é dado por $p_{j,i}$. Denotamos por C_j o tempo de conclusão de j, que corresponde à soma dos tempos das tarefas anteriores a j na mesma máquina mais o tempo de execução de j. O MWCT visa escalonar todas as tarefas minimizando $\sum_{j \in J} w_j C_j$. Este problema é APX-difícil com máquinas não relacionadas e quando todas as tarefas têm um prazo comum [46]. Isto é semelhante ao PKP, exceto que existem múltiplas mochilas, os "tamanhos" (neste problema, chamados de tempo de processamento) dependem da máquina, e o custo (em vez de um ganho) é uma função linear do tempo de conclusão (que equivale à altura mais o tamanho do item). Para este problema, Im e Li [50] propôs uma 1,8786-aproximação, e Im e Shadloo [51] propuseram uma 1,488-aproximação.

Em geral, o PKP é semelhante a escalonar tarefas em uma única máquina com prazo comum e com ganho variando de acordo com a posição da tarefa (que deve ser maximizada). Alguns trabalhos abordam versões do MWCT com apenas uma máquina [14, 45, 54, 62, 77]. Em Chekuri e Motwani [14], foi apresentada uma 2-aproximação para a variante do MWCT com uma máquina e precedência entre tarefas; isto é, uma tarefa só pode ser escalonada após todas as tarefas que a precedem também terem sido escalonadas. Em Lawler e Moore [62] e Kellerer e Strusevich [54] foi considerada uma versão em que o objetivo é minimizar o atraso das tarefas, ou seja, o quanto o tempo de conclusão ultrapassou o prazo. Estes artigos propuseram um algoritmo pseudopolinomial e um FPTAS para este problema.

Neste trabalho iniciamos o estudo do PKP. Primeiro, consideramos uma função de ganho linear simples, onde $g_i(h_i) = L - h_i$ para cada item i. Denotamos este caso particular de PKP por PKP-ℓ. Mostramos que o problema é NP-difícil mesmo para este caso e apresentamos um FPTAS. Como uma solução de PKP inclui a posição em que os itens são empacotados, a função objetivo depende dos tamanhos e valores dos itens empacotados. Isto é diferente do KP, cuja função objetivo depende apenas dos valores dos itens empacotados. Para lidar com essa dificuldade, nosso FPTAS utiliza uma abordagem de arredondamento recursivo. Também apresentamos um PTAS considerando funções mais gerais, assumindo certas condições de monotonicidade e continuidade. Mais especificamente, consideramos funções $g_i(\cdot)$ não-crescentes e que para todo $0 < \delta < 1$, existe um $\varepsilon > 0$ tal que $g_i(h + \varepsilon) \geq (1 - \delta) g_i(h)$ para todo h.

4.1 NP-dificuldade

Nesta seção, mostramos que o PKP é NP-difícil até mesmo para o caso particular do PKP-ℓ. Quando todos os itens têm o mesmo valor, o PKP-ℓ pode ser resolvido em tempo polinomial escolhendo de forma gulosa os menores itens que cabem na mochila. Da mesma forma, quando todos os itens têm o mesmo tamanho, o problema também pode ser resolvido em tempo polinomial escolhendo de forma gulosa os itens de maior valor enquanto cabem na mochila. Definimos a *eficiência* de um item i como $e_i = v_i/s_i$. Esta seção mostra que o PKP-ℓ é NP-difícil, mesmo quando todos os itens possuem eficiência 1.

O Partition Problem (PP) é um problema NP-completo clássico que consiste em decidir se é possível particionar um conjunto de inteiros positivos em duas partes com a mesma soma. O Equal-Cardinality Partition Problem (ECP) é uma variante que também exige que ambas as partes da solução tenham o mesmo número de elementos. Formalmente, dado um conjunto de índices $I = \{1, 2, \ldots, 2m\}$ e um número inteiro positivo a_i para cada $i \in I$, queremos decidir se existe um subconjunto $S \subseteq I$ tal que $|S| = m$ e $\sum_{i \in S} a_i = \sum_{i \in I \setminus S} a_i = \sum_{i \in I} a_i/2$. Assim como o Partition Problem, ECP também é NP-completo [35].

Diversas variantes do KP são trivialmente NP-difíceis, pois generalizam ou codificam facilmente instâncias do Partition Problem diretamente. Problemas ainda mais sofisticados podem ser demonstrados como NP-difíceis, desde que a função objetivo dependa apenas do valor total dos itens empacotados [58]. Para o PKP-ℓ, entretanto, na função objetivo, o valor de um item depende da posição onde está empacotado (além de seu valor), e as reduções padrão não são aplicáveis.

Tanto o KP quanto o PKP-ℓ têm os mesmos conjuntos de entradas e soluções viáveis. Apesar dessa semelhança, os itens de uma solução ótima para o KP não formam necessariamente uma solução ótima para o PKP-ℓ na mesma entrada, mesmo que todos os itens tenham eficiência 1.

Para ver um exemplo, considere uma instância com os itens descritos na Tabela 4.1 e uma mochila de capacidade $L = 10$. Duas soluções ótimas para uma instância do KP com esses itens são $\text{KP}_1^* = \{I_1, I_3, I_6\}$ e $\text{KP}_2^* = \{I_2, I_4, I_5\}$. Os valores dessas soluções quando consideramos a função objetivo de PKP-ℓ são $5 \cdot 10 + 3 \cdot 5 + 2 \cdot 2 = 69$ e $4 \cdot 10 + 3 \cdot 6 + 3 \cdot 3 = 67$, respectivamente. No entanto, uma solução ótima para PKP-ℓ é PKP-$\ell^* = \{I_1, I_2\}$ e tem valor $5 \cdot 10 + 4 \cdot 5 = 70$. Observe que PKP-$\ell^*$ não é uma solução ótima para o KP já que $\sum_{i \in \text{PKP-}\ell^*} v_i = 9$ e $\sum_{i \in \text{KP}_1^*} v_i = \sum_{i \in \text{KP}_2^*} v_i = 10$. Além disso, observe que KP_1^* e KP_2^* são soluções para o Partition Problem nesta instância, e PKP-ℓ^* não é uma solução para o PP na mesma instância. Essas soluções são mostradas na Figura 4.2.

i	1	2	3	4	5	6
$s_i = v_i$	5	4	3	3	3	2

Tabela 4.1: Instância para o exemplo da Figura 4.2..

A seguir, mostramos como reduzir o ECP para o PKP-ℓ. Dada uma instância do ECP, criamos uma instância do PKP-ℓ como segue. Seja $A = \sum_{i \in I} a_i/2$ e $M = 8A^2$. Defina a capacidade da mochila como $L = mM + A$ e, para cada $i \in I$, crie um item associado cujo

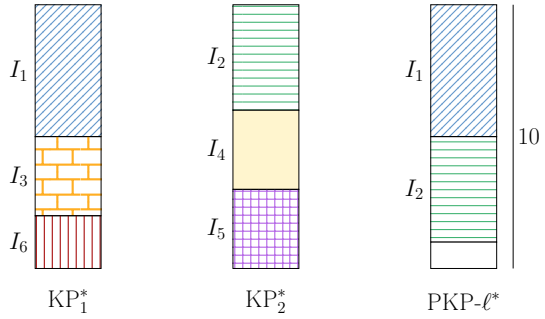

Figura 4.2: Exemplos de soluções para o KP e o PKP considerando os itens da Tabela 4.1.

tamanho e valor sejam iguais a $s_i = v_i = M + a_i$. A ideia por trás dessa redução é ter itens muito grandes, de modo que um número inteiro na instância original seja muito menor que o tamanho do item correspondente. Isto torna todos os itens muito semelhantes em tamanho, levando a algumas propriedades úteis fornecidas pelos lemas a seguir.

Seja $S \subseteq I$ uma solução para o PKP-ℓ considerando uma instância reduzida e suponha, sem perda de generalidade, que $S = \{1, 2, \ldots, r\}$. Além disso, defina $a(S) = \sum_{i \in S} a_i$ e $a^2(S) = \sum_{i \in S} a_i^2$. O valor de uma solução S é

$$
\begin{aligned}
f(S) &= \sum_{i=1}^{r} G_i = \sum_{i=1}^{r} s_i \left(L - \sum_{j=1}^{i-1} s_j \right) \\
&= L \sum_{i=1}^{r} s_i - \frac{1}{2} \sum_{i=1}^{r} \sum_{j=1}^{r} s_i s_j + \frac{1}{2} \sum_{i=1}^{r} s_i^2 \\
&= L \sum_{i=1}^{r} s_i - \frac{1}{2} \left(\sum_{i=1}^{r} s_i \right)^2 + \frac{1}{2} \sum_{i=1}^{r} s_i^2 \\
&= (mM + A) \sum_{i=1}^{r} (M + a_i) - \frac{1}{2} \left(\sum_{i=1}^{r} (M + a_i) \right)^2 + \frac{1}{2} \sum_{i=1}^{r} (M^2 + 2Ma_i + a_i^2) \\
&= (mM + A)\,(rM + a(S)) - \frac{1}{2}(rM + a(S))^2 + \frac{1}{2}(rM^2 + 2Ma(S) + a^2(S)) \\
&= \frac{1}{2} \left([rM(M - rM + 2mM + 2A)] + [2((1 - r + m)M + A)]a(S) - a(S)^2 + a^2(S) \right).
\end{aligned}
$$

No Lema 9, mostramos que uma solução ótima para a instância reduzida deve ter $m - 1$ ou m itens. No Lema 10, mostramos que qualquer solução com m itens tem um valor maior do que uma com $m - 1$ itens e, no Lema 11, mostramos que, dadas duas soluções que possuem m itens, aquela com maior preenchimento tem maior valor.

Lema 9. *Seja $S \subseteq I$ uma solução para o* PKP-ℓ *nessa redução, então $|S| \leq m$. Além disso, se S for ótimo, então $|S| \geq m - 1$.*

Demonstração. Suponha, por contradição, que $|S| > m$. Assim,

$$\sum_{i \in S} s_i > (m + 1)M > mM + A = L,$$

e S não seria uma solução viável.

Agora, suponha que $|S| \leq m-2$ e seja $s_{\max} = \max_{i \in I \setminus S} s_i$. Dessa forma,

$$\begin{aligned}\sum_{i \in S} s_i &\leq (m-2)M + \sum_{i \in S} a_i \leq mM - 2M + 2A \\ &= L - 2M + A = L - M - 8A^2 + A \\ &< L - M - A < L - s_{\max},\end{aligned}$$

e concluímos que poderíamos acrescentar um item sem estourar a capacidade da mochila e aumentar o valor de S. Portanto, S não é uma solução ótima. □

Lema 10. *Sejam $S' \subseteq I$ e $S'' \subseteq I$ soluções para o* PKP *neste caso. Se $|S'| = m$ e $|S''| = m-1$, então $f(S') > f(S'')$.*

Demonstração. Note que

$$\begin{aligned}f(S') &= \frac{1}{2}\Big(mM(M - mM + 2mM + 2A) + 2((1-m+m)M + A)a(S') - a(S')^2 + a^2(S')\Big) \\ &= \frac{1}{2}\Big(mM(M + mM + 2A) + 2(M + A)a(S') - a(S')^2 + a^2(S')\Big) \\ &> \frac{1}{2}\Big(mM(M + mM + 2A) + a(S')\ a(S') - a(S')^2 + a^2(S')\Big) \\ &> \frac{1}{2}\left(mM(M + mM + 2A)\right).\end{aligned}$$

A primeira desigualdade é válida porque $2(M + A) > a(S')$. Também note que

$$\begin{aligned}f(S'') &= \frac{1}{2}\Big((m-1)M(M - (m-1)M + 2mM + 2A) \\ &\quad + 2((1-(m-1)+m)M + A)a(S'') - a(S'')^2 + a^2(S'')\Big) \\ &= \frac{1}{2}\left((m-1)M(2M + mM + 2A) + (4M + 2A)a(S'') - a(S'')^2 + a^2(S'')\right) \\ &< \frac{1}{2}\left((m-1)M(2M + mM + 2A) + (4M + 2A)2A\right) \\ &= \frac{1}{2}\left((m-1)M(M + mM + 2A) + (m-1)M^2 + 8AM + 4A^2\right) \\ &< \frac{1}{2}\left((m-1)M(M + mM + 2A) + (m+1)M^2\right).\end{aligned}$$

A primeira desigualdade é válida porque $a(S'') < 2A$ e $a^2(S'') < a(S'')^2$, e a última desigualdade é válida porque $8h < M^2$ e $4A^2 < M^2$. A seguir, calculamos a diferença

$$\begin{aligned}f(S') - f(S'') &> \frac{1}{2}\left(M(M + mM + 2A) - (m+1)M^2\right) \\ &> \frac{1}{2}\left((m+1)M^2 - (m+1)M^2\right) = 0.\end{aligned}$$

Portanto, de fato $f(S') > f(S'')$. □

Lema 11. *Sejam $S' \subseteq I$ e $S'' \subseteq I$ soluções para a instância do* PKP-ℓ *tais que* $|S'| = |S''| = m$. *Se* $\sum_{i \in S'} s_i > \sum_{i \in S''} s_i$, *então* $f(S') > f(S'')$.

Demonstração. Lembre-se que

$$f(S') = \frac{1}{2}\left(mM(M + mM + 2A) + 2(M + A)a(S') - a(S')^2 + a^2(S')\right).$$

Seja $\delta = a(S') - a(S'')$ e $\Delta = a^2(S') - a^2(S'')$. Observe que $\delta \leq A$, já que $a(S') \leq A$ como S' é uma solução com m itens e a capacidade da mochila é $mM + A$. Além disso, observe que $\Delta \geq -A^2 + 1$, já que $a^2(S'') \leq a(S'')^2 \leq A^2$. Assim,

$$\begin{aligned}
f(S') &= \frac{1}{2}\left(mM(M + mM + 2A) + 2(M + A)a(S') - a(S')^2 + a^2(S')\right) \\
&= \frac{1}{2}\left(mM(M + mM + 2A) + 2(M + A)(a(S'') + \delta) - (a(S'') + \delta)^2 + (a^2(S'') + \Delta)\right) \\
&= \frac{1}{2}\Big(mM(M + mM + 2A) + 2(M + A)(a(S'') + \delta) \\
&\quad - (a(S'')^2 + 2\delta a(S'') + \delta^2) + (a^2(S'') + \Delta)\Big) \\
&= f(S'') + \frac{1}{2}\left(2(M + A)\delta - 2\delta a(S'') - \delta^2 + \Delta\right) \\
&= f(S'') + \frac{1}{2}\left(2\delta(M + A - a(S'')) - \delta^2 + \Delta\right) \\
&\geq f(S'') + \frac{1}{2}\left(2\delta M - \delta^2 + \Delta\right) \\
&\geq f(S'') + \frac{1}{2}\left(2M - A^2 - A^2 + 1\right) \\
&> f(S'').
\end{aligned}$$

A primeira desigualdade é válida porque $A \geq a(S'')$, e a última desigualdade é válida porque $2M > 2A^2 + 1$. □

Com estes lemas é possível verificar que o resultado da instância do ECP é "sim" se e somente se qualquer solução ótima para a instância reduzida tiver m itens e preencher toda a mochila. Isso implica o Teorema 2.

Teorema 2. PKP-ℓ *é NP-difícil mesmo quando restrito a instâncias onde todos os itens têm eficiência* 1.

Demonstração. Na versão de decisão do PKP-ℓ, dada uma instância do PKP-ℓ e um parâmetro k, deseja-se decidir se existe uma solução com valor pelo menos k.

Considere uma instância do ECP com um conjunto de índices $I = \{1, \ldots, 2m\}$ e valores a_i para $i \in I$, e crie a instância correspondente de PKP-ℓ. Observe que, para alguma solução S de PKP-ℓ, podemos reescrever a função objetivo como $f(S) = g(|S|, a(S), a^2(S))$, onde g é uma função não decrescente em cada um de seus parâmetros. Defina $k = g(m, A, m(A/m)^2)$.

A seguir, mostramos que a instância do ECP é "sim" se e somente se a instância da versão de decisão do PKP-ℓ for "sim". De fato, se a instância do ECP for "sim", seja

S um conjunto de índices m tais que $\sum_{i \in S} a_i = A$. Então, $f(S) = g(|S|, a(S), a^2(S)) \geq g(m, A, m(A/m)^2) = k$. Para a outra direção, assuma que a versão de decisão do PKP-ℓ é "sim". Assim, existe uma solução ótima S^* com um valor de pelo menos k.

Primeiro, afirmamos que $|S^*| = m$. Senão, então $|S^*| = m-1$ pelo Lema 9. Considere um conjunto S com m novos itens (fictícios) associados a números $a_i = A/m$ e tamanho e valor iguais a $s_i = v_i = M + a_i$ para cada $i \in S$. Já que $|S| = m$ e $|S^*| = m - 1$, o Lema 10 implica que

$$f(S^*) < f(S) = g(|S|, a(S), a^2(S)) = g(m, A, m(A/m)^2) = k.$$

Isto é uma contradição; portanto, de fato $|S^*| = m$. Agora, observe que $a(S^*) = A$, pois caso contrário teríamos $a(S^*) < a(S)$ e, pelo Lema 11, $f(S^*) < f(S) = k$. Novamente, isto é uma contradição; portanto, $a(S^*) = A$. Juntas, essas afirmações implicam que S^* é um conjunto de índices m tais que $\sum_{i \in S^*} a_i = A$, portanto, a instância de ECP é "sim". □

4.2 O PKP-ℓ

Nesta seção, consideramos o PKP-ℓ e apresentamos um algoritmo de tempo pseudo-polinomial e um FPTAS para ele. Lembre-se que, no PKP-ℓ, $g_i(h) = (L - h)v_i$ para cada item i, e a *eficiência* de um item i é dada por $e_i = v_i/s_i$. Nos Lemas 12 e 13, observamos que uma solução do PKP-ℓ pode ser representada apenas pelo conjunto de itens empacotados, pois mostramos que ordenar os itens em ordem não-crescente de eficiência maximiza o ganho total. Entretanto, observe que isso não significa que uma solução ótima para KP também seja uma solução ótima para PKP-ℓ (como mostrado no exemplo da Figura 4.2).

Lema 12. *Sejam i e j itens adjacentes em uma solução tal que $h_i < h_j$. Se $e_i < e_j$, então, trocar a ordem de i e j melhora o valor da solução para o* PKP-ℓ.

Demonstração. Primeiro, observe que, como $e_i < e_j$, temos que $v_i/s_i < v_j/s_j$ e, portanto, $s_j v_i < s_i v_j$. Além disso, observe que após a troca da ordem de i e j, o ganho dos itens colocados acima de i e abaixo de j permanecem inalterados. O ganho de i muda de $(L - h_i)v_i$ para $(L - h_i - s_j)v_i$ e o ganho de j muda de $(L - h_j)v_j$ para $(L - h_j + s_i)v_j$. A diferença entre os ganhos novos e anteriores é

$$(L - h_i - s_j)v_i - (L - h_i)v_i + (L - h_j + s_i)v_j - (L - h_j)v_j = -s_j v_i + s_i v_j > 0.$$

Assim, o valor da solução aumenta. □

Lema 13. *Seja S uma solução para o* PKP-ℓ *que empacota itens $I' \subseteq I$ em uma ordem ótima. Então, a ordem de empacotamento é não-crescente em termos de eficiência.*

Demonstração. Suponha, por contradição, que S empacota I' em uma ordem que não é não-crescente em relação à eficiência. Assim, existem itens adjacentes i e j com $h_i < h_j$ e $e_i < e_j$. Pelo Lema 12, podemos trocar a ordem de i e j e melhorar o valor da solução, o que contradiz a suposição de que S é ótima para I'. Assim, S empacota I' em ordem não-crescente de eficiência. □

Observe que quando a eficiência de todo item é igual a 1, ou seja, quando $v_i = s_i$ para todo item i, o valor da solução é o mesmo para qualquer ordem.

4.2.1 Um algoritmo de tempo pseudo-polinomial para o PKP-ℓ

Apresentamos um algoritmo pseudo-polinomial para o PKP-ℓ semelhante ao algoritmo de programação dinâmica apresentado por Ibarra e Kim [49] para o KP. Seja $V_{max} = \max_{i=1}^{n} v_i$ o maior valor de um item e lembre-se que $g_i(h) = (L-h)v_i$. Definimos o *pre-enchimento* de uma solução $S \subseteq I$ como a soma $\sum_{i \in S} s_i$.

Assumimos que os itens são ordenados em ordem não-crescente de eficiência. O algoritmo cria uma matriz A com dimensões $n \times nLV_{max}$ onde cada entrada $A(i,j)$ da matriz corresponde ao preenchimento mínimo de uma solução com ganho total exatamente j e considerando apenas os primeiros i itens. Se não existir tal solução, então $A(i,j) = \infty$. A seguinte recorrência calcula cada entrada da matriz:

$$A(i,j) = \begin{cases} \infty, & \text{se } i = 1 \text{ e } Lv_1 \neq j, \\ s_1, & \text{se } i = 1 \text{ e } Lv_1 = j, \\ \min(A(i-1,j),\ \min_k \left(A(i-1,k) + s_i\right)), & \text{caso contrário,} \end{cases}$$

onde o segundo mínimo no terceiro caso percorre cada número inteiro k tal que $0 \leq k < j$ e $k + (L - A(i-1,k))\, v_i = j$.

Podemos mostrar que esta recorrência é ótima para o PKP-ℓ. O tempo de execução do algoritmo é $\mathcal{O}(L^2 V_{max}^2 n^2)$, que é pseudo-polinomial no tamanho da instância. Embora este algoritmo não seja polinomial, usamos a recorrência como base para o FPTAS apresentado na Seção 4.2.

Observe que, se $L \geq \sum_{i \in I} s_i$, podemos empacotar todos os itens em ordem não-crescente de eficiência para obter uma solução ótima. Assim, podemos assumir que $L < \sum_{i \in I} s_i$. Portanto, se todos os valores e tamanhos dos itens forem limitados polinomialmente, esta recorrência resolve o PKP-ℓ em tempo polinomial.

4.2.2 Um FPTAS para o PKP-ℓ

Lembre-se que o Lema 13 afirma que os itens de uma solução ótima para o PKP-ℓ são empacotados em uma ordem não-crescente de eficiência. Assim, como antes, assumimos que os índices são ordenados nessa ordem.

Considere uma constante ε tal que $0 < \varepsilon < 1$, e defina $\delta = \varepsilon / n$. Além disso, para cada número $x \geq 0$, defina $R_\delta(x)$ como a maior potência inteira de $1+\delta$ que não é maior que x. Portanto, se $R_\delta(x) = (1+\delta)^j$ para algum j, então $(1+\delta)^j \leq x < (1+\delta)^{j+1}$. Dizemos que um valor x é *arredondado para baixo* para $R_\delta(x)$.

O algoritmo cria uma matriz A_δ com dimensões $n \times \left\lceil \log_{1+\delta}(nLV_{max}) \right\rceil$ tal que cada

entrada é calculada usando a seguinte recorrência:

$$A_\delta(i,j) = \begin{cases} \infty, & \text{se } i = 1 \text{ e } R_\delta(Lv_1) < (1+\delta)^j, \\ s_1, & \text{se } i = 1 \text{ e } R_\delta(Lv_1) \geq (1+\delta)^j, \\ \min(A_\delta(i-1,j),\ \min_k \left(A_\delta(i-1,k) + s_i\right)), & \text{caso contrário,} \end{cases}$$

onde o segundo mínimo no terceiro caso varia sobre cada número inteiro k tal que $0 \leq k < j$ e $R_\delta((L - A_\delta(i-1,k))\, v_i + (1+\delta)^k) \geq (1+\delta)^j$.

A ideia por trás dessa recorrência é usar um algoritmo de programação dinâmica semelhante ao apresentado na Subseção 4.2.1, mas aproximando o valor de uma solução arredondado para uma potência de $1+\delta$. Cada entrada $A_\delta(i,j)$ da matriz corresponde ao preenchimento de uma solução considerando apenas os primeiros i itens e com um ganho total arredondado de pelo menos $(1+\delta)^j$. Se não existe uma tal solução, definimos $A_\delta(i,j) = \infty$.

Seja S uma solução com m itens. Denotamos por S_i o i-ésimo item de S em ordem não crescente de eficiência. O ganho total de S é definido como $f(S) = \sum_{i \in S} G_i$. O *ganho total arredondado* de S é definido recursivamente como

$$f_\delta(S) = \begin{cases} 0, & \text{se } m = 0, \\ R_\delta(G_{S_m} + f_\delta(S \setminus \{S_m\})), & \text{se } m \geq 1. \end{cases}$$

Definimos $A^*_\delta(i,j)$ como o preenchimento mínimo de uma solução S considerando apenas os primeiros i itens e com ganho total arredondado $f_\delta(S) \geq (1+\delta)^j$. Se não existir tal solução, então $A^*_\delta(i,j) = \infty$. O Lema 14 prova que a recorrência calcula $A^*_\delta(i,j)$, ou seja, encontra a melhor solução se o ganho total for calculado por f_δ.

Lema 14. $A_\delta(i,j) = A^*_\delta(i,j)$.

Demonstração. Mostraremos o lema por indução em i. Se $i = 1$, uma solução S só pode incluir o item 1, portanto $f_\delta(S) = R_\delta(Lv_1)$. Nesse caso,

$$A_\delta\,(1,j) = \begin{cases} s_1, & \text{se } R_\delta(Lv_1) \geq (1+\delta)^j, \\ \infty, & \text{caso contrário.} \end{cases}$$

Assim, de fato, $A_\delta(1,j) = A^*_\delta(1,j)$.

Seja $i \geq 2$ e suponha que $A_\delta(i-1,k) = A^*_\delta(i-1,k)$, para cada $k < j$. Observe que, se $A_\delta(i,j) \neq \infty$, então podemos construir uma solução S considerando apenas os primeiros i itens com $f_\delta(S) \geq (1+\delta)^j$ por indução. E, então, $A_\delta(i,j) \geq A^*_\delta(i,j)$. A partir de agora, provaremos que $A_\delta(i,j) \leq A^*_\delta(i,j)$.

Se $A^*_\delta(i,j) = \infty$, então o resultado segue. Então, suponha que $A^*_\delta(i,j) \neq \infty$ e que existe uma solução S^* considerando apenas os primeiros i itens com $f_\delta(S^*) \geq (1+\delta)^j$.

Primeiro, suponha que o item i não esteja em S^*. Isso implica que $A^*_\delta(i-1,j) = A^*_\delta(i,j)$, e então $A_\delta(i-1,j) = A^*_\delta(i,j)$ já que $A_\delta(i-1,j) = A^*_\delta(i-1,j)$ pela hipótese de indução. Mas $A_\delta(i,j) \leq A_\delta(i-1,j)$, portanto temos que $A_\delta(i,j) \leq A^*_\delta(i,j)$.

Agora, suponha que o item i esteja em S^*, e seja G_i o ganho (não arredondado) de i em relação à solução S^*. Além disso, seja k o número inteiro tal que $f_\delta(S^* \setminus \{i\}) = (1+\delta)^k$. Observe que $k < j$, caso contrário, $f_\delta(S^* \setminus \{i\}) \geq (1+\delta)^j$, e poderíamos remover i de S^*, mantendo o ganho total arredondado em pelo menos $(1+\delta)^j$ e diminuindo o preenchimento.

Observe que $S^* \setminus \{i\}$ é uma solução considerando apenas os primeiros $i-1$ itens com ganho total arredondado de pelo menos $(1+\delta)^k$ que tem preenchimento mínimo. Caso contrário, então pode-se diminuir o preenchimento de S^* substituindo $S^* \setminus \{i\}$ pelo conjunto de itens correspondentes a $A^*_\delta(i-1,k)$. Segue-se que $\sum_{t \in S^* \setminus \{i\}} s_t = A^*_\delta(i-1,k)$. Como os itens estão em ordem não crescente de eficiência, temos que

$$G_i = \left(L - \textstyle\sum_{t \in S^* \setminus \{i\}} s_t\right) v_i = \left(L - A^*_\delta(i-1,k)\right) v_i.$$

Pela hipótese de indução, $A_\delta(i-1,k) = A^*_\delta(i-1,k)$, então

$$G_i = \left(L - A_\delta(i-1,k)\right) v_i.$$

Assim,

$$\begin{aligned} f_\delta(S^*) &= R_\delta(G_i + f_\delta(S^* \setminus \{i\})) \\ &= R_\delta(\left(L - A_\delta(i-1,k)\right) v_i + (1+\delta)^k) v_i \\ &\geq (1+\delta)^j. \end{aligned}$$

Concluímos que a entrada $A_\delta(i-1,k)$ é considerada na recorrência ao calcular $A_\delta(i,j)$, e assim

$$A_\delta(i,j) \leq A_\delta(i-1,k) + s_i = A^*_\delta(i-1,k) + s_i = A^*_\delta(i,k). \qquad \square$$

Pelo Lema 14, para cada entrada $A_\delta(i,j)$ da matriz, o algoritmo pode calcular uma solução correspondente S considerando apenas os primeiros i itens com ganho total arredondado $f_\delta(S) \geq (1+\delta)^j$ que minimiza o preenchimento. Seja S a solução calculada pelo algoritmo correspondente à entrada da matriz $A_\delta(n,j)$ tal que $A_\delta(n,j) \leq L$ e j seja máximo. Dizemos que S é a solução produzida pelo algoritmo. Usando o lema anterior, pode-se comparar o valor desta solução com o valor ótimo para o PKP-ℓ.

Lema 15. *Seja S^* uma solução ótima para o* PKP-ℓ *e S a solução gerada pelo algoritmo. Então, $f_\delta(S) \geq \exp(-2\varepsilon) f(S^*)$.*

Demonstração. Lembre-se de que o valor arredondado de um número x é $R_\delta(x)$ e que $R_\delta(x) \leq x < (1+\delta)R_\delta(x)$, portanto, temos $R_\delta(x) > (1-\delta)x$.

Sem perda de generalidade, assuma que $S^* = \{1, 2, \ldots, m\}$. Mostraremos por indução em m que

$$f_\delta(S^*) \geq \sum_{i=1}^{m} (1-\delta)^i G_{m-i+1}.$$

Para $m = 1$, temos que

$$f_\delta(S^*) = R_\delta(G_1 + f_\delta(S^* \setminus \{1\})) = R_\delta(G_1) \geq (1-\delta)G_1,$$

e a premissa é válida. Assim, seja $m \geq 2$ e assuma que a afirmação vale para $m-1$. Temos que

$$\begin{aligned}
f_\delta(S^*) &= R_\delta(G_m + f_\delta(S^* \setminus \{m\})) \\
&\geq (1-\delta)(G_m + f_\delta(S^* \setminus \{m\})) \\
&\geq (1-\delta)\left(G_m + \sum_{i=1}^{m-1}(1-\delta)^i G_{(m-1)-i+1}\right) \\
&= (1-\delta)G_m + (1-\delta)\sum_{i=2}^{m}(1-\delta)^{i-1}G_{m-i+1} \\
&= \sum_{i=1}^{m}(1-\delta)^i G_{m-i+1}.
\end{aligned}$$

Agora, como $n \geq m$,

$$\begin{aligned}
f_\delta(S^*) &\geq \sum_{i=1}^{m}(1-\delta)^i G_{m-i+1} \\
&\geq (1-\delta)^n \sum_{i=1}^{m} G_{m-i+1} \\
&= (1-\delta)^n f(S^*).
\end{aligned}$$

Observe que $f_\delta(S) \geq f_\delta(S^*)$, porque S é ótimo em relação ao ganho total arredondado f_δ. Por isso,

$$f_\delta(S) \geq f_\delta(S^*) \geq (1-\delta)^n f(S^*).$$

Substituindo δ por $\frac{\varepsilon}{n}$, e como $n \geq 2$,

$$f_\delta(S) \geq \left(1 - \frac{\varepsilon}{n}\right)^n f(S^*) \geq \exp(-2\varepsilon) f(S^*). \qquad \square$$

Observe que o algoritmo executa em tempo polinomial.

Lema 16. *O algoritmo executa em tempo polinomial no tamanho da representação da instância e em* $1/\varepsilon$.

Demonstração. Ordenar os itens por eficiência leva tempo $\mathcal{O}(n \log n)$. O número de entradas na matriz é $\mathcal{O}(n \log_{1+\delta}(nLV_{max}))$ e cada entrada pode ser calculada no tempo

$\mathcal{O}(\log_{1+\delta}(nLV_{max}))$. Assim, como $\delta = \varepsilon/n$, a complexidade do algoritmo é

$$\begin{aligned}\mathcal{O}\left(n\log n + n\log^2_{1+\frac{\varepsilon}{n}}(nLV_{max})\right) &= \mathcal{O}\left(\frac{n\log^2(nLV_{max})}{\log^2\left(1+\frac{\varepsilon}{n}\right)}\right)\\ &= \mathcal{O}\left(\frac{n\log^2(nLV_{max})}{\left(\frac{\varepsilon}{2n}\right)^2}\right)\\ &= \mathcal{O}\left(\frac{n^3\log^2(nLV_{max})}{\varepsilon^2}\right).\end{aligned}$$

A segunda igualdade é válida porque $\log(1+x) \geq \frac{x}{2}$ para $x \in [0,1]$. Observe que $\frac{\varepsilon}{n} \in [0,1]$, já que $n \geq 1$ e $0 < \varepsilon < 1$. Assim, $\log(1+\frac{\varepsilon}{n}) \geq \frac{\varepsilon}{2n}$. □

Combinando os Lemas 15 e 16, mostramos que o algoritmo é um FPTAS para o PKP-ℓ.

Teorema 3. *Existe um FPTAS para o* PKP-ℓ.

Demonstração. Pelo Lema 16, o algoritmo roda em tempo polinomial no tamanho da representação da instância e em $1/\varepsilon$. Pelo Lema 15, a solução de saída S tem valor pelo menos $f(S) \geq f_\delta(S) \geq \exp(-2\varepsilon)\text{OPT}$, onde OPT é o valor de uma solução ótima. Agora observe que para cada ε' tal que $0 < \varepsilon' < 1/2$, podemos definir $\varepsilon = \frac{1}{2}\ln(1/(1-\varepsilon'))$, tal que

$$f(S) \geq \exp(-2\varepsilon)\text{OPT} = (1-\varepsilon')\text{OPT}.$$

Além disso, observe que o tempo de execução do algoritmo é polinomial em $1/\varepsilon'$ já que $0 < \varepsilon < 1$ e

$$\frac{1}{\varepsilon} = \frac{1}{\frac{1}{2}\ln\left(\frac{1}{1-\varepsilon'}\right)} = \frac{-2}{\ln(1-\varepsilon')} \leq \frac{2}{\varepsilon'} = \mathcal{O}(1/\varepsilon').$$ □

4.3 Versão mais geral do PKP

Esta seção considera variantes do PKP com funções de ganho mais gerais. Na Seção 4.3.1, descrevemos as suposições sobre as funções de ganho e damos alguns exemplos. Na Seção 4.3.2, apresentamos um PTAS para a variante geral.

4.3.1 Suposições Sobre as Funções de Ganho

Dizemos que uma função g é *contínua* se, para todo x_0 e para todo $\delta > 0$, existe um $\varepsilon > 0$ tal que $|g(x) - g(x_0)| < \delta$, para todo x tal que $|x - x_0| < \varepsilon$. Uma função g é dita *uniformemente contínua* se, para todo $\delta > 0$, existe um $\varepsilon > 0$ tal que $|g(x) - g(y)| < \delta$, para todo x e y tais que $|x - y| < \varepsilon$. Neste trabalho, assumimos que a função de ganho para cada item satisfaz uma noção de continuidade, conforme definido a seguir.

Definição 1. *Uma função não-crescente* $g\colon [0,L] \to \mathbb{R}_{\geq 0}$ *é* relativamente contínua *se, para todo* $0 < \delta < 1$*, existe* $0 < \varepsilon < 1$ *tal que* $g(h+\varepsilon) \geq (1-\delta)g(h)$ *para todo* $h \in [0,L]$.

Exemplos de funções com esta propriedade são funções não-crescentes uniformemente contínuas com valor mínimo um, conforme declarado no Teorema 4.

Teorema 4. *Se uma função* $g\colon [0, L] \to \mathbb{R}_{\geq 1}$ *é não-crescente e uniformemente contínua, então* g *é relativamente contínua.*

Demonstração. Como g é uniformemente contínua e não-crescente, para todo $\delta' > 0$, existe um $\varepsilon' > 0$ tal que $|g(x) - g(y)| < \delta'$, para todo x e y tais que $|x - y| < \varepsilon'$. Sejam ε e δ números tais que $\varepsilon' > \varepsilon > 0$ e $0 < \delta < \delta'$. Portanto, $g(h) - g(h + \varepsilon) \leq \delta$ para todo h, e

$$g(h + \varepsilon) \geq g(h) - \delta \geq g(h) - \delta g(h) = (1 - \delta)g(h).$$

A segunda desigualdade é válida porque $g(h) \geq 1$ para todo $h \in [0, L]$. □

As funções lineares estão entre os exemplos mais simples de funções uniformemente contínuas. Outros exemplos são as funções Lipschitz contínuas e as funções α-Hölder contínuas. Além disso, qualquer função contínua no intervalo $[0, L]$ também é uniformemente contínua [79], portanto, funções como $g'(h) = (L^2 - h^2) + 1$ ou $g''(h) = (\sqrt{L} - \sqrt{h}) + 1$ são exemplos de funções não-crescentes e uniformemente contínuas para as quais o Teorema 4 se mantém.

Em nosso algoritmo, assumimos que a função de ganho de cada item é um elemento de algum conjunto de funções relativamente contínuas $\mathcal{G}$. Mais fortemente, assumimos que as desigualdades da definição se mantêm simultaneamente.

Definição 2. *Um conjunto* $\mathcal{G}$ *de funções não crescentes* $g\colon [0, L] \to \mathbb{R}_{\geq 0}$ *é* relativamente contínuo *se, para todo* $0 < \delta < 1$*, existe* $0 < \varepsilon < 1$ *tal que* $g(h + \varepsilon) \geq (1 - \delta)g(h)$ *para todo* $g \in \mathcal{G}$ *e* $h \in [0, L]$*.*

Observe que as premissas são satisfeitas por qualquer conjunto finito de funções relativamente contínuas ou por qualquer união finita de conjuntos relativamente contínuos. Para cada conjunto fixo de funções $\mathcal{G}$, consideramos uma variante de PKP em que cada item $i \in I$ está associado a alguma função $g_i \in \mathcal{G}$, o ganho associado a i quando é adicionado na posição h_i é $G_i = g_i(h_i)v_i$. A representação da função ganho depende do conjunto $\mathcal{G}$. Por exemplo, se $\mathcal{G}$ for finito, então uma instância poderá conter, para cada item i, o índice correspondente à função g_i.

4.3.2 Um PTAS para PKP com Funções $\mathcal{G}$

A seguir, considere um conjunto fixo de funções $\mathcal{G}$ que é relativamente contínuo e considere uma instância da variante correspondente de PKP. Supomos, sem perda de generalidade, que a capacidade da mochila é $L = 1$.

Seja $\delta > 0$ alguma constante fixa. Nesta seção, o objetivo é encontrar uma solução cujo valor esteja em um fator de pelo menos $1 - \mathcal{O}(\delta)$ do valor ótimo. Como $\mathcal{G}$ é relativamente contínuo, existe uma constante $\varepsilon > 0$ com $\varepsilon \leq \delta$ tal que $1/\varepsilon$ é um número inteiro e, para cada item $i \in I$, temos $g_i(h + \varepsilon) \geq (1 - \delta)g_i(h)$ para todo $h \in [0, 1]$.

A seguir, denotamos por (S^*, h^*) alguma solução ótima fixa, de modo que cada item selecionado $i \in S^*$ tem posição inicial h_i^*. Suponha que $|S^*| \geq 1/\varepsilon^2$, caso contrário, poderíamos resolver a instância em tempo polinomial.

Nosso algoritmo é descrito a seguir. Consideramos o subconjunto de itens V^* de S^* contendo os itens $1/\varepsilon^2$ com os maiores ganhos, bem como todos os itens com tamanho pelo menos ε^2. Observe que, como uma solução pode ter no máximo $1/\varepsilon^2$ itens com tamanho pelo menos ε^2, a cardinalidade de V^* é no máximo $2/\varepsilon^2$, que é limitado por uma constante. Assim, o algoritmo pode "adivinhar" o conjunto V^* e a ordem em que esses itens são colocados em tempo polinomial. Para simplificar, suponha que $V^* = \{1, 2, \ldots, |V^*|\}$ e que os itens estejam posicionados nesta ordem, pois poderíamos alterar os índices se isso não fosse o caso.

Como as funções de ganho não são crescentes, o espaço entre itens consecutivos de V^* na solução ótima contém itens cujo tamanho é menor que ε^2. Dizemos que cada espaço corresponde a um *intervalo* da mochila. O intervalo imediatamente antes de um item $i \in V^*$ é identificado pelo índice $j = i$, e o último intervalo é identificado pelo índice $j = |V^*| + 1$. Além disso, seja $h^*_{|V^*|+1} = 1$, denotamos por c_j^* o tamanho do intervalo j, portanto, este intervalo termina na posição h_j^* e corresponde ao conjunto de posições $[h_j^* - c_j^*, h_j^*)$.

Observe que, como já adivinhamos a ordem de V^*, a posição final h_j^* de cada intervalo j poderia ser determinada se soubéssemos o vetor c^* com os comprimentos dos intervalos. Não podemos enumerar todos os vetores candidatos porque o número de possibilidades é exponencial. Em vez disso, para cada intervalo j, gostaríamos de encontrar um número c_j que seja múltiplo de ε, tal que c_j se aproxime de c_j^* com uma diferença absoluta de no máximo ε. Como a mochila tem capacidade unitária, mostramos que o número de vetores candidatos para c é limitado por uma constante.

Definimos um vetor de comprimentos c e um vetor correspondente de posições finais h modificando as posições dos itens em V^* iterativamente. Primeiro, defina c_1 como o menor múltiplo de ε tal que $c_1 \geq c_1^*$, e defina $h_1 = c_1$. Então, para cada intervalo $j \geq 2$ em ordem, defina c_j como o menor múltiplo não negativo de ε tal que $c_j \geq h_j^* - (h_{j-1} + s_{j-1})$ e defina $h_j = h_{j-1} + s_{j-1} + c_j$. Intuitivamente, pode-se pensar neste procedimento como "empurrar para baixo" os itens de V^*. O lema a seguir observa que as posições finais não mudam muito.

Lema 17. *Para cada intervalo j, temos $0 \leq h_j - h_j^* < \varepsilon$.*

Demonstração. A prova é por indução em j. A posição final do primeiro intervalo aumenta em no máximo ε; assim, a desigualdade vale para $j = 1$. Agora, seja $j \geq 2$, e suponha por indução que $0 \leq h_{j-1} - h_{j-1}^* < \varepsilon$. Se $c_j > 0$, o intervalo j tem comprimento diferente de zero e a posição final de j é aumentada em no máximo ε. Caso contrário, $c_j = 0$ e $h_j = h_{j-1} + s_{j-1}$. Como $h_j^* \geq h_{j-1}^* + s_{j-1}$ e $h_j \geq h_j^*$,

$$0 \leq h_j - h_j^* \leq (h_{j-1} + s_{j-1}) - (h_{j-1}^* + s_{j-1}) = h_{j-1} - h_{j-1}^* < \varepsilon. \qquad \square$$

Como corolário, os intervalos modificados têm a propriedade chave de que o comprimento cumulativo é pelo menos tão grande quanto o da solução ótima. Isto é formalizado no seguinte corolário.

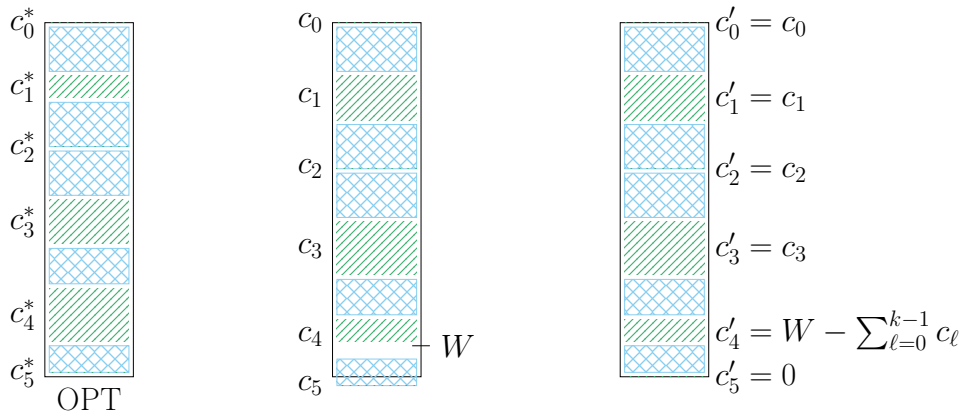

Figura 4.3: A primeira solução é um empacotamento ótimo OPT onde, em azul (linhas diagonais cruzadas) estão os itens de V^*, em verde (linhas diagonais paralelas) a área de intervalos entre eles em OPT, e c^* corresponde ao vetor de tamanhos dos intervalos. A segunda solução é um empacotamento para o mesmo conjunto de itens em OPT considerando os intervalos como um vetor c com múltiplos de ε. E a terceira solução é um empacotamento do mesmo conjunto de itens usando c', obtido modificando c de acordo com (4.1).

Corolário 3. *Para cada intervalo j, temos $\sum_{\ell=1}^{j} c_\ell \geq \sum_{\ell=1}^{j} c_\ell^*$.*

Demonstração. Por cálculo direto e usando o Lema 17,

$$\sum_{\ell=1}^{j} c_\ell = h_j - \sum_{\ell=1}^{j-1} s_\ell \geq h_j^* - \sum_{\ell=1}^{j-1} s_\ell = \sum_{\ell=1}^{j} c_\ell^*. \qquad \square$$

Como as posições podem aumentar, pode ocorrer que, na solução modificada, os últimos itens de V^* não caibam na mochila. Para garantir que todos os itens de V^* sejam empacotados, definimos um vetor ajustado de comprimentos de intervalo c'. Seja W o comprimento total dos intervalos na solução ótima, ou seja, $W = \sum_{j=1}^{|V^*|+1} c_j^*$. Consideramos cada intervalo j em ordem e definimos $c'_j = c_j$ até o intervalo k no qual o comprimento cumulativo é de pelo menos W, e definimos o comprimento dos próximos intervalos ajustados como zero. Mais precisamente, seja k o primeiro índice tal que $\sum_{\ell=1}^{k} c_\ell \geq W$. Para cada intervalo j, defina

$$c'_j = \begin{cases} c_j & \text{se } j < k, \\ W - \sum_{\ell=0}^{k-1} c_\ell, & \text{se } j = k, \\ 0 & \text{se } j > k. \end{cases} \tag{4.1}$$

Veja a Figura 4.3.

Seja h'_j a posição final do intervalo j na solução modificada correspondente aos comprimentos de intervalo ajustados c'. Como antes, as posições finais ajustadas não podem aumentar muito.

Lema 18. *Para cada intervalo j, temos $0 \leq h'_j - h_j^* < \varepsilon$.*

Demonstração. A prova é por indução em j. A posição final do primeiro intervalo aumenta no máximo ε; assim, a desigualdade vale para $j = 1$. Agora, seja $j \geq 2$, e suponha por indução que $0 \leq h'_{j-1} - h^*_{j-1} < \varepsilon$. Se $c_j > 0$, o intervalo j tem comprimento diferente de zero e a posição final de j é aumentada em no máximo ε. Caso contrário, $c_j = 0$ e

$h'_j = h'_{j-1} + s_{j-1}$. Como $h^*_j \geq h^*_{j-1} + s_{j-1}$ e $h'_j \geq h^*_j$,

$$0 \leq h'_j - h^*_j \leq (h'_{j-1} + s_{j-1}) - (h^*_{j-1} + s_{j-1}) = h'_{j-1} - h^*_{j-1} < \varepsilon. \qquad \square$$

Corolário 4. *Para cada intervalo j, temos $\sum_{\ell=1}^{j} c'_\ell \geq \sum_{\ell=1}^{j} c^*_\ell$.*

Demonstração. Por cálculo direto e usando o Lema 18,

$$\sum_{\ell=1}^{j} c'_\ell = h'_j - \sum_{\ell=1}^{j-1} s_\ell \geq h^*_j - \sum_{\ell=1}^{j-1} s_\ell = \sum_{\ell=1}^{j} c^*_\ell. \qquad \square$$

Agora, para algum subconjunto de itens S, defina $f(S)$ como o ganho total de itens em S quando cada item $i \in S$ é colocado na posição h^*_i, ou seja, defina $f(S) = \sum_{i \in S} g_i(h^*_i)v_i$. Analogamente, defina $f'(S) = \sum_{i \in S} g_i(h'_i)v_i$ como o ganho total de S conforme as posições h'. Além disso, seja $R^* = S^* \setminus V^*$. Assim, o valor da solução ótima é $f(S^*) = f(V^*) + f(R^*)$.

Como as posições iniciais dos itens em V^* mudam no máximo ε e as funções de ganho são relativamente contínuas, a perda no valor da solução é limitada por uma fração do valor ótimo.

Lema 19. $f'(V^*) \geq (1 - \delta)f(V^*)$.

Demonstração. Considere algum item $i \in V^*$. Pelo Lema 18, temos que $h'_i \leq h^*_i + \varepsilon$. Como g_i é relativamente contínua, $g_i(h'_i) \geq g_i(h^*_i + \varepsilon) \geq (1 - \delta)g_i(h^*_i)$. Portanto,

$$f'(S) = \sum_{i \in V^*} g_i(h'_i)v_i \geq \sum_{i \in V^*} (1 - \delta)g_i(h^*_i)v_i = (1 - \delta)f(V^*). \qquad \square$$

A seguir, abordamos os itens restantes. Seja U o conjunto de itens que não estão em V^* e cujo tamanho é menor que ε^2. Consideramos o problema residual de alocação de itens de U nos intervalos correspondentes a c'. Gostaríamos de encontrar uma alocação cujo ganho se aproximasse do ganho correspondente aos itens de R^* na solução ótima. Para fazer isso, construiremos e resolveremos uma instância do Generalized Assignment Problem (GAP).

Lembre-se que uma instância do GAP consiste em um conjunto de recipientes, chamados de *bins*, onde cada *bin* j tem capacidade t_j, e um conjunto de itens, onde cada item i tem valor p_{ij} e peso w_{ij} quando adicionado ao *bin* j. O objetivo é maximizar o valor dos itens empacotados sem ultrapassar a capacidade de nenhum *bin*. O GAP possui um PTAS para o caso em que o número de *bins* é limitado por uma constante [13, 33].

Construímos uma instância do GAP da seguinte maneira. Para cada intervalo não vazio j, crie $q_j = \lceil c'_j/\varepsilon \rceil$ *bins*, denotadas por $\{j^1, j^2, \ldots, j^{q_j}\}$. Cada *bin* tem capacidade ε, possivelmente com exceção da última, cuja capacidade é $c'_j - \varepsilon(q_j - 1)$. Observe que o conjunto de *bins* corresponde a uma partição dos intervalos em subintervalos. Como o comprimento de todos os intervalos, exceto um, é múltiplo de ε, no máximo um *bin* tem capacidade menor que ε, correspondendo ao último *bin* do último intervalo não vazio. Portanto, o número de *bins* é limitado por $1/\varepsilon$.

Seja G_{min} o ganho mínimo entre os itens de V^*. Para cada $i \in U$, consideramos um item correspondente da instância do GAP. O peso e o valor de um item i quando adicionado a um compartimento j^k são, respectivamente,

$$w_{ij^k} = s_i, \tag{4.2}$$

$$p_{ij^k} = \begin{cases} 0, & \text{se } g_i\big(h'_j - c'_j + k\varepsilon\big)v_i > G_{min}, \\ g_i\big(h'_j - c'_j + k\varepsilon\big)v_i, & \text{caso contrário.} \end{cases} \tag{4.3}$$

Observe que o ganho de um item i quando adicionado ao *bin* j^k é o ganho do item quando colocado na posição final do subintervalo correspondente. Observe também que o ganho de um item i quando adicionado a um *bin* j^k é $p_{ij^k} = 0$ se $g_i\big(h'_j - c'_j + k\varepsilon\big)v_i > G_{min}$. Isso significa que nenhum item pode valer mais que o pior item de V^*; caso contrário, esse item deverá ser um dos itens mais valiosos. Isso conclui a construção da instância. O lema a seguir fornece um limite inferior para o valor ótimo da instância do GAP.

Lema 20. *O valor ótimo da instância do* GAP *é pelo menos* $(1-\delta)(f(R^*) - \varepsilon f(V^*))$.

Demonstração. Construímos uma solução viável para a instância do GAP "movendo" iterativamente os itens de R^* na ordem em que eles aparecem na solução ótima (S^*, h^*). Primeiro, considere uma lista dos *bins* ordenados de acordo com os subintervalos correspondentes e tome o primeiro *bin* como atual. Para cada item $i \in R^*$, colocamos i na primeira posição livre do subintervalo correspondente ao *bin* atual se ele couber, e, caso contrário, descartamos i, fechamos o *bin* atual e definimos o próximo *bin* como atual.

Considere um item não descartado i. Seja j^k o compartimento ao qual i é atribuído e seja j^* o índice de intervalo contendo i na solução ótima (S^*, h^*). Seja d o tamanho total dos itens de R^* que precedem i na solução ótima. Como i está contido no intervalo j^* e $s_i > 0$, temos que $\sum_{\ell=1}^{j^*} c^*_\ell > d$. Além disso, o tamanho desses itens (incluindo os descartados) ocupa totalmente os *bins* fechados, portanto $\sum_{\ell=1}^{j-1} c'_\ell \leq d$. Combinando esta desigualdade com o Corolário 4, obtemos

$$\sum_{\ell=1}^{j^*} c^*_\ell > \sum_{\ell=1}^{j-1} c'_\ell \geq \sum_{\ell=1}^{j-1} c^*_\ell.$$

Mas então $j^* > j-1$, e $j^* \geq j$.

Seja h'_i a posição inicial de i após ele ser movido. Novamente, o tamanho total dos itens em R^* que precedem i é pelo menos $h'_i - \sum_{\ell=1}^{j-1} s_\ell$ e no máximo $h^*_i - \sum_{\ell=1}^{j^*-1} s_\ell$, portanto $h'_i - \sum_{\ell=1}^{j-1} s_\ell \leq h^*_i - \sum_{\ell=1}^{j^*-1} s_\ell$, e

$$h'_i \leq h^*_i + \sum_{\ell=1}^{j-1} s_\ell - \sum_{\ell=1}^{j^*-1} s_\ell \leq h^*_i.$$

A seguir, definimos o valor de um item não descartado da solução do GAP. Lembre-se que o valor de um item atribuído a um *bin* corresponde ao ganho desse item quando colocado no final do subintervalo correspondente a esse *bin*. Como um subintervalo

tem comprimento no máximo ε, cada item não descartado i tem valor de pelo menos $g_i(h'_i + \varepsilon)v_i \geq (1-\delta)g_i(h'_i)v_i \geq (1-\delta)g_i(h^*_i)v_i$.

Para contabilizar os itens descartados, lembre-se que V^* contém os $1/\varepsilon^2$ itens com os maiores ganhos na solução ótima (S^*, h^*) e que G_{min} é o ganho mínimo entre esses itens. Como existem pelo menos $1/\varepsilon^2$ itens em V^* com ganho de pelo menos G_{min}, sabemos que $1/\varepsilon^2\, G_{min} \leq f(V^*)$ e, portanto, $1/\varepsilon\, G_{min} \leq \varepsilon f(V^*)$. Lembre-se de que existem no máximo $1/\varepsilon$ *bins*; assim, no máximo $1/\varepsilon$ itens de R^* são descartados. Como cada item em R^* tem ganho no máximo G_{min}, o ganho total de itens não descartados é de pelo menos $f(R^*) - 1/\varepsilon\, G_{min} \geq f(R^*) - \varepsilon f(V^*)$. Como o valor da solução construída é pelo menos uma fração $(1-\delta)$ do ganho dos itens não descartados, o lema segue. □

Para cada combinação de conjuntos V' com até $2/\varepsilon^2$ itens, permutação de V' e vetor c, definimos h' adequadamente, construímos uma instância do GAP e encontramos uma $(1-\delta)$-aproximação para a instância usando algum PTAS conhecido para o GAP. A solução encontrada para esta instância induz um conjunto de itens R' a serem adicionados à mochila e um vetor de posições correspondente h. Fazemos a união $S' = V' \cup R'$ e verificamos se (S', h') é uma solução viável. O algoritmo gera a solução viável que maximiza o valor de $f'(S')$. Um pseudocódigo resumindo todo o PTAS é apresentado no Algoritmo 5.

Algoritmo 5 PTAS para PKP com um conjunto de funções $\mathcal{G}$..

procedimento PTAS-PKP-$\mathcal{G}_{\delta,\varepsilon}(I, s, v, g)$
  Seja $B = \{i \in I \mid s_i \geq \varepsilon^2\}$.
  para cada $I' \subseteq I$ tal que $|I'| \leq 1/\varepsilon^2$ **faça**
    para cada $B' \subseteq B$ tal que $|B'| \leq 1/\varepsilon^2$ **faça**
      Seja $V' = I' \cup B'$.
      para cada permutação de V' **faça**
        para cada vetor c com $|V'|+1$ múltiplos de ε t.q. $\sum_{\ell=1}^{|V'|+1} c_\ell \leq 1 - \sum_{j \in V'} s_j$ **faça**
          Calcule os comprimentos ajustados c' como em (4.1).
          Defina a posição h'_j de cada $j \in V'$ de acordo com c'.
          Para cada $j = 1, 2, \ldots, |V'|+1$, crie *bins* $\{j^1, j^2, \ldots, j^{q_j}\}$.
          Defina G_{min} como o menor ganho dentre os itens de V'.
          Seja $U = I \setminus V'$.
          Para cada $i \in U$ crie um item com peso e valor como em (4.2) e (4.3).
          Crie uma instância do GAP com itens e *bins* criados.
          Encontre uma $(1-\delta)$-aproximação para a instância do GAP.
          Sejam R' os itens retornados com posição h'_i para cada $i \in R'$.
          Seja $S' = V' \cup R'$ e calcule o valor da solução (S', h').
  devolve uma solução (S', h') com máximo $f(S')$.

Lema 21. *O Algoritmo 5 é executado em tempo polinomial.*

Demonstração. Como $|I'|$, $|B'|$ e $|V'|$ são limitados por uma constante, o número de escolhas e permutações de V' é polinomial. Além disso, o número de vetores distintos contendo no máximo $\mathcal{O}(1/\varepsilon^2)$ múltiplos de ε é uma constante. Para cada permutação fixa de V' e escolha de c, o tempo de execução para cada etapa é polinomial e, portanto, o algoritmo é polinomial. □

Teorema 5. *O Algoritmo 5 é um PTAS para o* PKP *com um conjunto de funções* $\mathcal{G}$.

Demonstração. Pelo Lema 21, o algoritmo executa em tempo polinomial no tamanho da instância.

Como o algoritmo tenta cada permutação possível de um conjunto com até $2/\varepsilon^2$ itens, e cada vetor possível c contendo $|V^*|+1$ múltiplos de ε, considera a permutação de V^* e o vetor c correspondente à solução ótima (S^*, h^*). Seja (S', h') a solução construída para essas escolhas. Além disso, lembre-se de que o valor desta solução é $f'(S') = f'(V') + f'(R')$, e o valor da solução ótima é $f(S^*) = f(V^*) + f(R^*)$.

Pelo Lema 19, $f'(V') = f'(V^*) \geq (1-\delta)f(V^*)$. Além disso, o valor $f'(R')$ é pelo menos o valor da solução construída para a instância do GAP. Como R' corresponde a uma $(1-\delta)$-aproximação para esta instância, o Lema 20 implica que $f'(R') \geq (1-\delta) \cdot (1-\delta)(f(R^*) - \varepsilon f(V^*))$. Combinando essas desigualdades, e lembrando que $\varepsilon \leq \delta$, temos:

$$\begin{aligned} f'(S') &\geq (1-\delta)f(V^*) + (1-\delta)\cdot(1-\delta)(f(R^*) - \varepsilon f(V^*)) \\ &\geq (1-2\delta)f(V^*) + (1-2\delta)f(R^*) - \delta f(V^*) \\ &\geq (1-3\delta)(f(V^*) + f(R^*)) \\ &= (1-3\delta)f(S^*). \end{aligned}$$

Como o algoritmo produz a melhor solução viável, isso completa o teorema. □

Capítulo 5

Extensible Bin Packing

O Extensible Bin Packing (EBP) é uma variante do Bin Packing Problem com número de *bins* fixo que permite que a capacidade dos *bins* seja estendida. Nele, são dados K *bins* com altura L, que podem ser estendidos, e um conjunto I de itens, em que cada item possui um tamanho $s_i \leq L$. Em uma solução, todos os itens devem ser alocados aos *bins* e o custo é dado pelo somatório dos custos dos *bins*. O custo de um *bin* é L mais o espaço excedido naquele *bin*. O objetivo do EBP é minimizar o custo da solução encontrada [16].

Este problema de empacotamento surge em uma variedade de problemas de escalonamento e alocação de armazenamento (*storage allocation*). Considere, por exemplo, o caso de um conjunto de tarefas que devem ser atribuídas a um conjunto de recursos idênticos (por exemplo, trabalhadores) que estão disponíveis a um custo fixo por um determinado período (o tempo regular) e podem ser adquiridos a um custo adicional, proporcional ao tempo (horas extras), em caso de necessidade. O problema de atribuir as tarefas aos trabalhadores visando minimizar o custo total é equivalente ao EBP. O Extensible Bin Packing também pode ser aplicado aos problemas de *storage allocation* nos quais uma capacidade extra pode ser obtida de um conjunto fixo de locais (*bins*) a um custo proporcional e desejamos minimizar o custo total [26].

Dell'Olmo et al. [26] mostraram que o algoritmo *Longest Processing Time* (LPT) é uma 13/12-aproximação para o EBP. O LPT, também conhecido como Algoritmo de Graham [40], consiste em ordenar os itens em ordem não-crescente de tamanho e inseri-los, nessa ordem, no *bin* mais vazio.

Posteriormente, Coffman Jr e Lueker [16] apresentaram um *Asymptotic Fully Polynomial Time Approximation Scheme* (AFPTAS) para o *Extensible Bin Packing* e mostraram que não existe um *Fully Polynomial Time Approximation Scheme* (FPTAS) para esse problema, a menos que P = NP.

Ye e Zhang [85] consideraram a versão online do Extensible Bin Packing com tamanhos de *bins* diferentes e apresentaram um limite justo de um algoritmo de escalonamento de lista para cada coleção de tamanhos de *bins* e cada número de *bins*.

O Extensible Bin Packing foi bastante explorado na literatura do ponto de vista de algoritmos de aproximação, mas não possui resultados utilizando métodos exatos. Métodos exatos já se mostraram bastante promissores em problemas semelhantes ao EBP, como o Bin Packing Problem. Por esses motivos, nesse trabalho, focamos no desen-

volvimento de algoritmos exatos para o EBP.

Um modelo trivial de Programação Linear Inteira para o EBP pode ser obtido a partir da formulação fraca apresentada por Martello e Toth [66] para o BPP. Esse modelo é apresentado a seguir e, durante os experimentos, foi comparado com os algoritmos propostos. A variável binária $x_{i,j}$ indica se o item i foi adicionado ao *bin* j e a variável inteira y_j indica em quanto a capacidade do *bin* j ultrapassou L.

$$\text{minimizar } KL + \sum_{j=1}^{K} y_j \tag{5.1}$$

$$\text{sujeito a: } \sum_{j=1}^{K} x_{i,j} = 1 \qquad \forall i \in I \tag{5.2}$$

$$\sum_{i \in I} s_i x_{i,j} \leq L + y_j \qquad j = 1, 2, \ldots, K \tag{5.3}$$

$$x_{i,j} \in \{0, 1\} \qquad \forall i \in I, j = 1, 2, \ldots, K \tag{5.4}$$

$$y_j \geq 0 \qquad j = 1, 2, \ldots, K \tag{5.5}$$

$$y_j \in \mathbb{R} \qquad j = 1, 2, \ldots, K \tag{5.6}$$

A função objetivo considera que o valor mínimo de uma solução é KL e que esse valor é acrescido do quanto cada *bin* estourou a capacidade L. As restrições (5.2) garantem que cada item foi empacotado exatamente uma vez e as restrições (5.3) garantem que o preenchimento de cada *bin* j é exatamente $L + y_j$.

O Lema 22 apresenta duas condições suficientes, mas não necessárias, para as quais sabemos que uma solução S é ótima. Essas condições serão usadas pelas heurísticas nos algoritmos para verificar se uma solução é ótima.

Lema 22. *Seja S uma solução para o* EBP *e seja $Pre(j)$ o preenchimento de um bin j. Se todo bin j de S possui preenchimento pelo menos L ou todo bin j de S possui preenchimento no máximo L, ou seja, para todo bin j de S, $Pre(j) \geq L$ ou, para todo bin j de S, $Pre(j) \leq L$, então S é uma solução ótima.*

Demonstração. Seja OPT uma solução ótima para o EBP e seja $f(X)$ o valor de uma solução X. Um limite inferior para $f(\text{OPT})$ é $\max\{KL, \sum_{i \in I} s_i\}$.

Considere o caso em que todo *bin* j de S possua preenchimento pelo menos L. Como todos os *bins* estão totalmente preenchidos, então $f(S) = \sum_{i \in I} s_i$, que é um limite inferior para o valor de uma solução do EBP. Portanto, S é ótima.

Agora considere que todo *bin* j de S possua preenchimento no máximo L. Nesse caso, $f(S) = KL$, que também é um limite inferior para o valor de uma solução do EBP. Portanto, S também é ótima nesse caso. □

A seguir, apresentamos um algoritmo baseado na formulação *arc-flow* e um algoritmo *branch-cut-and-price* para o Extensible Bin Packing.

5.1 Formulação *Arc-flow*

No modelo *arc-flow* apresentado a seguir, agrupamos os itens de mesmo tamanho e definimos como d_i a quantidade de itens de tamanho s_i.

Seja $W = \sum_{i \in I} s_i d_i$ a ocupação total dos itens, definimos um grafo $G = (V, A)$, tal que, $V = \{0, 1, \ldots, W\}$ e $A = \{(u, v) : \exists i \in I$ tal que $0 \leq u \leq v \leq W$ e $v - u = s_i\}$. Também adicionamos arcos de perda $(k, k+1)$ para todo $k \in V \setminus \{W\}$.

A seguir, apresentamos um modelo *arc-flow* para o EBP considerando o grafo descrito. A variável inteira $x_{u,v}$ indica quantas vezes o arco (u, v) foi selecionado e a variável inteira $y_{k,k+1}$ indica quantas vezes o arco de perda $(k, k+1)$ foi selecionado.

$$\text{(F) minimizar } KL + \sum_{\substack{(u,v)\in A \\ v>L}} (v - \max\{u, L\}) x_{u,v} \tag{5.7}$$

$$\text{sujeito a: } y_{0,1} + \sum_{(0,k)\in A} x_{0,k} = K \tag{5.8}$$

$$y_{v-1,v} - y_{v,v+1} + \sum_{(u,v)\in A} x_{u,v} - \sum_{(v,k)\in A} x_{v,k} = 0 \qquad v = 1, 2, \ldots, W-1 \tag{5.9}$$

$$y_{W-1,W} + \sum_{(u,W)\in A} x_{u,W} = K \tag{5.10}$$

$$\sum_{(k,k+s_i)\in A} x_{k,k+s_i} \geq d_i \qquad i = 1, 2, \ldots, n \tag{5.11}$$

$$y_{k,k+1} \geq y_{k-1,k} \qquad k = 2, \ldots, W-1 \tag{5.12}$$

$$x_{u,v} \geq 0 \qquad \forall (u, v) \in A \tag{5.13}$$

$$x_{u,v} \in \mathbb{Z} \qquad \forall (u, v) \in A \tag{5.14}$$

$$y_{k,k+1} \geq 0 \qquad k = 1, 2, \ldots, W-1 \tag{5.15}$$

$$y_{k,k+1} \in \mathbb{Z} \qquad k = 1, 2, \ldots, W-1 \tag{5.16}$$

A função objetivo considera que o valor mínimo de uma solução é KL e que esse valor é acrescido do quanto cada recipiente tem sua capacidade excedida. As restrições (5.8) e (5.10) garantem que exatamente K padrões serão selecionados. As restrições (5.9) garantem a conservação do fluxo, as restrições (5.11) garantem que pelo menos d_i itens de cada tamanho serão selecionados e as restrições (5.12) garantem que os arcos de perda são sempre adicionados no final do *bin*.

Note que o número de vértices de G pode ser muito grande em relação à quantidade de itens. Por isso, criamos um segundo modelo em que consideramos apenas itens que estão pelo menos parcialmente em algum *bin*. Itens que forem empacotados totalmente fora de um *bin* terão seus custos acrescidos integralmente ao custo final da solução, não importando em quais *bins* forem alocados.

Nesse segundo modelo, definimos $W = L + \max_{i \in I} s_i - 1$ e criamos o grafo $G = (V, A)$ como apresentado: $V = \{0, 1, \ldots, W\}$ e $A = \{(u, v) : 0 \leq u \leq v \leq W \text{ e } v - u = s_i, i \in I\}$. Novamente adicionamos arcos de perda $(k, k+1)$ para todo $k \in V \setminus \{W\}$. A variável inteira z_i indica quantas vezes um item de tamanho s_i foi empacotado totalmente fora de um *bin*.

$$\text{(F1) minimizar } KL + \sum_{\substack{(u,v) \in A \\ u < L \\ v > L}} (v - L) x_{u,v} + \sum_{i \in I} s_i z_i \tag{5.17}$$

$$\text{sujeito a: (5.8)} - \text{(5.10) e (5.12)} - \text{(5.16)} \tag{5.18}$$

$$\sum_{(k, k+s_i) \in A} x_{k,k+s_i} + z_i \geq d_i \qquad i = 1, 2, \ldots, n \tag{5.19}$$

$$z_i \in \mathbb{Z} \qquad \forall i \in I \tag{5.20}$$

$$z_i \geq 0 \qquad \forall i \in I \tag{5.21}$$

No modelo (F1), a função objetivo considera separadamente o valor dos itens que estão pelo menos parcialmente em algum *bin* e o valor dos itens que estão totalmente fora dos *bins*. Nas restrições (5.19), garantimos que as demandas dos itens sejam satisfeitas, considerando também os itens que estão totalmente foram dos *bins*.

Apesar da redução do grafo, o tamanho de W ainda pode ser consideravelmente grande em relação ao número de itens, pois basta considerar uma instância em que o maior item seja arbitrariamente grande. Com isso, propusemos o terceiro modelo que considera a técnica de *reflect*, proposta em Delorme e Iori [28] para o Bin Packing Problem e o Cutting Stock Problem, e posteriormente aplicada também ao Skiving Stock Problem por Martinovic et al. [67].

A ideia é que precisamos encontrar apenas metade de um padrão e *unir* com outra metade para obter um padrão inteiro. Para isso, definimos $W = \lceil L/2 \rceil$ e criamos um grafo $G = (V, A)$: $V = \{0, 1, \ldots, W\}$ e $A = \{(u, v) : 0 \leq u \leq v \leq W \text{ e } v - u = s_i, i \in I\}$. Novamente adicionamos arcos de perda $(k, k+1)$ para todo $k \in V \setminus \{W\}$. Além disso, adicionamos arcos de reflexão $R = \{(u, L - v) : u \leq W, v \geq W \text{ e } v - u = s_i, i \in I\} \cup \{(W, W)\}$ que serão utilizados para *unir* os padrões. Com isso, cada padrão será formado por dois *meio-padrões* e um arco de reflexão para uni-los.

Considere uma instância com 3 itens de tamanhos $(4, 3, 2)$ e *bin* de tamanho 10. No modelo *arc-flow* sem *reflect*, um padrão possível para esses itens seria o apresentado na Figura 5.1.

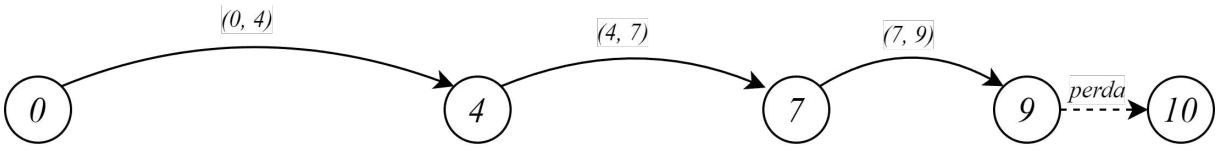

Figura 5.1: Exemplo de padrão *arc-flow* para 3 itens de tamanhos $(4, 3, 2)$ e *bin* de tamanho 10.

Considerando a técnica de *reflect*, podemos modelar o mesmo padrão da Figura 5.1 como dois padrões da metade do tamanho unidos por um arco de reflexão. Na Figura 5.2,

temos um padrão formado pelo item de tamanho 4 e um arco de perda e outro padrão formado pelos itens de tamanhos 3 e 2. Esses padrões são unidos pelo arco de reflexão $(5, 5, r)$. Na Figura 5.3, temos outra forma de representar o padrão da Figura 5.1 usando *reflect*. Observe que dessa vez o item de tamanho 2 é representado pelo arco de reflexão $(4, 4, r)$.

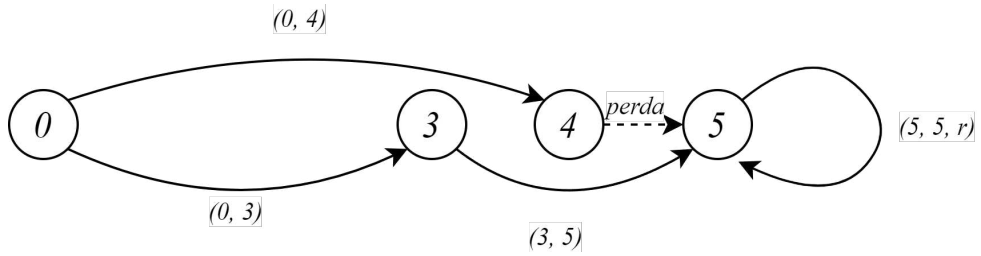

Figura 5.2: Exemplo de padrão *arc-flow* com *reflect* para 3 itens de tamanhos $(4, 3, 2)$ e *bin* de tamanho 10. O arco $(5, 5, r)$ é utilizado para unir as duas metades do padrão.

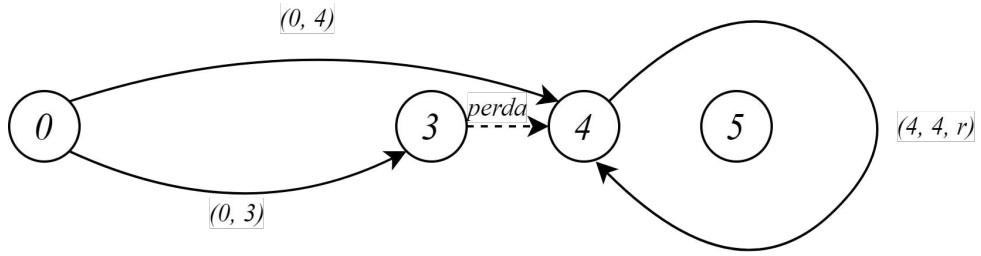

Figura 5.3: Exemplo de padrão *arc-flow* com *reflect* para 3 itens de tamanhos $(4, 3, 2)$ e *bin* de tamanho 10. O arco $(4, 4, r)$ é utilizado para unir as duas metades do padrão e também corresponde ao item de tamanho 2.

Note que agora todos os padrões completos (compostos pela junção de dois *meio-padrões*) precisam ter tamanhos no máximo L, o que impossibilita que itens sejam adicionados parcialmente em alguma *bin*. Para contornar isso, adicionamos itens *fictícios* de tamanhos $1, 2, \ldots, H$. Definimos H como o maior tamanho de item da solução, dado por $\max_{i \in I} s_i - 1$. Com essa escolha de H, desejamos limitar as possibilidades de tamanhos de itens *fictícios*, permitindo que sejam criados apenas itens que tenham tamanhos menores que o maior item da instância. Esses itens *fictícios* serão adicionados a um padrão para representar a parte que estaria dentro do *bin* de um item parcialmente adicionado. Veja o exemplo da Figura 5.4.

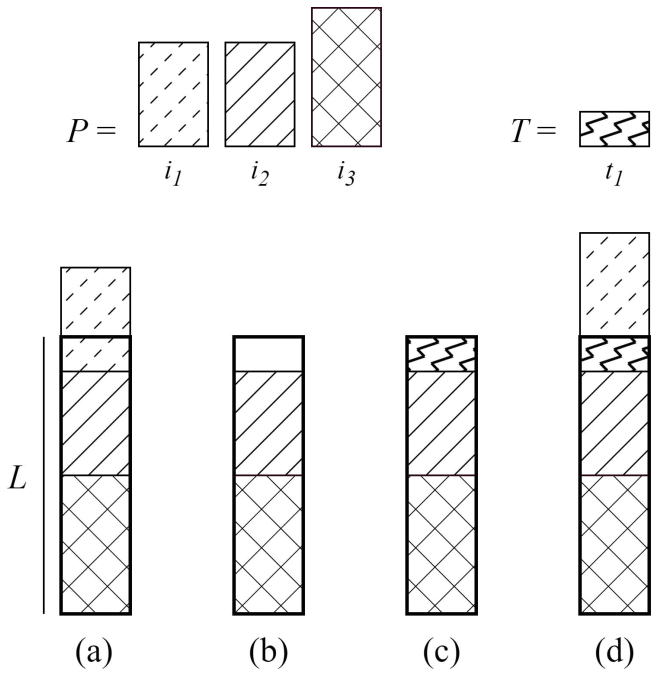

Figura 5.4: Exemplo da alocação dos itens de P considerando a adição de itens *fictícios* T.

A Figura 5.4 apresenta um exemplo de como funciona o preenchimento dos padrões com itens *fictícios*. Em (a) temos todos os itens do padrão P adicionados. Note que (a) não será possível no novo modelo, pois não permite que o *bin* seja estourado. Em (b) temos a remoção do item i_1. Observe que essa alocação possui valor L, ainda que não preencha todo o *bin*. Em (c) adicionamos o item t_1 para preencher o restante do *bin*. Na função objetivo, o item t_1 entra como um desconto no custo da solução, o que faz com que essa alocação tenha valor $L - s_{t_1}$. Em (d) temos a adição de i_1 à solução final. Note que o valor de (d) é igual ao valor de (a), pois o item t_1 tem custo negativo de valor igual ao espaço que seria ocupado por i_1 no *bin*.

A adição de itens *fictícios* permite que tratemos os itens reais apenas como os que estão totalmente em algum *bin* ou totalmente fora delas. Para representar os itens *fictícios*, adicionamos ao grafo arcos T para itens de tamanhos $1, 2, \ldots, H$, assim como os arcos para itens reais. Os arcos de T serão utilizados pelo modelo para preencher os padrões com itens *fictícios*, que serão descontados na função objetivo.

Seja $AT = A \cup T$ e seja $ATR = AT \cup R$, apresentamos o seguinte modelo de *arc-flow* com *reflect* para o EBP. A variável inteira z_i indica quantas vezes um item de tamanho s_i não foi empacotado. A variável inteira $x^o_{u,v}$ indica quanto de fluxo passa pelo arco (u, v) do tipo o. Os tipos de arcos são: a para arcos de itens da entrada, dado pelo conjunto A; r para arcos de reflexão, dado pelo conjunto R; e t para arcos de itens *fictícios*, dado pelo conjunto T. A variável inteira $y_{k,k+1}$ indica quanto de fluxo passa pelo arco de perda $(k, k+1)$.

$$\text{(F2) min. } KL - \sum_{(u,v)\in T} (v-u)x^t_{u,v} + \sum_{i\in I} z_i s_i \tag{5.22}$$

$$\text{suj. a: } \sum_{(u,v)\in R} x^r_{u,v} \leq K \tag{5.23}$$

$$y_{0,1} + \sum_{(0,k,o)\in AT} x^o_{0,k} = 2 \sum_{(u,v)\in R} x^r_{u,v} \tag{5.24}$$

$$y_{v-1,v} + \sum_{(k,v,t)\in AT} x^t_{k,v} = \sum_{(k,v)\in R} x^r_{k,v} + \sum_{(v,u,o)\in ATR} x^o_{v,u} + y_{v,v+1} \qquad v = 1,2,\ldots,W-1 \tag{5.25}$$

$$y_{W-1,W} + \sum_{(k,W,t)\in AT} x^t_{k,W} = \sum_{(k,W)\in R} x^r_{k,W} + \sum_{(W,u)\in R} x^r_{W,u} \tag{5.26}$$

$$\sum_{(k,k+s_i)\in A} x^a_{k,k+s_i} + \sum_{\substack{(u,v)\in R \\ 2W-v-u=s_i}} x^r_{u,v} + z_i \geq d_i \qquad i = 1,2,\ldots,n \tag{5.27}$$

$$\sum_{\substack{(u,v)\in T \\ v-u\geq l}} x^t_{u,v} \leq \sum_{\substack{i\in I \\ s_i > l}} z_i \qquad l = 1,2,\ldots,H \tag{5.28}$$

$$x^o_{u,v} \in \mathbb{Z} \qquad \forall (u,v,o) \in ATR \tag{5.29}$$

$$x^o_{u,v} \geq 0 \qquad \forall (u,v,o) \in ATR \tag{5.30}$$

$$y_{k,k+1} \in \mathbb{Z} \qquad k = 1,2,\ldots,W-1 \tag{5.31}$$

$$y_{k,k+1} \geq 0 \qquad k = 1,2,\ldots,W-1 \tag{5.32}$$

$$z_i \in \mathbb{Z} \qquad \forall i \in I \tag{5.33}$$

$$z_i \geq 0 \qquad \forall i \in I \tag{5.34}$$

A restrição (5.23) garante que serão selecionados no máximo K padrões, pois limita o número de arcos de reflexão utilizados em K e cada padrão deve utilizar exatamente um arco de reflexão. A restrição (5.24) garante que o fluxo que sai da origem (vértice 0) é exatamente o dobro do número de arcos de reflexão utilizados, ou seja, cada arco de reflexão unirá dois fluxos. As restrições (5.25) e (5.26) garantem a conservação do fluxo. As restrições (5.27) garantem que pelo menos d_i itens de cada tamanho serão selecionados. Essa restrição soma os itens fora dos *bins*, os arcos de tamanho s_i selecionados e os arcos de reflexão de tamanho s_i selecionados. E as restrições (5.28) garantem que itens *fictícios* serão selecionados apenas se existirem itens fora dos *bins* de tamanhos iguais ou superiores, que possam substituir tais itens *fictícios* na solução final. As restrições (5.28) são importantes porque, quando um item *fictício* de tamanho s_i é adicionado, ele tem seu tamanho descontado da solução objetivo e precisa que exista um item j com $s_j \geq s_i$ empacotado fora dos *bins* para compensar esse valor descontado.

Em todos os algoritmos de *arc-flow* usamos a técnica *Reduced Cost Variable Fixing* para tentar reduzir o número de arcos dos modelos. Essa técnica será apresentada na Seção 5.1.1.

5.1.1 Fixação de Variáveis por Custo Reduzido

A fixação de variáveis por custo reduzido (RCVF) é uma técnica amplamente aplicada na literatura, principalmente em modelos *arc-flow* [24, 28], para fixar variáveis referentes a arcos em 0 e diminuir o número de variáveis do modelo.

Para aplicar essa técnica, precisamos do valor $\bar{\lambda}_{\mathbb{Z}}$ de uma solução inteira para o modelo, os custo reduzidos $\bar{c}$ das variáveis obtidas de uma solução dual viável S e o valor $\bar{\pi}$ da solução S. Hadjar et al. [44] demonstraram que qualquer variável x tal que $\bar{\pi}+\bar{c}_x > \bar{\lambda}_{\mathbb{Z}}-1$, em que $\bar{c}_x$ é o custo reduzido de x em S, não pode pertencer a uma solução primal com valor menor que $\bar{\lambda}_{\mathbb{Z}}$. Ou seja, podemos fixar em 0 todos os arcos que satisfaçam essa condição.

Nos nossos algoritmos, antes de iniciar a resolução do modelo *arc-flow*, criamos uma solução inteira com o algoritmo de Graham e resolvemos a relaxação do modelo para encontrar o valor e os custos reduzidos de uma solução dual viável. Em seguida, aplicamos a técnica de RCVF para fixar arcos em 0 e resolvemos os modelos *arc-flow* após esse processo.

5.2 Branch-Cut-and-Price

Nesta seção, apresentamos um algoritmo *Branch-Cut-and-Price* para o Extensible Bin Packing. Neste algoritmo, devido às estratégias de ramificação utilizadas, consideramos cada item de forma independente, isto é, os itens não são agrupados com outros itens de mesmo tamanho.

Seja w_p o custo de um padrão p,

$$w_p = \begin{cases} L, & \sum_{i\in p} s_i \leq L, \\ \sum_{i\in p} s_i, & \textit{caso contrário}, \end{cases}$$

e seja a_{ip} a quantidade de vezes que um item i aparece no padrão p. A seguir, apresentamos o modelo primal utilizado no algoritmo.

$$\text{(F3) minimizar} \quad \sum_{p\in P} w_p x_p + \sum_{i=1}^{n} s_i z_i \tag{5.35}$$

$$\text{sujeito a:} \quad \sum_{p\in P} x_p \geq K \tag{5.36}$$

$$\sum_{p\in P} a_{ip} x_p + z_i \geq 1 \qquad i = 1, 2, \ldots, n \tag{5.37}$$

$$x_p \geq 0 \qquad \forall p \in P \tag{5.38}$$

$$z_i \geq 0 \qquad i = 1, 2, \ldots, n \tag{5.39}$$

$$x_p \in \mathbb{Z} \qquad \forall p \in P \tag{5.40}$$

$$z_i \in \mathbb{Z} \qquad i = 1, 2, \ldots, n \tag{5.41}$$

A variável inteira x_p indica quantas vezes o padrão p foi selecionado e a variável inteira z_i indica quantas vezes um item i aparece fora de um *bin*. As restrições (5.36) garantem que pelo menos um padrão para cada *bin* será selecionado e as restrições (5.37) garantem que pelo menos uma cópia de cada item será selecionada.

A seguir, apresentamos um modelo dual para o modelo apresentado anteriormente,

$$\text{(D) maximizar} \quad \sum_{i=1}^{n} \alpha_i + K\beta \tag{5.42}$$

$$\text{sujeito a:} \quad \sum_{i \in p} \alpha_i + \beta \leq w_p \qquad \forall p \in P \tag{5.43}$$

$$\alpha_i \leq s_i \qquad i = 1, 2, \ldots, n \tag{5.44}$$

$$\alpha_i \geq 0, \beta \geq 0 \tag{5.45}$$

As variáveis α se referem às restrições (5.37) e a variável β se refere à restrição (5.36) do modelo (F3). Utilizamos a restrição (5.43) para resolver o problema de precificação e adicionar novos padrões ao modelo. O algoritmo de *pricing* será apresentado na Seção 5.2.6. As estratégias de *branching* serão apresentadas na Seção 5.2.1 e os cortes adicionados são apresentados na Seção 5.2.3.

5.2.1 Estratégias de Ramificação

O algoritmo *Branch-Cut-and-Price* foi implementado usando o modelo primal (F3) apresentado na Seção 5.2. Utilizamos duas estratégias de *branching*. A primeira é semelhante a apresentada por Ceselli e Righini [12] para o Ordered Open-End Bin-Packing Problem. A cada nível da árvore, um item i é escolhido de acordo com quanto dele, na solução relaxada do nó atual, se deve a padrões em que i aparece totalmente dentro do *bin* ou a padrões em que i aparece estourando o *bin*. Especificamente, a cada nó, escolhemos o item i em que o somatório dos x_p referente aos padrões em que i aparece totalmente dentro do *bin* mais se aproxima de 0.5, e quando mais de um item ficam empatados, escolhemos o maior deles.

Nessa estratégia, fixamos se i deve aparecer sempre em um padrão ou se i deve sempre ser um item que estoura um padrão. No primeiro caso, removemos todos os padrões em que i aparece como item que estoura um padrão e permitimos adicionar apenas padrões em que o item i aparece totalmente dentro do *bin* ou não aparece. Já no segundo caso, removemos todos os padrões em que i aparece totalmente dentro do *bin* e permitimos adicionar apenas padrões em que i aparece como o item que estourou o *bin* ou não aparece. Quando K itens são fixados como o item que estoura um *bin*, o algoritmo começa a utilizar a segunda estratégia de *branching* nos nós filhos.

Na segunda estratégia utilizamos a ideia proposta por Ryan e Foster [74], que seleciona um par de itens para que, ou sejam empacotados no mesmo *bin*, ou não possam ser empacotados no mesmo *bin*. Em cada nó, escolhemos os dois itens a e b que maximizam $s_a + s_b$ e não aparecem integralmente juntos na solução ótima fracionária. Se tais itens não existirem, selecionamos a e b como os dois itens que maximizam $s_a + s_b$ e que não foram

ramificados juntos ainda. Com tais itens escolhidos, ramificamos considerando que a e b sempre aparecem juntos ou sempre aparecem separados. No primeiro caso, eliminamos do modelo todos os padrões em que a e b aparecem separados e no segundo caso eliminamos do modelo todos os padrões em que a e b aparecem juntos.

Note que essa estratégia de *branching* também deve ser refletida no *pricing* (Seção 5.2.6), uma vez que os padrões gerados em cada nó devem considerar que a e b podem estar juntos ou separados.

5.2.2 Heurísticas Primais

O *Relax-and-Fix* [7] foi a principal heurística primal utilizada no nosso algoritmo e é executada em cada nó do *branch-and-bound*. Com essa heurística fixamos variáveis de forma gulosa e resolvemos o modelo, a fim de encontrar novas variáveis a serem fixadas, até que uma solução seja obtida. No nosso algoritmo, a cada passo fixamos todas as variáveis x_i inteiras e quando não há nenhuma, fixamos a variável x_i de maior valor. Em seguida, resolvemos o modelo e adicionamos padrões violados novos com o algoritmo de *pricing*. Esse processo é repetido até que K variáveis sejam fixadas e uma solução seja obtida. Os padrões adicionados durante o *Relax-and-Fix* se mantém no conjunto de padrões após a execução da heurística.

A cada iteração do *Relax-and-Fix* (onde novas variáveis são fixadas), executamos também uma heurística baseada no algoritmo de Graham [40]. Na nossa heurística, consideramos a solução parcial das variáveis fixadas e adicionamos o restante dos itens usando o algoritmo de Graham.

5.2.3 Cortes

Usamos cortes *Subset-Row* (SR), que são um caso particular dos cortes de Chvátal-Gomory de Rank 1 [15, 39]. Esses cortes não são robustos, ou seja, a adição deles aumenta a complexidade do problema de *pricing*.

Seja I o conjunto de itens e $\mathcal{S} \subseteq \{S \subset I : |S| = 3\}$ o conjunto de triplas desses itens. Seja I_p os itens de um padrão p e seja $P(S) = \{p \in P : |I_p \cap S| \geq 2\}$ o conjunto de padrões que contém ao menos dois itens de S. Adicionamos a restrição a seguir para garantir que no máximo um padrão de $P(S)$ seja escolhido por vez

$$\sum_{p \in P(S)} x_p \leq 1, \qquad \forall S \in \mathcal{S}. \tag{5.46}$$

Seja λ_S a variável dual relacionada a cada restrição (5.46) no modelo (F3). A função objetivo do modelo (D) se torna

$$\sum_{i=1}^{n} \alpha_i + K\beta - \sum_{S \in \mathcal{S}} \lambda_S.$$

Além disso, esses cortes modificam o problema de *pricing*, já que as variáveis relacionadas

a essa restrição aparecem na restrição do dual como

$$\sum_{i \in p} \alpha_i + \beta - \sum_{\substack{S \in \mathcal{S}: \\ |I_p \cap S| \geq 2}} \lambda_S \leq w_p, \qquad \forall p \in P.$$

Seja $\delta_{i,j} = \sum_{p \in P} \alpha_i \alpha_j x_p$ a afinidade entre dois itens i e j. Como demonstrado em Da Silva e Schouery [19], uma tripla de itens $\{i, j, k\}$ deve ser adicionada no corte (5.46) se $\delta_{i,j} + \delta_{i,k} + \delta_{j,k} > 1$ e pelo menos dois termos de $\delta_{i,j} + \delta_{i,j} + \delta_{j,k}$ forem maiores que 0.

Utilizamos uma matriz de afinidade para calcular as afinidades entre os itens. Note que essa matriz é atualizada uma vez por padrão e cada par de itens no padrão, o que nos dá um custo de $O(n^2 \cdot |P|)$ para calcular toda a matriz.

Também usamos listas de adjacências para os itens a fim de manter informações sobre afinidades e acelerar a geração de novos cortes. Seja E_i a lista de adjacências de um item i, um item $j \in E_i$ se e somente se $\delta_{i,j} > 0$. A cada nó do *branch-and-bound*, criamos as listas de adjacências a partir da matriz de afinidades e verificamos se existe alguma tripla de itens $\{i, j, k\}$ tal que $\delta_{i,j} + \delta_{i,k} + \delta_{j,k} > 1$. O custo de criar as listas é $\Theta(n^2)$ e o custo de verificar cada tripla é $O(n \cdot \sum_{i \in I} |E_i|^2)$. Se os padrões possuem muitos itens, o custo dessa implementação se aproxima do custo da implementação trivial, $O(n^3)$. Mas, quando não há muitos itens por padrão (o que geralmente ocorre em instâncias mais difíceis), essa implementação consegue ser mais eficiente.

5.2.4 Rounded Capacity Inequality

A ideia dessa técnica é adicionar um limitante inferior para o valor da solução ótima, visando diminuir o número de iterações da geração de colunas [34, 83]. Seja LB um limitante para o ótimo, adicionamos a seguinte restrição ao modelo

$$\sum_{p \in P} w_p x_p + \sum_{i=1}^{n} s_i z_i \geq \lceil LB \rceil.$$

Iniciamos LB como $\max\{L \cdot K, \sum_{i=1}^{n} s_i / K\}$, e em cada nó atualizamos esse valor para o valor da solução dual viável do nó pai, calculada como na Seção 5.2.5. Essa restrição é simples de lidar no dual, pois apenas adiciona uma variável à função objetivo e à restrição utilizada no *pricing*. Chamaremos de γ essa variável. A restrição do modelo dual utilizada no *pricing* fica

$$\sum_{i \in p} \alpha_i + \beta - \sum_{\substack{S \in \mathcal{S}: \\ |I_p \cap S| \geq 2}} \lambda_S + w_p \gamma \leq w_p \qquad \forall p \in P \tag{5.47}$$

e a função objetivo adicionando essa variável fica

$$\sum_{i=1}^{n} \alpha_i + K\beta - \sum_{S \in \mathcal{S}} \lambda_S + \lceil LB \rceil \gamma.$$

A adição dessa nova restrição implicou em erros numéricos no Gurobi. Esses problemas foram contornados usando a opção NumericFocus = 3 do *solver*, que promete resolver o

modelo com maior precisão e resulta em custo computacional também maior. Apesar disso, com essa restrição foi possível encontrar soluções ótimas para mais instâncias.

5.2.5 Podando

Em cada nó do *branch-and-bound*, calculamos o valor de uma solução dual viável e podamos o nó quando esse valor for maior ou igual ao valor da melhor solução encontrada até o momento. Isso porque o valor da solução dual viável nunca diminuirá com a adição de novos padrões e, portanto, não é possível melhorar a solução a partir daquele nó. O valor de uma solução dual viável é calculado como mostrado na Seção 5.2.5.

Solução Dual Viável

Utilizamos o Gurobi para a implementação das formulações propostas. Como ele trabalha com um limite de precisão ε, são adicionados ao modelo apenas padrões que estão violados em pelo menos ε. Isso faz com que podar nós baseados no valor da solução dual não seja seguro, já que padrões violados em menos de ε não foram adicionados.

Por isso, precisamos viabilizar o valor da solução dual e, para isso, utilizamos a técnica apresentada por Pessoa et al. [72]. Sejam α, β, λ e γ uma solução dual do modelo e lembre-se que, após a inserção dos cortes e da *Rounded Capacity Inequality*, a função objetivo se tornou

$$\sum_{i=1}^{n} \alpha_i + K\beta - \sum_{S \in \mathcal{S}} \lambda_S + \lceil LB \rceil \gamma.$$

Calculamos o valor dual viável de uma solução dual como

$$\sum_{i \in I} \varepsilon \lfloor \alpha_i \varepsilon^{-1} \rfloor + \varepsilon \lfloor K\beta\varepsilon^{-1} \rfloor - \sum_{S \in \mathcal{S}} \varepsilon \lfloor \lambda_S \varepsilon^{-1} \rfloor + \varepsilon \lfloor \lceil LB \rceil \gamma \varepsilon^{-1} \rfloor.$$

Para evitar problemas numéricos relacionados ao uso de números de ponto flutuante, utilizamos ponto fixo para o cálculo da solução dual viável, assim como apresentado em Da Silva e Schouery [19]. Isso é feito multiplicando os valores por $2^{36} \approx 1.4 \cdot 10^{11}$ e convertendo para inteiros de 64 *bits*. Com essa precisão, é possível representar soluções de valor até 100 milhões sem *overflow*, o que foi suficiente para o *benchmark* utilizado.

A precisão do Gurobi foi definida como 10^{-9}. Como estamos trabalhando com números de ponto fixo, definimos nosso ε como $2^{-29} \approx 1.8 \cdot 10^{-9}$, já que dessa forma, todos os padrões violados no Gurobi também estarão violados em ε.

5.2.6 Precificação

Nesta seção, apresentamos algoritmo utilizado para gerar novos padrões. Note que o problema de *pricing* foi alterado pela adição de cortes e pela estratégia *Rounded Capacity Inequality*, apresentada na Seção 5.2.4. O novo problema de *pricing* é apresentado na restrição 5.47, a partir da qual extraímos os seguintes pares de condições para um padrão ser considerado violado:

$$\sum_{i \in p} \alpha_i > (1-\gamma)L - \beta + \sum_{\substack{S \in \mathcal{S}: \\ |I_p \cap S| \geq 2}} \lambda_S \qquad \text{t.q.} \qquad \sum_{i \in p} s_i \leq L \tag{5.48}$$

e

$$\sum_{i \in p} \alpha_i > (1-\gamma)\sum_{i \in p} s_i - \beta + \sum_{\substack{S \in \mathcal{S}: \\ |I_p \cap S| \geq 2}} \lambda_S \qquad \text{t.q.} \qquad L < \sum_{i \in p} s_i. \tag{5.49}$$

Em (5.48), consideramos o caso em que o padrão não estoura a capacidade do *bin* e, em (5.49), consideramos o caso em que a capacidade é estourada.

Note que a estratégia de *branching* utilizada também impacta no *pricing*, uma vez que os padrões gerados em cada nó devem considerar que dois itens a e b estão juntos ou separados e considerar quando itens estouram ou não o *bin*. Por isso, mantemos um grafo de conflito e uma estrutura *Disjoint Set Union* (DSU) durante as ramificações do *Branch-Cut-and-Price*. No grafo de conflitos, adicionamos uma aresta entre os itens a e b se eles devem aparecer separados e, no DSU, unimos itens que devem ser considerados juntos. Note que dois itens nunca podem aparecer unidos e possuir conflitos ao mesmo tempo. No algoritmo de *pricing*, consideramos todos os itens de um mesmo conjunto no DSU como um único item de tamanho que corresponde a soma dos tamanhos dos itens do conjunto. Em seguida, ordenamos a lista de itens por tamanho, do menor para o maior, colocamos os itens com conflito ao final da lista, e executamos o algoritmo com itens nessa ordem.

Utilizamos a programação dinâmica para o problema da mochila para obter padrões que satisfaçam as condições (5.48), consequentemente adicionando-os ao modelo. Nessa programação dinâmica, consideramos que o valor de um item i é α_i e o seu tamanho é s_i. Criamos uma matriz de capacidade $n \times L$, em que cada entrada $A(i, l)$ corresponde ao maior valor de solução considerando apenas os i primeiros itens e que possui preenchimento exatamente l. Devido à regra de *branching* de Ryan e Foster [74], os itens são divididos em dois conjuntos, itens sem conflitos e itens com conflitos, e dentro de cada conjunto os itens são ordenados do menor para o maior. A programação dinâmica considera a ordenação de itens dada pela concatenação dos conjuntos de itens sem conflitos e itens com conflito, nessa ordem. Segue a recorrência da programação dinâmica:

$$A(i,l) = \begin{cases} 0, & \text{se } l = 0, \\ -\infty, & \text{se } i = 1 \text{ e } s_1 \neq l, \\ \alpha_1, & \text{se } i = 1 \text{ e } s_1 = l, \\ A(i-1,l), & \text{se } s_i > l, \\ \max(A(i-1,l), A(i-1,l-s_i) + \alpha_i), & \text{caso contrário.} \end{cases}$$

Utilizamos o Algoritmo 6 para recuperar padrões a partir da programação dinâmica. Esse algoritmo recebe como parâmetro o índice do item que está sendo considerado i, o espaço restante l no padrão que está sendo construído, o padrão que está sendo construído p, o conjunto de padrões adicionados P e o número de chamadas recursivas q. O algoritmo considera adicionar o item i ao padrão atual p ou não o adicionar, fazendo chamadas recursivas para cada caso. Para considerar adicionar o item i, o algoritmo verifica se i não

Algoritmo 6 Algoritmo para encontrar padrões na matriz A.

1: Sejam n e L dados na entrada
2: Defina k como o número máximo de padrões por item
3: Defina *limit* como o limite de chamadas recursivas do algoritmo
4: **procedimento** FindPatterns(i, l, p, P, q)
5: **se** $q > limit$ **então devolve** ▷ Limite de chamadas recursivas
6: **se** $i < 0$ ou $l = 0$ **então devolve**
7: **se** $s_i = l$ e $p \cup \{i\}$ é um padrão violado **então**
8: $P \leftarrow P \cup \{p\}$ ▷ Apenas k padrões de menor custo reduzido por item
9: FindPatterns$(i-1,\ l,\ p,\ P,\ q+1)$
10: **se** i possui conflito com os itens de p ou $s_i > l$ **então devolve**
11: **se** i está fixado no topo ou $A(i-1, l-s_i) = -\infty$ **então devolve**
12: **se** $\sum_{i' \in p \cup \{i\}} \alpha_{i'} - A(i-1, l-s_i)$ satisfaz (5.48) ou (5.49) **então**
13: FindPatterns$(i-1,\ l-s_i,\ p \cup \{i\},\ P,\ q+1)$
14: **devolve**
15: $s_{min} \leftarrow \min_{i \in I} s_i$
16: $P \leftarrow \emptyset$
17: **para** $j \leftarrow L, L-1, \ldots, s_{min}$ **faça**
18: backtracking$(n,\ j,\ \emptyset,\ P,\ 0)$ ▷ Apenas padrões com preenchimento j
19: **para** $i \leftarrow i, i+1, \ldots, N$ **faça**
20: **para** $j \leftarrow 1, 2, \ldots, s_i - 1$ **faça**
21: backtracking$(i,\ L-j,\ \emptyset,\ P,\ 0)$ ▷ Padrões que estouram o item i

possui conflitos com os itens já adicionados ao padrão atual, se pode ser adicionado em l de espaço, se não está fixado no topo, se existe uma solução na matriz A que adiciona i na posição $l - s_i$ e se o padrão atual p adicionado de $A(i-1, l-s_i)$ satisfaz (5.48) ou (5.49). Quando um padrão violado é encontrado na Linha 8, o algoritmo adiciona ao conjunto total de padrões encontrados P.

Muitos padrões podem ser gerados de uma única vez e pode ser que o modelo fique muito grande e não resulte em grande melhoria. Por isso, para cada item, selecionamos apenas os k padrões violados de menor custo reduzido por iteração. O valor definido para k foi 6.

Durante o FindPatterns, se um conflito for encontrado, a ramificação é podada. Definimos $limit = nL/20$ como um limite na quantidade de chamadas recursivas em cada execução do algoritmo FindPatterns, para que este não fique muito tempo preso na enumeração de padrões. Garantimos que a otimalidade não seja afetada por esse limite, já que pelo menos o padrão mais violado será adicionado ao modelo.

No caso de padrões que não estouram o *bin* (condições (5.48)), consideramos apenas itens que não estão fixados como itens que estouram um *bin* pela primeira estratégia de *branching*. Nesse caso, o FindPatterns é executado para todos os tamanhos de padrões.

Já quando buscamos padrões que estouram o *bin* (condições (5.49)), primeiro enumeramos qual item estourará o *bin* e em qual posição. Nesse passo, o algoritmo ignora itens que foram fixados para aparecer dentro do *bin* pela primeira estratégia de *branching*. Seja i o índice de um item fixado para aparecer fora do *bin*, para evitar simetrias, enu-

meramos as possíveis posições em que i pode aparecer estourando o *bin* e executamos o FindPatterns considerando apenas os $i-1$ itens que apareceram antes na ordem da programação dinâmica e o espaço restante no padrão.

Também usamos uma técnica chamada *Waste Reduction* para evitar a enumeração de padrões que não podem melhorar a solução. Apresentamos essa técnica na Seção a seguir.

Waste Reduction

O *Waste Reduction* é uma técnica que nos permite restringir o preenchimento dos padrões que poderão ser adicionados ao modelo [19]. Se em determinado momento do algoritmo temos uma solução incumbente S de valor X, para melhorar esse incumbente precisamos encontrar uma solução S' de valor no máximo $X-1$. Seja $o = X - LK$ o preenchimento da solução S que estoura a capacidade dos *bins*. Se $o \leq 0$, a solução S é ótima, logo considere que $o > 0$. Com isso, todos os *bins* de S' possuem preenchimento no máximo $L + o - 1$, caso contrário a preenchimento que estoura em S' seria pelo menos o e S' não seria uma solução de valor melhor que S. Portanto, estaremos interessados em padrões p tal que

$$\sum_{i \in p} s_i \leq L + o - 1 = L + X - LK - 1 < X - L(K-1).$$

Por outro lado, também podemos limitar o mínimo que o padrão precisa estar cheio. Isso porque se um padrão está pouco cheio, os itens restantes devem ser adicionados pelos demais padrões. Seja $A = \sum_{i \in I} s_i$ o espaço total dos itens, no melhor dos casos a solução S empacotou $L(K-1)$ desse espaço em $K-1$ *bins* e $o = X - LK$ fora dos *bins*, restando $l = A - L(K-1) - o$ desse espaço para o *bin* menos ocupado de S. Para que S' tenha valor melhor que S, obrigatoriamente, todos os padrões de S' precisam ter preenchimento de pelo menos l. Assim, estaremos interessados em padrões p tal que

$$\sum_{i \in p} s_i \geq A - L(K-1) - o.$$

Dessa forma, sempre que uma nova solução é encontrada, atualizamos o valor de X e limitamos os padrões que poderão ser adicionados pelo algoritmo de *pricing*.

Como o *Waste Reduction* é aplicada ao preenchimento do padrão, podemos eliminar enumerações do FindPatterns antes mesmo de iniciá-las. Também usamos para eliminar padrões já adicionados ao modelo e que não satisfaçam mais as condições apresentadas nesta seção.

5.3 Experimentos

Os experimentos foram realizados com 2451 instâncias da literatura do Bin Packing Problem, obtidas a partir do benchmark *Bin Packing Problem Library* [30], e adaptadas para o Extensible Bin Packing. Foram usadas as classes de instâncias AI, ANI e aleatórias de Delorme et al. [29], além das instâncias de Falkenauer [31], Gschwind e Irnich [43], Scholl et al. [76], Schwerin e Wäscher [78], Wäscher e Gau [82] e a classe Hard

de Schoenfield [75]. Como essas instâncias foram originalmente desenvolvidas para BPP, elas não possuem o número de *bins* definido. Nos experimentos, definimos o número de *bins* de cada instância como $LB = \lceil \sum_{i \in I} s_i / L \rceil$, esse é um clássico limitante inferior para o BPP.

A Figura 5.5 apresenta um gráfico com diferentes valores para o número de *bins* e a porcentagem de instâncias que o algoritmo de Graham [40] consegue resolver de forma ótima em cada caso. Utilizamos o Lema 22 para identificar se a solução do Graham é ótima ou não. Em alguns casos (principalmente quando o número de *bins* aumenta), podemos encontrar uma solução ótima e não saber. Observe que quanto mais o valor aumenta ou diminui em relação à LB, mais instâncias o algoritmo de Graham consegue resolver, o que indica que as instâncias se tornam mais fáceis. Isso reforça que LB seja um bom valor para o número de *bins* nas instâncias, visando estressar nossos algoritmos.

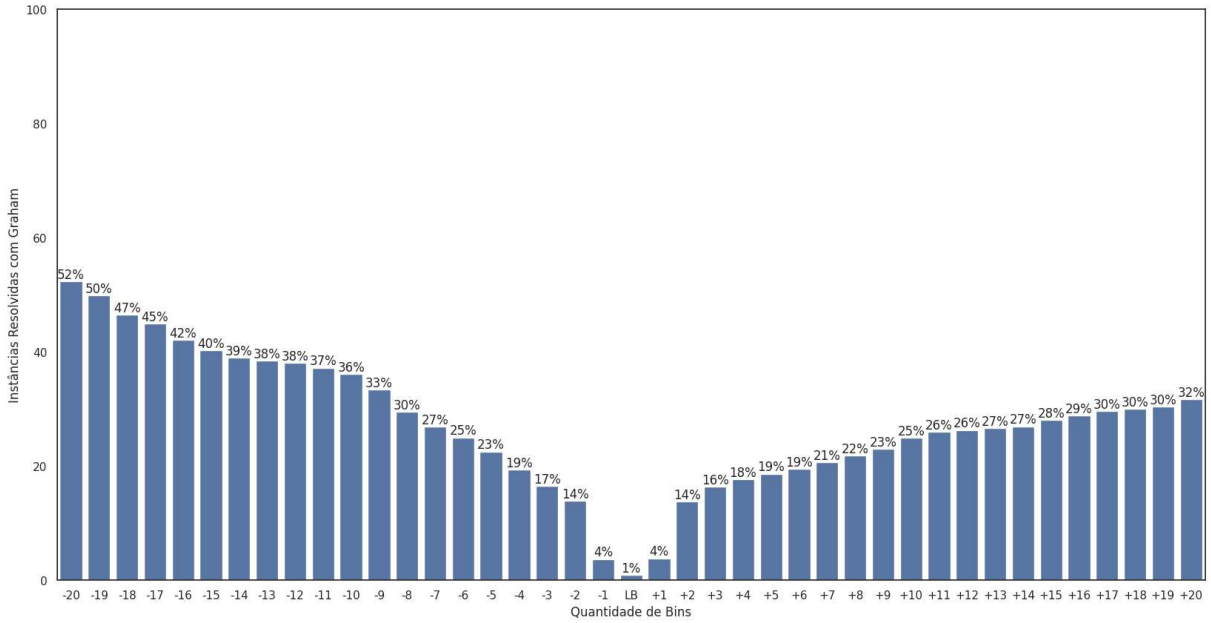

Figura 5.5: Porcentagem de instâncias que o algoritmo de Graham consegue resolver de forma ótima quando variamos o número de *bins* em relação à LB.

Foram desenvolvidos dois algoritmos *Branch-Cut-and-Price*, aplicando as técnicas apresentadas, mas variando apenas a estratégia de *branching*. O B&P é um algoritmo que usa apenas a ramificação de Ryan e Foster [74] no *branching*, ou seja, apenas fixa itens juntos ou separados. Já o B&P2 é um algoritmo que utiliza as duas estratégias de *branching*, como apresentado na Seção 5.2.1. Também foram desenvolvidos dois algoritmos de *Arc-Flow*, um com *reflect* e outro sem. O algoritmo *Arc-Flow* sem *reflect* corresponde à formulação (5.17)-(5.21) e chamaremos de AF. Já o algoritmo *Arc-Flow* com *reflect* corresponde à formulação (5.22)-(5.34) e chamaremos de AF+reflect.

Implementamos todo o funcionamento dos algoritmos *Branch-Cut-and-Price*, utilizamos o Gurobi apenas para resolver a relaxação linear do modelo. A implementação foi feita em C++.

Os testes foram executados para os algoritmos *Branch-Cut-and-Price* e *Arc-Flow*. A título de comparação, também executamos os testes com o modelo PLI baseado na formulação fraca de Martello e Toth [66] para o BPP, apresentado no início deste capítulo.

O tempo limite utilizado foi de 3600 segundos por execução e os testes foram executados em um computador com Intel(R) Xeon(R) E5-2420 1.90GHz, 32GB de memória RAM e Ubuntu 18.04.6 LTS.

5.4 Análise dos Resultados

Os resultados dos experimentos estão descritos na Tabela 5.1. A análise desses resultados revela algumas observações importantes sobre o desempenho relativo desses algoritmos, tanto em termos de eficácia na obtenção de soluções ótimas quanto na eficiência em relação ao tempo de execução. A média do tempo foi calculada apenas para as instâncias em que o algoritmo obteve o ótimo.

A coluna "Classe" identifica os conjuntos de instâncias testadas, enquanto "Total" indica o número total de instâncias em cada classe. As colunas "Opt" mostram a quantidade de instâncias resolvidas de forma ótima por cada método, e "Tempo" registra o tempo médio de execução em segundos. Os valores em negrito indicam os melhores resultados em cada categoria. Os tempos destacados em negrito consideram apenas aos métodos que resolveram o maior número de instâncias em cada classe, pois métodos que resolveram menos instâncias podem ter obtido tempos menores por resolverem, possivelmente, com instâncias mais simples. As colunas "# Vars" e "# Cols" representam, respectivamente, o número médio de variáveis e colunas utilizadas pelos métodos, refletindo os tamanhos dos modelos.

Em termos de sucesso na obtenção de soluções ótimas, os algoritmos de *Branch-Cut-and-Price* (B&P e B&P2) mostraram um desempenho superior em várias classes de instâncias, superando os demais algoritmos. O AF+reflect também apresentou bons resultados em algumas classes, mas foi menos consistente se comparado aos algoritmos de *Branch-Cut-and-Price*. O modelo PLI baseado na formulação fraca de Martello e Toth [66], por outro lado, falhou em encontrar soluções ótimas na maioria dos casos, indicando sua limitação em lidar com este tipo de problema. O B&P2 obteve melhores resultados que o B&P em quase todas as classes de instâncias, o que indica que combinar estratégias de *branching* melhorou o desempenho do algoritmo.

Quando se observa o tempo de execução, o B&P2 se destacou como o método mais eficiente, apresentando tempos menores em diversas classes de instâncias, como na classe *AI 1003*. O AF+reflect teve desempenho variável, sendo rápido em algumas classes de instâncias, como *FalkenauerU*, *Scholl* e *Schwerin*, mas apresentando maiores tempos em outras, como *ANI 201*. Para instâncias *AI* e *ANI*, os algoritmos baseados no método *Arc-Flow* e a formulação PLI mostraram tempos muito elevados ou, na maioria dos casos, nem conseguiram terminar a execução dentro do tempo limite estabelecido.

A análise de tamanho das formulações baseadas em *Arc-Flow*, observada através da média do número de variáveis, indica que o algoritmo AF tende a ter um número muito elevado de variáveis, especialmente em instâncias mais difíceis, como as da classe *Irnich*. Em contraste, o algoritmo AF+reflect possui menos variáveis em todas as instâncias, sugerindo uma maior eficiência em termos de uso de memória e complexidade computacional. As maiores reduções no número de variáveis no AF+reflect se deram nas classes *Falkenau-*

		PLI		AF			AF+reflect			B&P			B&P2		
Classe	**Total**	**Opt**	**Tempo**	**Opt**	**Tempo**	**# Vars**	**Opt**	**Tempo**	**# Vars**	**Opt**	**Tempo**	**# Cols**	**Opt**	**Tempo**	**# Cols**
AI 202	50	–	–	46	1407.0	204873	**50**	463.8	**135787**	**50**	55.4	10069	**50**	**26.8**	10209
AI 403	50	–	–	–	–	1448978	–	–	**936559**	41	1030.2	37185	**43**	**952.5**	38086
AI 601	50	–	–	–	–	4554909	–	–	**2891535**	**30**	1790.6	61785	**30**	**1734.1**	62531
AI 802	50	–	–	–	–	12045184	–	–	**7598208**	22	2441.3	104449	**25**	**2156.4**	104836
AI 1003	50	–	–	–	–	29059813	–	–	**18081554**	12	3200.9	148769	**16**	**2983.3**	153778
ANI 201	50	–	–	4	3524.2	202760	3	3578.7	**134862**	**42**	663.2	10444	**42**	**632.5**	11800
ANI 402	50	–	–	–	–	1441014	–	–	**934436**	12	3002.2	39957	**15**	**2772.8**	40583
ANI 600	50	–	–	–	–	4538218	–	–	**2886754**	**2**	**3521.0**	68652	–	–	70835
ANI 801	50	–	–	–	–	12011957	–	–	**7584992**	–	–	102557	–	–	272562
ANI 1002	50	–	–	–	–	28995287	–	–	**18065694**	–	–	1294357	–	–	2247405
FalkenauerT [31]	80	–	–	**80**	98.7	96519	**80**	23.1	**42606**	**80**	8.4	4666	**80**	**5.6**	4627
FalkenauerU [31]	80	30	2483.2	**80**	7.7	29679	**80**	**0.7**	**5784**	**80**	62.9	31288	**80**	49.8	31307
Hard [75]	28	–	–	**28**	**383.7**	61309	27	398.5	**57607**	24	576.2	5175	21	921.5	5107
Irnich [43]	240	–	–	–	–	90507217	–	–	**82043123**	10	3226.3	159722	**23**	**3002.3**	139431
Randomly [29]	96	4	3461.3	37	28.0	42195	21	8.1	**21055**	**89**	233.6	6594	**89**	228.3	6785
Scholl [76]	1210	541	2039.9	669	242.2	86138	630	15.4	**71906**	1197	115.0	8626	**1204**	**76.1**	8611
Schwerin [78]	200	161	708.2	**200**	31.0	37253	**200**	**0.8**	**11401**	**200**	9.3	10952	**200**	**6.0**	10871
Wäscher [82]	17	3	3036.1	6	2944.0	644822	14	1365.6	**247878**	**15**	**509.6**	14022	**15**	511.1	13539

Tabela 5.1: Comparação dos algoritmos desenvolvidos (tempo limite de 3600s).

erT com 80.5% de redução, *Schwerin* com 69.3% de redução e *Wäscher* com 61.5% de redução. Em média, houve uma redução de 39.4% no número de variáveis do AF+reflect em relação ao algoritmo AF.

Em classes como *AI 202*, todos os algoritmos, exceto o PLI, foram capazes de encontrar soluções ótimas, com o B&P2 mostrando o melhor desempenho em termos de tempo de execução. Nas classes *AI 403* e *AI 601*, o B&P2 novamente se destacou, tanto em termos de tempo quanto no número de soluções ótimas encontradas. Já em classes mais difíceis, como *ANI 801* e *ANI 1002*, nenhum dos algoritmos conseguiu obter a solução ótima para alguma das instâncias dentro do tempo limite.

Os algoritmos de *Branch-Cut-and-Price*, especialmente o B&P2, obtiveram melhores resultados para o Extensible Bin Packing. O B&P2 foi o algoritmo que resolveu mais instâncias em praticamente todas as classes, com exceção de *ANI 600* e *Hard*. O AF+reflect obteve bons resultados em alguns casos específicos, especialmente em relação ao tempo de execução, mas em termos gerais conseguiu resolver bem menos instâncias que o B&P e o B&P2. O modelo PLI baseado na formulação fraca de Martello e Toth [66] mostrou-se inadequado para este problema, não sendo competitivo em relação aos outros métodos analisados.

Capítulo 6

Conclusão

Desenvolvemos novos algoritmos para problemas de empacotamento com número fixo de recipientes. Mais especificamente, para os problemas MAXSPACE-RDV, POSITIONAL KNAPSACK PROBLEM e EXTENSIBLE BIN PACKING.

Apresentamos um *Fully Polynomial-Time Approximation Scheme* (FPTAS) para o POSITIONAL KNAPSACK PROBLEM com uma função de variação específica e um *PolynomialTime Approximation Scheme* (PTAS) para um conjunto de funções mais geral. Esses são os primeiros algoritmos de aproximação para esse problema.

Uma versão preliminar desse FPTAS foi apresentada no "*XII Latin-American Algorithms, Graphs and Optimization Symposium* - LAGOS" [70]. A versão completa do FPTAS e um PTAS para o POSITIONAL KNAPSACK PROBLEM foram submetidos para um periódico internacional. Para trabalhos futuros, pretendemos investigar para quais outros tipos de funções o FPTAS para o PKP continua válido e para quais funções o PKP se torna inaproximável.

Ainda em algoritmos de aproximação, apresentamos um PTAS para o MAXSPACE-RDV com K constante. Publicamos esse algoritmo no periódico "*Theory of Computing Systems* - TOCS" [71]. Esse é o primeiro algoritmo de aproximação para esse problema e é o melhor fator de aproximação possível, já que o MAXSPACE-RDV generaliza o MULTIPLE KNAPSACK PROBLEM, que é fortemente NP-difícil até mesmo com $K = 2$.

No contexto de algoritmos exatos, aplicamos as técnicas de *Branch-Cut-and-Price* e *Arc-flow* para desenvolver novos algoritmos para o EXTENSIBLE BIN PACKING. Realizamos experimentos com esses algoritmos utilizando o *benchmark Bin Packing Problem Library* [30]. Os algoritmos foram comparados entre si e com um modelo baseado na formulação fraca de Martello e Toth [66] para o BPP.

Os algoritmos de *Branch-Cut-and-Price*, obtiveram melhores resultados para o EXTENSIBLE BIN PACKING. O B&P2, algoritmo de *Branch-Cut-and-Price* que combina estratégias de *branching* diferentes, foi o algoritmo que resolveu mais instâncias em praticamente todas as classes. O AF+reflect, algoritmo de *Arc-Flow* que utiliza uma técnica chamada *reflect*, obteve bons resultados em alguns casos específicos, especialmente em relação ao tempo de execução, já que houve uma redução de até 80% no número de variáveis em relação ao algoritmo de *Arc-Flow* sem *reflect*. Mas, em termos gerais, os algoritmos baseados em *Arc-flow* conseguiram resolver menos instâncias que os algoritmos de *Branch-Cut-and-Price*. O modelo PLI baseado na formulação fraca de Martello e Toth [66]

mostrou-se inadequada para este problema, não sendo competitiva em relação aos outros métodos analisados.

Referências Bibliográficas

[1] Micah Adler, Phillip B Gibbons, and Yossi Matias. Scheduling space-sharing for internet advertising. *In Proceedings of Journal of Scheduling*, 5(2):103–119, 2002.

[2] Joachim H Ahrens and Gerd Finke. Merging and sorting applied to the zero-one knapsack problem. *Operations Research*, 23(6):1099–1109, 1975. doi: 10.1287/opre.23.6.1099.

[3] Ali Amiri and Syam Menon. Scheduling web banner advertisements with multiple display frequencies. *In Proceedings of IEEE Transactions on Systems, Man, and Cybernetics-Part A: Systems and Humans*, 36(2):245–251, 2006.

[4] Cynthia Barnhart, Ellis L Johnson, George L Nemhauser, Martin WP Savelsbergh, and Pamela H Vance. Branch-and-price: Column generation for solving huge integer programs. *Operations research*, 46(3):316–329, 1998.

[5] Richard S Barr and G Terry Ross. A linked list data structure for a binary knapsack algorithm. Technical report, Texas Univ at Austin Center for Cybernetic Studies, 1975.

[6] Mokhtar S Bazaraa, John J Jarvis, and Hanif D Sherali. *Linear programming and network flows*. John Wiley & Sons, 2011.

[7] Gaetan Belvaux and Laurence A Wolsey. bc—prod: A specialized branch-and-cut system for lot-sizing problems. *Management Science*, 46(5):724–738, 2000.

[8] Victor Boskamp, Alex Knoops, Flavius Frasincar, and Adriana Gabor. Maximizing revenue with allocation of multiple advertisements on a web banner. *Computers & Operations Research*, 38(10):1412–1424, 2011. doi: 10.1016/j.cor.2011.01.006.

[9] Rex Briggs and Nigel Hollis. Advertising on the web: Is there response before click-through? *Journal of Advertising Research*, 37(2):33–46, 1997.

[10] Valentina Cacchiani, Manuel Iori, Alberto Locatelli, and Silvano Martello. Knapsack problems — an overview of recent advances. part i: Single knapsack problems. *Computers & Operations Research*, 143:105692, 2022. doi: 10.1016/j.cor.2021.105692.

[11] Valentina Cacchiani, Manuel Iori, Alberto Locatelli, and Silvano Martello. Knapsack problems — an overview of recent advances. part ii: Multiple, multidimensional, and quadratic knapsack problems. *Computers & Operations Research*, 143:105693, 2022. doi: 10.1016/j.cor.2021.105693.

[12] Alberto Ceselli and Giovanni Righini. An optimization algorithm for the ordered open-end bin-packing problem. *Operations Research*, 56(2):425–436, 2008.

[13] Chandra Chekuri and Sanjeev Khanna. A polynomial time approximation scheme for the multiple knapsack problem. *SIAM Journal on Computing*, 35(3):713–728, 2005. doi: 10.1137/S0097539700382820.

[14] Chandra Chekuri and Rajeev Motwani. Precedence constrained scheduling to minimize sum of weighted completion times on a single machine. *Discrete Applied Mathematics*, 98(1-2):29–38, 1999. doi: 10.1016/S0166-218X(98)00143-7.

[15] Vasek Chvátal. Edmonds polytopes and a hierarchy of combinatorial problems. *Discrete mathematics*, 4(4):305–337, 1973.

[16] Edward G Coffman Jr. and George S Lueker. Approximation algorithms for extensible bin packing. In *Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms*, pages 586–588, 2001.

[17] Mauro R. C. Da Silva, Rafael C. S. Schouery, and Lehilton L. C. Pedrosa. A polynomial-time approximation scheme for the maxspace advertisement problem. *Electronic Notes in Theoretical Computer Science*, 346:699–710, 2019.

[18] Mauro Roberto Costa da Silva and Rafael Crivellari Saliba Schouery. Local-search based heuristics for advertisement scheduling. *RAIRO-Operations Research*, 58(4): 3203–3231, 2024.

[19] Renan FF da Silva and Rafael Schouery. A branch-and-cut-and-price algorithm for cutting stock and related problems. *arXiv preprint arXiv:2308.03595*, 2023.

[20] George B Dantzig and Mukund N Thapa. Duality and theorems of the alternatives. *Linear Programming: 2: Theory and Extensions*, pages 43–65, 2003.

[21] Milind Dawande, Subodha Kumar, and Chelliah Sriskandarajah. Performance bounds of algorithms for scheduling advertisements on a web page. *In Proceedings of Journal of Scheduling*, 6(4):373–394, 2003.

[22] JM Valério De Carvalho. Exact solution of bin-packing problems using column generation and branch-and-bound. *Annals of Operations Research*, 86:629–659, 1999.

[23] W Fernandez De La Vega and George S. Lueker. Bin packing can be solved within $1+\varepsilon$ in linear time. *Combinatorica*, 1(4):349–355, 1981.

[24] Vinícius Loti de Lima, Manuel Iori, and Flávio Keidi Miyazawa. Exact solution of network flow models with strong relaxations. *Mathematical Programming*, 197(2): 813–846, 2023.

[25] Brian C Dean and Michel X Goemans. Improved approximation algorithms for minimum-space advertisement scheduling. In *In Proceedings of International Colloquium on Automata, Languages, and Programming*, pages 1138–1152, 2003.

[26] Paolo Dell'Olmo, Hans Kellerer, Maria Grazia Speranza, and Zsolt Tuza. A 1312 approximation algorithm for bin packing with extendable bins. *Information Processing Letters*, 65(5):229–233, 1998.

[27] Maxence Delorme. *Mathematical models and decomposition algorithms for cutting and packing problems*. PhD thesis, alma, 2017.

[28] Maxence Delorme and Manuel Iori. Enhanced pseudo-polynomial formulations for bin packing and cutting stock problems. *INFORMS Journal on Computing*, 32(1): 101–119, 2020.

[29] Maxence Delorme, Manuel Iori, and Silvano Martello. Bin packing and cutting stock problems: Mathematical models and exact algorithms. *European Journal of Operational Research*, 255(1):1–20, 2016.

[30] Maxence Delorme, Manuel Iori, and Silvano Martello. Bpplib: a library for bin packing and cutting stock problems. *Optimization Letters*, 12(2):235–250, 2018.

[31] Emanuel Falkenauer. A hybrid grouping genetic algorithm for bin packing. *Journal of heuristics*, 2(1):5–30, 1996.

[32] Ari Freund and Joseph Seffi Naor. Approximating the advertisement placement problem. In *In Proceedings of International Conference on Integer Programming and Combinatorial Optimization*, pages 415–424, 2002.

[33] Alan M Frieze, Michael RB Clarke, et al. Approximation algorithms for the m-dimensional 0-1 knapsack problem: worst-case and probabilistic analyses. *European Journal of Operational Research*, 15(1):100–109, 1984.

[34] Mujin Gao, Yanru Chen, Junheng Li, and MIM Wahab. Hybrid branch-and-price-and-cut algorithm for the two-dimensional vector packing problem with time windows. *Computers & Operations Research*, 157:106267, 2023. doi: 10.1016/j.cor.2023.106267.

[35] Michael R Garey and David S Johnson. *Computers and Intractability*. Freeman, 1979.

[36] Stanisław Gawiejnowicz, Nir Halman, and Hans Kellerer. Knapsack problems with position-dependent item weights or profits. *Annals of Operations Research*, pages 1–20, 2023. doi: 10.1007/s10479-023-05265-x.

[37] Paul C Gilmore and Ralph E Gomory. A linear programming approach to the cutting-stock problem. *Operations research*, 9(6):849–859, 1961.

[38] Ralph E Gomory. Outline of an algorithm for integer solutions to linear programs. *Bulletin of the American Mathematical Society*, 64(5):275–278, 1958.

[39] Ralph E Gomory. *Outline of an algorithm for integer solutions to linear programs and an algorithm for the mixed integer problem*. Springer, 2010.

[40] Ronald L Graham. Bounds for certain multiprocessing anomalies. *Bell system technical journal*, 45(9):1563–1581, 1966.

[41] Ronald L Graham, Eugene L Lawler, Jan Karel Lenstra, and AHG Rinnooy Kan. Optimization and approximation in deterministic sequencing and scheduling: a survey. In *Annals of Discrete Mathematics*, pages 287–326. Elsevier, 1979.

[42] Harold Greenberg and Robert L Hegerich. A branch search algorithm for the knapsack problem. *Management Science*, 16(5):327–332, 1970. doi: 10.1287/mnsc.16.5.327.

[43] Timo Gschwind and Stefan Irnich. Dual inequalities for stabilized column generation revisited. *INFORMS Journal on Computing*, 28(1):175–194, 2016.

[44] Ahmed Hadjar, Odile Marcotte, and François Soumis. A branch-and-cut algorithm for the multiple depot vehicle scheduling problem. *Operations Research*, 54(1):130–149, 2006.

[45] Leslie A Hall, Andreas S Schulz, David B Shmoys, and Joel Wein. Scheduling to minimize average completion time: Off-line and on-line approximation algorithms. *Mathematics of operations research*, 22(3):513–544, 1997. doi: 10.1287/moor.22.3.513.

[46] Han Hoogeveen, Petra Schuurman, and Gerhard J Woeginger. Non-approximability results for scheduling problems with minsum criteria. In *Integer Programming and Combinatorial Optimization: 6th International IPCO Conference Houston, Texas, June 22–24, 1998 Proceedings*, pages 353–366. Springer, 1998. doi: 10.1007/3-540-69346-7_27.

[47] Ellis Horowitz and Sartaj Sahni. Computing partitions with applications to the knapsack problem. *Journal of the ACM*, 21(2):277–292, 1974. doi: 10.1145/321812.321823.

[48] IAB. Internet advertising revenue report: Full year 2022, 2022. URL `https://www.iab.com/wp-content/uploads/2023/04/IAB_PwC_Internet_Advertising_Revenue_Report_2022.pdf`. [Online; Accessed on: 2023-05-03].

[49] Oscar H Ibarra and Chul E Kim. Fast approximation algorithms for the knapsack and sum of subset problems. *Journal of the ACM*, 22(4):463–468, 1975. doi: 10.1145/321906.321909.

[50] Sungjin Im and Shi Li. Better unrelated machine scheduling for weighted completion time via random offsets from non-uniform distributions. In *2016 IEEE 57th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 138–147. IEEE, 2016. doi: 10.1109/FOCS.2016.23.

[51] Sungjin Im and Maryam Shadloo. Weighted completion time minimization for unrelated machines via iterative fair contention resolution*. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 2790–2809. SIAM, 2020. doi: 10.1137/1.9781611975994.170.

[52] Narendra Karmarkar. A new polynomial-time algorithm for linear programming. In *Proceedings of the sixteenth annual ACM symposium on Theory of computing*, pages 302–311, 1984.

[53] Narendra Karmarkar and Richard M Karp. An efficient approximation scheme for the one-dimensional bin-packing problem. In *23rd Annual Symposium on Foundations of Computer Science*, pages 312–320. IEEE, 1982.

[54] Hans Kellerer and Vitaly A Strusevich. A fully polynomial approximation scheme for the single machine weighted total tardiness problem with a common due date. *Theoretical Computer Science*, 369(1-3):230–238, 2006. doi: 10.1016/j.tcs.2006.08.030.

[55] Hans Kellerer, Ulrich Pferschy, and David Pisinger. Introduction to NP-Completeness of knapsack problems. In *Knapsack Problems*, pages 483–493. Springer, 2004. doi: 10.1007/978-3-540-24777-7_16.

[56] Gwang Kim and Ilkyeong Moon. Online banner advertisement scheduling for advertising effectiveness. *Computers & Industrial Engineering*, 140:106226, 2020. doi: 10.1016/j.cie.2019.106226.

[57] Peter J Kolesar. A branch and bound algorithm for the knapsack problem. *Management Science*, 13(9):723–735, 1967. doi: 10.1287/mnsc.13.9.723.

[58] Mikhail Y Kovalyov and Erwin Pesch. A generic approach to proving NP-hardness of partition type problems. *Discrete applied mathematics*, 158(17):1908–1912, 2010. doi: 10.1016/j.dam.2010.08.001.

[59] Subodha Kumar. *Optimization Issues in Web and Mobile Advertising: Past and Future Trends*. Springer, 2015.

[60] Subodha Kumar, Varghese S Jacob, and Chelliah Sriskandarajah. Scheduling advertisements on a web page to maximize revenue. *European Journal of Operational Research*, 173(3):1067–1089, 2006.

[61] Eugene L Lawler. Fast approximation algorithms for knapsack problems. *Mathematics of Operations Research*, 4(4):339–356, 1979. doi: 10.1109/SFCS.1977.11.

[62] Eugene L Lawler and J Michael Moore. A functional equation and its application to resource allocation and sequencing problems. *Management science*, 16(1):77–84, 1969. doi: 10.1287/mnsc.16.1.77.

[63] Eugene L Lawler and David E Wood. Branch-and-bound methods: A survey. *Operations research*, 14(4):699–719, 1966.

[64] Marco E Lübbecke and Jacques Desrosiers. Selected topics in column generation. *Operations research*, 53(6):1007–1023, 2005.

[65] Silvano Martello and Paolo Toth. An upper bound for the zero-one knapsack problem and a branch and bound algorithm. *European Journal of Operational Research*, 1(3): 169–175, 1977. doi: 10.1016/0377-2217(77)90024-8.

[66] Silvano Martello and Paolo Toth. *Knapsack problems: algorithms and computer implementations*. John Wiley & Sons, Inc., 1990.

[67] John Martinovic, Maxence Delorme, Manuel Iori, Guntram Scheithauer, and Nico Strasdat. Improved flow-based formulations for the skiving stock problem. *Computers & Operations Research*, 113:104770, 2020.

[68] Gerard Meurant. *Algorithms and complexity*. Elsevier, 2014.

[69] Robert M Nauss. An efficient algorithm for the 0-1 knapsack problem. *Management Science*, 23(1):27–31, 1976. doi: 10.1287/mnsc.23.1.27.

[70] Lehilton LC Pedrosa, Mauro RC da Silva, and Rafael CS Schouery. Positional knapsack problem: Np-hardness and approximation scheme (brief announcement). *Procedia Computer Science*, 223:400–402, 2023.

[71] Lehilton LC Pedrosa, Mauro RC da Silva, and Rafael CS Schouery. Approximation algorithms for the maxspace advertisement problem. *Theory of Computing Systems*, pages 1–20, 2024.

[72] Artur Pessoa, Ruslan Sadykov, Eduardo Uchoa, and François Vanderbeck. A generic exact solver for vehicle routing and related problems. *Mathematical Programming*, 183:483–523, 2020. doi: 10.1007/s10107-020-01523-z.

[73] Karri Rantasila and Lauri Ojala. Measurement of national-level logistics costs and performance. International Transport Forum Discussion Paper, 2012.

[74] David M Ryan and Brian A Foster. An integer programming approach to scheduling. *Computer scheduling of public transport urban passenger vehicle and crew scheduling*, pages 269–280, 1981.

[75] Jon E Schoenfield. Fast, exact solution of open bin packing problems without linear programming. *Draft, US Army Space and Missile Defense Command, Huntsville, Alabama, USA*, 2002.

[76] Armin Scholl, Robert Klein, and Christian Jürgens. Bison: A fast hybrid procedure for exactly solving the one-dimensional bin packing problem. *Computers & Operations Research*, 24(7):627–645, 1997.

[77] Andreas S Schulz. Scheduling to minimize total weighted completion time: Performance guarantees of lp-based heuristics and lower bounds. In *International conference on integer programming and combinatorial optimization*, pages 301–315. Springer, 1996. doi: 10.1007/3-540-61310-2_23.

[78] Petra Schwerin and Gerhard Wäscher. The bin-packing problem: A problem generator and some numerical experiments with ffd packing and mtp. *International Transactions in Operational Research*, 4(5-6):377–389, 1997.

[79] Michael Spivak. *Calculus*. Reverté, 2019.

[80] Paolo Toth. Dynamic programming algorithms for the zero-one knapsack problem. *Computing*, 25(1):29–45, 1980.

[81] Vijay V Vazirani. *Approximation algorithms*. Springer Science & Business Media, 2013.

[82] Gerhard Wäscher and Thomas Gau. Heuristics for the integer one-dimensional cutting stock problem: A computational study. *Operations-Research-Spektrum*, 18(3): 131–144, 1996.

[83] Lijun Wei, Minghui Lai, Andrew Lim, and Qian Hu. A branch-and-price algorithm for the two-dimensional vector packing problem. *European Journal of Operational Research*, 281(1):25–35, 2020. doi: 10.1016/j.ejor.2019.08.024.

[84] Zhao Wendan and Wang Dingwei. Study on internet advertising placement problem. In *2010 International Conference on Logistics Systems and Intelligent Management*, volume 3, pages 1798–1801, 2010.

[85] Deshi Ye and Guochuan Zhang. On-line extensible bin packing with unequal bin sizes. In *International Workshop on Approximation and Online Algorithms*, pages 235–247. Springer, 2003.

[86] Bowen Yuan, Yaxu Liu, Jui-Yang Hsia, Zhenhua Dong, and Chih-Jen Lin. Unbiased ad click prediction for position-aware advertising systems. In *Proceedings of the 14th ACM Conference on Recommender Systems*, pages 368–377, 2020. doi: 10.1145/3383313.3412241.

[87] Zeyuan Allen Zhu, Weizhu Chen, Tom Minka, Chenguang Zhu, and Zheng Chen. A novel click model and its applications to online advertising. In *Proceedings of the third ACM international conference on Web search and data mining*, pages 321–330, 2010. doi: 10.1145/1718487.1718528.

[88] Andris A Zoltners. A direct descent binary knapsack algorithm. *Journal of the ACM*, 25(2):304–311, 1978. doi: 10.1145/322063.322073.