ALGORITMOS EXATOS E HEURÍSTICOS PARA VARIAÇÕES DE PROBLEMAS DE ROTEAMENTO, EMPACOTAMENTO E CORTE GUILHOTINADO

EDUARDO THEODORO BOGUE

ALGORITMOS EXATOS E HEURÍSTICOS PARA VARIAÇÕES DE PROBLEMAS DE ROTEAMENTO, EMPACOTAMENTO E CORTE GUILHOTINADO

Tese apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Ciências Exatas da Universidade Federal de Minas Gerais como requisito parcial para a obtenção do grau de Doutor em Ciência da Computação.

Orientador: Thiago Ferreira de Noronha

Belo Horizonte

Julho de 2020

Ficha catalográfica elaborada pela bibliotecária Belkiz Inez Rezende Costa
CRB 6ª Região nº 1510

Bogue, Eduardo Theodoro.

B675a Algoritmos exatos e heurísticos para variações de problemas de roteamento, empacotamento e corte guilhotinado / Eduardo Theodoro Bogue — Belo Horizonte, 2020.
xviii, 68 f. il.; 29 cm.

Tese (doutorado) - Universidade Federal de Minas Gerais – Departamento de Ciência da Computação.
Orientador: Thiago Ferreira de Noronha.

1. Computação – Teses. 2. Programação dinamica. 3. Programação linear inteira. 4. Problema de roteamento de veículos. I. Orientador. II.Título.

CDU 519.6*61 (043)

UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

FOLHA DE APROVAÇÃO

Algoritmos Exatos e Heurísticos para Variações de Problemas de Roteamento, Empacotamento e Corte Guilhotinado

EDUARDO THEODORO BOGUE

Tese defendida e aprovada pela banca examinadora constituída pelos Senhores:

Prof. Thiago Ferreira de Noronha - Orientador
Departamento de Ciência da Computação - UFMG

Prof. Marcone Jamilson Freitas Souza
Departamento de Computação - UFOP

Prof. Sebastián Alberto Urrutia
Departamento de Ciência da Computação - UFMG

Prof. Martín Gomez Ravetti
Departamento de Ciência da Computação - UFMG

Prof. Haroldo Gambini Santos
Departamento de Ciência da Computação - UFOP

Belo Horizonte, 23 de Julho de 2020.

Agradecimentos

Com muito carinho, gostaria de agradecer aos meus pais, Moacir e Elizabeth, que sempre estiveram lá me apoiando e se sacrificaram para que eu pudesse ter uma boa educação.

Agradeço também ao meu orientador, Prof. Dr. Thiago Ferreira de Noronha, por todos os ensinamentos que me guiaram durante todo o doutorado, sempre estando presente e disposto a abdicar do seu tempo para que eu pudesse evoluir como profissional e pessoa.

Por fim, gostaria de agradecer a minha esposa e parceira Jussara, e ao nosso filho de quatro patas Plin, que sempre estiveram comigo me guiando, me apoiando e me aturando durante toda essa jornada, como também nos últimos 6 felizes anos em que estamos juntos!

Resumo

Problemas de otimização combinatória ocorrem em diversas situações práticas do dia a dia, como por exemplo em áreas de logística de transporte e distribuição, alocação de recursos, entre outras. Problemas de corte, empacotamento, e de roteamento de veículos são exemplos de problemas clássicos e muito bem estudados em otimização combinatória. Novas variantes destes problemas surgem à medida que aplicações exigem restrições adicionais ou relaxam algumas restrições do problema. Em particular, nesta tese estamos interessados em três variantes de problemas clássicos da literatura: o Problema do Roteamento de Veículos com Múltiplas Janelas de Tempo (VRPMTW), o Problema da Configuração de Produtos (PCP), e o Problema do Corte Guilhotinado Bidimensional em Três Estágios com Restrições de Precedência em Lote (2DCSP-BC). Para o VRPMTW, um algoritmo de Geração de Colunas (CG) e uma heurística pós-otimização baseada em uma heurística *Variable Neighborhood Search* (VNS) são propostos para fornecer limites inferiores e superiores ao custo da solução ótima. Os resultados mostraram que CG foi capaz de produzir limites inferiores para 66,7% das instâncias. Além disso, a heurística de pós-otimização foi capaz de melhorar a solução fornecida pela heurística VNS em 28,9%, de modo que para as instâncias onde limites inferiores são conhecidos, o *gap* de otimalidade foi de 6,0% em média. Para o PCP, uma formulação de programação linear inteira e um algoritmo exato de programação dinâmica de tempo pseudo-polinomial foram propostos. Experimentos computacionais mostraram que ambos os algoritmos foram capazes de encontrar soluções ótimas para todas as instâncias avaliadas com até 10.000 componentes. Por fim, três heurísticas foram desenvolvidas para o 2DCSP-BC, sendo duas heurísticas estendidas da literatura, e uma heurística baseada em um algoritmo de programação dinâmica. Os resultados mostraram que a heurística baseada em programação dinâmica obteve resultados superiores aos das heurísticas estendidas da literatura.

Palavras-chave: Programação Dinâmica, Programação Linear Inteira, Problema do Roteamento de Veículos com Múltiplas Janelas de Tempo, Problema do Corte Gui-

lhotinado Bidimensional em Três Estágios com Restrições de Precedência em Lote, Problema da Configuração de Produtos.

Abstract

Combinatorial optimization problems occur in many everyday practical situations, such as transportation and distribution logistics, resource allocation, among others. Cutting, packing and vehicle routing problems are examples of classic and well-studied combinatorial optimization problems. New variants of these problems emerge as applications require additional constraints or relax some constraints on the problem. In this thesis, we are interested in three variants of classical problems in literature: the Vehicle Routing Problem with Multiple Time Windows (VRPMTW), the Product Configuration Problem (PCP), and the 2-Dimensional Cutting Stock Problem with Batch Constraints (2DCSP-BC). For the Vehicle Routing Problem with Multiple Time Windows, we propose a Column Generation (CG) algorithm and a post optimization heuristic based on a Variable Neighborhood Search (VNS) to provide both lower and upper bounds for the cost of optimal solutions to VRPMTW. The results showed that CG was able to produce lower bounds, within one hour of running time, for 66.7% of the instances. Besides, the post optimization heuristic was able to improve the solution provided by the VNS heuristic in 28.9%, finding integer optimal solutions for 39.9% of the instances. Moreover, for the instances where lower bounds are known, the average optimality gap was 6.0% on average. In the Product Configuration Problem, an integer linear programming formulation and an exact pseudo-polynomial time dynamic programming algorithm were proposed. Computational experiments showed that both algorithms were able to find optimal solutions for all instances evaluated with up to 10.000 components. Finally, three heuristics were developed for the 2-Dimensional Cutting Stock Problem with Batch Constraints, where two heuristics are based on heuristics from the literature, and one heuristic is based on a dynamic programming algorithm. Computational experiments performed on realistic instances showed that the dynamic programming heuristic obtained the best results among the constructive heuristics.

Keywords: Dynamic Programming, Integer Linear Programming, Vehicle Routing

Problem with Multiple Time Windows, 2-Dimensional Cutting Stock Problem with Batch Constraints, Product Configuration Problem.

Lista de Figuras

Lista de Tabelas

Sumário

Capítulo 1

Introdução

Problemas de otimização combinatória ocorrem em diversas situações práticas do dia a dia, como por exemplo em áreas de logística de transporte e distribuição, alocação de recursos, entre outras. Problemas de corte, de empacotamento, e de roteamento de veículos são exemplos de problemas clássicos e muito bem estudados em otimização combinatória. Novas variantes destes problemas surgem à medida que aplicações exigem restrições adicionais ou relaxam algumas restrições do problema. Em particular, neste trabalho estamos interessados em três variantes de problemas clássicos da literatura: o Problema do Roteamento de Veículos com Múltiplas Janelas de Tempo, o Problema da Configuração de Produtos, e o Problema do Corte Guilhotinado Bidimensional em Três Estágios com Restrições de Precedência em Lote.

A motivação para investigar os problemas mencionados acima origina-se de suas aplicações práticas de necessidade industrial real, que são descritas ao decorrer deste trabalho. A proposta desta tese é investigar os problemas do ponto de vista algorítmico. Mais precisamente, deseja-se avaliar a adequabilidade de algoritmos heurísticos e exatos que se utilizam de técnicas de Programação Dinâmica (PD) e Programação Linear Inteira para resolver os problemas. Além da motivação oriunda das variadas aplicações práticas mencionadas acima, a direção de pesquisa escolhida nesta tese se justifica pelas seguintes razões. Primeiramente, todos os problemas mencionados são *NP-Difíceis*. Ademais, são poucos os trabalhos encontrados na literatura onde tenha sido feito um tratamento algorítmico destas variantes dos problemas. As contribuições dessa tese para o Problema do Roteamente de Veículos com Múltiplas Janelas de Tempo, o Problema da Configuração de Produtos, e o Problema do Corte Guilhotinado Bidimensional em Três Estágios com Restrições de Precedência em Lote, são descritas nas Seções 1.1, 1.2 e 1.3, respectivamente.

1.1 O Problema do Roteamento de Veículos com Múltiplas Janelas de Tempo

Dado um conjunto de veículos com a mesma capacidade e um conjunto de clientes, onde cada cliente possui uma demanda, um tempo de serviço e uma ou mais janelas de tempo, o Problema de Roteamento de Veículos com Janelas Múltiplas Janelas de Tempo (do inglês, *Vehicle Routing Problem with Multiple Time Windows* - VRPMTW) [Favaretto et al., 2007] consiste em encontrar um conjunto de rotas que começam e terminam em um único depósito, de modo que todos os clientes são visitados apenas uma vez e o tempo de serviço começa dentro de uma das respectivas janelas de tempo. O objetivo é minimizar uma função de custo que combina o tamanho da frota utilizada e o tempo gasto para visitar todos os clientes e retornar ao depósito [Belhaiza et al., 2014].

Para o VRPMTW, um algoritmo de geração de colunas que calcula a relaxação linear do problema é proposto. Ademais, uma heurística *Variable Neighborhood Search* e uma heurística de pós-otimização são desenvolvidas.

1.2 O Problema da Configuração de Produtos

Dado um conjunto de componentes que representam funcionalidades de um software, onde cada componente possui um custo e um benefício ao ser integrado a um produto, além de um conjunto de relacionamentos entre componentes, de modo que relacionamentos definem como os componentes podem ser combinados para obter um produto, o Problema da Configuração de Produtos (do inglês, *Product Configuration Problem* - PCP) consiste em selecionar um subconjunto de componentes que melhor atenda aos requisitos e preferências de um cliente, sujeito a restrições de orçamento.

Para o PCP, uma formulação de programação linear inteira, que é resolvida pelo algoritmo *branch-and-bound* do CPLEX, e um algoritmo exato de programação dinâmica de tempo pseudo-polinomial são propostos.

1.3 O Problema do Corte Guilhotinado Bidimensional em Três Estágios com Restrições de Precedência em Lote

No Problema do Corte Guilhotinado Bidimensional em Três Estágios com Restrições de Precedência em Lote (do inglês, *2-Dimensional Cutting Stock Problem with Batch Constraints* - 2DCSP-BC), um conjunto de placas e um conjunto de itens que devem ser cortados das placas são fornecidos, de modo que o objetivo do problema consiste em cortar todos os itens utilizando as placas disponíveis, visando minimizar a quantidade de material utilizado. A diferença entre o problema abordado nesta tese e outros na literatura é que os itens são organizados em lotes. Cada lote consiste no pedido de um cliente e é represendo por uma pilha, onde uma pilha $s = (\pi_1, \pi_2, \ldots, \pi_n)$ descreve a ordem na qual os itens devem ser cortados, de modo que os itens π_k sejam cortados antes dos itens π_{k+1}, para todo $k = 1, \ldots, n-1$.

Para o 2DCSP-BC, duas heurísticas construtivas extendidas da literatura e uma heurística baseada em programação dinâmica são propostas.

1.4 Organização do Texto

Os capítulos restantes desta tese estão organizados da seguinte maneira. O Capítulo 2 descreve o Problema do Roteamento de Veículos com Múltiplas Janelas de Tempo, onde dois algoritmos heurísticos e um algoritmo de geração de colunas são apresentados, em conjunto com uma avaliação experimental desses algoritmos. No Capítulo 3 é apresentado o Problema da Configuração de Produtos. Um algoritmo de programação dinâmica é proposto e comparado a um algoritmo de *branch-and-bound* desenvolvido. Experimentos computacionais são conduzidos para se avaliar a eficácia de cada um destes algoritmos. O Capítulo 4 descreve duas heurísticas construtivas e uma heurística baseada em programação dinâmica para o Problema do Corte Guilhotinado Bidimensional em Três Estágios com Restrições de Precedência em Lote. A heurística de programação dinâmica desenvolvida utiliza como subrotina o algoritmo exato de programação dinâmica proposto para o Problema da Mochila Retangular com Restrições de Precedência em Lote. Experimentos computacionais são realizados para se avaliar a eficácia das heurísticas propostas. Por fim, o Capítulo 5 apresenta as conclusões desta tese.

Capítulo 2

O Problema do Roteamento de Veículos com Múltiplas Janelas de Tempo

O Problema do Roteamento de Veículos com Múltiplas Janelas de Tempo é formalmente definido em [Belhaiza et al., 2014] da seguinte maneira. Seja $G = (V, A)$ um grafo completo não-direcionado, onde $V = \{0\} \cup C$ é o conjunto de nós, com 0 representando o depósito e $C = \{1, 2, ..., |C|\}$ representando os clientes, e A é o conjunto de arcos, com um tempo de viagem $t_{ij} \in \mathbb{R}^+$ associado a cada arco $(i, j) \in A$. Cada nó $i \in C$ tem uma demanda q_i e um tempo de serviço s_i. Além disso, cada nó em V (incluindo o depósito) possui um conjunto de janelas de tempo J_i, em que cada janela $p = [l_i^p, u_i^p] \in J_i$ define um período de tempo em que um veículo pode atender i. Além disso, seja R um conjunto de veículos, onde cada veículo $k \in R$ tem uma capacidade Q. Supõe-se que $|R| \geq |C|$ e que $q_i \leq Q$, ou seja, existem veículos suficientes para atender a todas as demandas dos clientes. As rotas devem começar e terminar no nó 0, e cada cliente $i \in C$ deve ser visitado uma vez por um dos veículos em R. Se um veículo $k \in R$ chegar a um cliente i fora de qualquer janela de tempo, ele deverá aguardar até o início da janela seguinte de i. Além disso, a soma das demandas de todos os clientes visitados na rota de um veículo $k \in R$ não pode ser maior que a capacidade Q do veículo. A função objetivo consiste em minimizar o tempo total de viagem e o número de veículos utilizados. Esses dois objetivos são escalarizados em [Belhaiza et al., 2014] por uma constante F, que representa o custo de frete, expresso em unidades de tempo, do uso de um veículo k. Esta função é declarada na Equação (2.1), em que $y_{ij}^k = 1$ se o arco $(i, j) \in A$ estiver na rota do veículo $k \in R$ e $y_{ij}^k = 0$ caso contrário. Além disso, $r^k = 1$ se o veículo $k \in R$ for usado e $r^k = 0$ caso contrário.

$$f(y) = \sum_{k \in R} \sum_{i \in V, j \in V \setminus \{i\}} t_{ij} y_{ij}^k + F \sum_{k \in R} r^k \tag{2.1}$$

Como o VRPMTW é uma generalização do Problema de Roteamento de Veículos Capacitados [Dantzig & Ramser, 1959], ele também é *NP-Difícil*. Até onde sabemos, nenhum método exato foi proposto até o momento capaz de resolver com eficiência pequenas e médias instâncias. Logo, nenhuma informação é conhecida sobre o quão boas são as soluções fornecidas pelas heurísticas para o VRPMTW.

Neste capítulo é proposto um algoritmo de geração de colunas (do inglês, *Column Generation* - CG) que fornece limites inferiores do custo das soluções ótimas. Como nos algoritmos de CG para VRP, o problema *master* é baseado em uma formulação do Problema da Cobertura de Conjuntos (do inglês, *Weighted Set Covering* - WSC). No entanto, devido às múltiplas janelas de tempo, o subproblema de *pricing* é um Problema do Caminho Elementar Mais Curto com Múltiplas Janelas de Tempo e Restrições de Capacidade, que é uma generalização do conhecido problema *NP-Difícil* do Caminho Elementar de Caminho Mais Curto com uma Janela de Tempo e Restrições de Capacidade [Desaulniers et al., 2006]. Além disso, uma heurística de pós-otimização que combina as rotas das soluções fornecidas pela heurística VNS e as geradas pelo algoritmo de *pricing* é proposta, a fim de melhorar os melhores limites superiores do custo das soluções ótimas.

O restante deste capítulo está organizado da seguinte forma. Trabalhos relacionados são discutidos na Seção 2.1. Uma formulação de PLI para VRPMTW, onde cada variável representa uma rota viável, é apresentada na Seção 2.2. Na Seção 2.3, um algoritmo de geração de colunas é proposto para calcular a relaxação linear dessa formulação. Como o custo computacional desse algoritmo aumenta exponencialmente com o tamanho das instâncias, apresentamos na Seção 2.4 uma heurística *Variable Neighborhood Search*, e na Seção 2.5 a heurística de pós-otimização. Os experimentos computacionais são apresentados na Seção 2.6, enquanto as observações finais são descritas na última seção.

2.1 Trabalhos Relacionados

Uma heurística de Sistema de Colônia de Formigas Múltiplas (do inglês, *Multiple Ant Colony System* - MACS) foi proposta em [Favaretto et al., 2007] para o Problema do Roteamento de Veículos com Múltiplas Janelas de Tempo e Múltiplas Visitas. Essa heurística começa com a construção de uma solução usando uma heurística baseada

em *Nearest Neighbor* [Lin & Kernighan, 1973] e, em seguida, duas colônias de formigas trabalham nessa solução. A primeira visa minimizar o número de rotas na solução, e a segunda visa reduzir o tempo total de viagem na solução fornecida pela primeira. Experimentos computacionais mostraram que os tempos de execução dessa heurística aumentam significativamente com a quantidade de janelas de tempo [Favaretto et al., 2007].

Uma formulação de Programação Linear Inteira (PLI) foi proposta em [Belhaiza et al., 2014], mas como o algoritmo exato com base nessa formulação não conseguiu resolver as menores instâncias, também foi proposta uma heurística *Hybrid Variable Neighborhood Tabu Search* (HVNTS). As restrições relacionadas à capacidade do veículo, às janelas de tempo e à duração máxima da rota são relaxadas e adicionadas como penalidades fixas na função objetivo. Essas penalidades devem ser grandes o suficiente para que soluções inviáveis sejam permitidas, mas desencorajadas se existirem soluções viáveis semelhantes. A fase de perturbação é baseada em cinco vizinhanças, e a fase de busca local consiste em uma busca tabu baseada em oito vizinhanças. Embora não seja garantido encontrar soluções viáveis, experimentos computacionais mostraram que soluções viáveis foram encontradas para todas as instâncias testadas. Esses experimentos também mostraram que o HVNTS obtém soluções de melhor custo que MACS.

Em [Ferreira et al., 2018], apresentamos um artigo sobre uma heurística *Variable Neighborhood Search* (VNS) que lida apenas com soluções viáveis. Neste trabalho preliminar, argumentamos que essa abordagem pode ser melhor do que a de trabalhos anteriores, visto que nenhum esforço computacional é gasto em soluções inviáveis e é garantido que soluções viáveis sejam encontradas (o que não é o caso do HVNTS). Os operadores de vizinhança usados são divididos em operadores de rota única e de múltiplas rotas. A heurística é composta por três estágios e em todos os estágios são permitidas apenas operações de rota única ou de múltiplas rotas que resultem em soluções viáveis. Primeiro, uma heurística gulosa cria uma solução viável. Em seguida, é executada uma heurística de Minimização de Rotas para diminuir o número de rotas de veículo não vazias nesta solução. Em seguida, o resultado da heurística de Minimização de Rotas é usado como a solução inicial da heurística VNS. Experimentos computacionais mostraram que o VNS encontrou soluções tão boas quanto as fornecidas pela heurística HVNTS.

2.2 Formulação de Programação Linear Inteira

Seja $\{a_{i\ell}\}$ uma matriz binária $m \times n$, com um conjunto de linhas M e um conjunto de colunas N. Dizemos que uma coluna $\ell \in N$ cobre a linha $i \in M$ se $a_{i\ell} = 1$. Além disso, seja c_ℓ o custo da coluna $\ell \in N$. O WSC [Karp, 1972] consiste em encontrar um subconjunto de colunas $N' \subseteq N$ com custo mínimo que cubra todas as linhas, ou seja, que minimize $\sum_{ell \in N'} c_\ell$, de modo que $\sum_{\ell \in N'} a_{i\ell} \geq 1, \forall i \in M$.

Nesta tese, modelamos o VRPMTW como um WSC. Cada linha $i \in M$ representa um cliente em C e existe uma coluna $\ell \in N$ associada a cada rota que aparece em qualquer solução do VRPMTW. Uma coluna ℓ cobre uma linha i se a rota representada por ℓ atende ao cliente representado por i, e c_ℓ é igual à soma dos tempos de viagem e do custo de frete da rota representada por ℓ.

Observamos que existe uma solução ótima N^* tal que nenhuma linha é coberta por mais de uma coluna em N, visto que todas as rotas viáveis do VRPMTW são representadas por uma coluna em N e a desigualdade triangular vale para os tempos de viagem t_{ij}, para todo $(i, j) \in A$. Portanto, uma solução ótima para VRPMTW pode ser construída com as rotas representadas pelas colunas em N^*. A partir desse resultado, podemos formular VRPMTW como um problema de programação linear inteira definido por (2.2) - (2.4), em que as variáveis $z_\ell = 1$ se rota correspondente a coluna $\ell \in N$ é usada e $z_\ell = 0$ caso contrário.

$$\text{Minimize} \quad \sum_{\ell \in N} c_\ell z_\ell \tag{2.2}$$

$$\text{s.t.} \quad \sum_{\ell \in N} a_{i\ell} z_\ell \geq 1, \qquad \forall i \in M, \tag{2.3}$$

$$z_\ell \in \{0, 1\} \qquad \forall \ell \in N \tag{2.4}$$

A função objetivo (2.2) minimiza a soma dos custos associados às rotas na solução. As desigualdades em (2.3) impõem que todos os clientes sejam visitados, e as restrições em (2.4) garantem a integralidade de z_ℓ, para todos os $\ell \in N$.

2.3 Algoritmo de Geração de Colunas

Como WSC é *NP-Difícil* [Karp, 1972] e o número de variáveis z cresce exponencialmente com o número de clientes no VRPMTW, estendemos o algoritmo de geração de colunas de [Desaulniers et al., 2006] para calcular a relaxação linear de (2.2) - (2.4), a

fim de fornecer limitantes inferiores para o VRPMTW. Esse procedimento vem do fato de que, em uma solução ótima de (2.2) - (2.4), a maioria das variáveis são não-básicas, ou seja, elas assumem um valor zero.

A geração de colunas é um método para resolver problemas de Programação Linear (PL) com um número exponencialmente grande de variáveis, que tem sido utilizado com sucesso na literatura [Barnhart et al., 1998; Chabrier, 2006; Gianessi et al., 2016; Taş et al., 2014]. A idéia por trás desse método é gerar sob demanda apenas as variáveis com custos reduzidos negativos a cada iteração do algoritmo. Portanto, variáveis com custos reduzidos positivos não precisam ser explicitamente adicionadas à formulação. Para esse fim, o problema é dividido em dois subproblemas. O *restriced master problem* é o problema original com apenas um subconjunto $N^h \subseteq N$ de variáveis, em que h é o contador de iteração do algoritmo. O subproblema de *pricing* é um novo problema criado para identificar uma variável z_ℓ, com $\ell \notin N^h$, com o menor custo reduzido. Sendo assim, o *restricted master problem* (2.5) - (2.7) é a relaxação linear da formulação (2.2) - (2.4), com um subconjunto restrito $N^h \subseteq N$ de variáveis. Assume-se que N^h admite pelo menos uma solução viável para as restrições em (2.6) - (2.7).

$$\text{Minimize} \quad \sum_{\ell \in N^h} c_\ell z_\ell \tag{2.5}$$

$$\text{s.t.} \quad \sum_{\ell \in N^h} a_{i\ell} z_\ell \geq 1, \qquad \forall i \in M, \tag{2.6}$$

$$z_\ell \in [0,1] \qquad \forall \ell \in N^h \tag{2.7}$$

O subproblema de *pricing* é modelado como sugerido em [Desaulniers et al., 2006]. O dual da formulação (2.5) - (2.7) é calculado para obter a solução ótima do dual π_i, para todos os $i \in M$, usados para calcular os custos reduzidos $\beta_\ell = \sum_{(i,j)\in A} \hat{c}_{ij} \bar{x}^\ell_{ij}$, em que $\hat{c}_{ij} = t_{ij} - \pi_i$ e $\bar{x}^\ell_{ij} = 1$ se o arco $(i,j) \in A$ estiver na rota $\ell \in N$ e $\bar{x}^\ell_{ij} = 0$ caso contrário. Para encontrar a coluna $\ell \in N$ com o menor valor de β_ℓ, resolvemos um Problema do Caminho Elementar Mais Curto com Múltiplas Janelas de Tempo e Restrições de Capacidade (ESPMTW), onde elementar significa que cada cliente pode aparecer no máximo uma vez no caminho mais curto. ESPMTW é definido sobre um grafo $\hat{G} = (\hat{V}, \hat{A})$, em que $\hat{V} = V \cup \{|C| + 1\}$ e $\hat{A} = A \cup \{(i, |C| + 1) : i \in C\}$, ou seja, uma cópia $|C| + 1$ do depósito 0 é adicionada ao grafo G. Além disso, temos que $J_{|C|+1} = J_0$ e $t_{i,|C|+1} = t_{0,i}$ para todos os $i \in C$. ESPMTW consiste em encontrar um caminho mais curto elementar de 0 a $|C| + 1$, de modo que cada cliente $i \in C$ seja visitado dentro de uma das janelas de tempo em J_i, e a soma das demandas do todos

os clientes nesse caminho não excedem a capacidade Q. Como uma generalização do Problema do Caminho Elementar Mais Curto com uma Janela de Tempo e Restrições de Capacidade [Desaulniers et al., 2006], o ESPMTW também é *NP-Difícil*. Como não há algoritmo ad-hoc para esse problema na literatura, o problema é formulado como um problema de Programação Linear Inteira e a formulação resultante é resolvida com um resolvedor de PLI.

ESPMTW é formulado pelas variáveis binárias $x_{ij} \in \{0,1\}$, para todo $(i,j) \in \hat{A}$, de modo que $x_{ij} = 1$ se o arco (i,j) está no caminho mais curto, e $x_{ij} = 0$ caso contrário. Também utilizamos variáveis auxiliares $v_{ip} \in \{0,1\}$, para todo $i \in \hat{V}$ e $p \in J_i$, de modo que $v_{ip} = 1$ se o nó i pode ser servido na janela de tempo $[l_i^p, u_i^p]$, e $v_{ip} = 0$ caso contrário, e variáveis $w_i \in \mathbb{R}$, para todo $i \in \hat{V}$, de modo que w_i seja igual ao tempo em que o serviço de i começa. A formulação de PLI resultante é dada por (2.8) - (2.18).

$$\text{Minimize} \quad \sum_{i \in \hat{V}} \sum_{j \in \hat{V}} \hat{c}_{ij} x_{ij} \tag{2.8}$$

$$\text{s.t.} \quad \sum_{i \in C} q_i \sum_{j \in \hat{V}} x_{ij} \leq Q, \tag{2.9}$$

$$\sum_{j \in \hat{V}} x_{0j} = 1, \tag{2.10}$$

$$\sum_{i \in \hat{V}} x_{iw} - \sum_{j \in \hat{V}} x_{wj} = 0, \qquad \forall w \in \hat{V} \tag{2.11}$$

$$\sum_{i \in \hat{V}} x_{i,|C|+1} = 1, \tag{2.12}$$

$$w_i + s_i + t_{ij} - M(1 - x_{ij}) \leq w_j, \qquad \forall i,j \in \hat{V} \tag{2.13}$$

$$w_i \geq \sum_{p \in J_i} l_i^p v_{ip}, \qquad \forall i \in \hat{V} \tag{2.14}$$

$$w_i \leq \sum_{p \in J_i} u_i^p v_{ip}, \qquad \forall i \in \hat{V} \tag{2.15}$$

$$\sum_{p \in J_i} v_{ip} = 1, \qquad \forall i \in \hat{V} \tag{2.16}$$

$$x_{ij} \in \{0,1\}, \qquad \forall i,j \in \hat{V} \tag{2.17}$$

$$v_{ip} \in \{0,1\} \qquad \forall i \in \hat{V}, \forall p \in J_i \tag{2.18}$$

A função objetivo (2.8) minimiza a soma dos custos associados aos arcos na

solução. As restrições em (2.9) reforçam as restrições de capacidade. As restrições em (2.10), (2.11) e (2.12) são restrições de fluxo que asseguram um caminho do depósito 0 para o depósito $|C| + 1$. As restrições em (2.13) garantem que se o cliente j for visitado imediatamente após o cliente i, o tempo de serviço do cliente i mais o tempo de viagem t_{ij} deverá ser menor ou igual ao tempo de serviço do cliente j, em que $M = \max_{i \in \hat{V}, p \in J_i}\{u_i^p\}$ é uma constante escalar. Essas restrições também garantem que o caminho seja elementar. As restrições em (2.14), (2.15) e (2.16) garantem que cada cliente seja atendido dentro de uma janela de tempo exclusiva. Por fim, as restrições em (2.17) e (2.18) são restrições de integralidade.

O pseudocódigo do algoritmo de geração de colunas (CG) é exibido no Algoritmo 1. O algoritmo inicia com um conjunto inicial de colunas (variáveis) $N^0 \subseteq N$ que é formado por todas as colunas de uma solução viável computada por qualquer heurística. Como o número de colunas geradas no procedimento de geração de colunas tende a reduzir com uma boa solução inicial [Barnhart et al., 1998], inicializamos N^0 com as variáveis em N que correspondem às rotas fornecidas pela heurística VNS descrita na Seção 2.4. Ademais, visando melhorar a solução retornada pela heurística VNS, aplicamos no grafo induzido pelos vértices presentes em cada rota $r \in N$ o algoritmo de programação dinâmica descrito na Seção 2.5.1 para o Problema do Caixeiro Viajante com Múltiplas Janelas de Tempo (do inglês, *Travelling Salesman Problem with Multiple Time Windows* - TSPMTW). O contador de iteração h é inicializado na linha 1. A cada iteração do loop das linhas 2-7, uma ou mais novas colunas são separadas. O dual do *restricted master problem* (2.6) - (2.7) é computado na linha 3, a fim de calcular os novos multiplicadores π^h e o valor γ^h da solução ótima. Em seguida, o subproblema de *pricing* é resolvido na linha 4 com base nesses multiplicadores para encontrar um conjunto L de colunas com custos reduzidos negativos. Como em [Desaulniers et al., 2006], L contém todas as soluções existentes com custos negativos encontrados pelo resolvedor de PLI. Além disso, esse algoritmo é encerrado assim que não encontra mais nenhuma coluna com custo reduzido negativo. Em seguida, o contador de iteração é incrementado na linha 6. Esse loop se repete enquanto pelo menos uma coluna com um custo reduzido negativo é encontrada. Caso contrário, temos que π^h é ótimo para o dual de (2.5) - (2.7). Portanto, γ^{h-1} é retornado na linha 8, pois é um limite inferior válido para (2.2) - (2.4). Além disso, se a solução primal associada a π^h for um número inteiro, a solução ótima para o *restricted master problem* também é ótima para o problema original.

Algoritmo 1 CG(N^0)

1: $h \leftarrow 0$
2: **do**
3: $(\pi^h, \gamma^h) \leftarrow$ Solve-Master's-Dual(N^h)
4: $L \leftarrow$ Solve-Pricing(π^h)
5: $N^{h+1} \leftarrow N^h \cup L$
6: $h \leftarrow h + 1$
7: **while** $L \neq \emptyset$
8: **return** γ^{h-1}

2.4 Heurística VNS

Conforme descrito na seção anterior, CG começa com um conjunto inicial de variáveis $N^0 \subseteq N$ que correspondem às rotas fornecidas pela heurística VNS que introduzimos em [Ferreira et al., 2018]. A heurística VNS é composta por três etapas. Em todas as etapas, apenas operações de rota única ou de múltiplas rotas que resultem em soluções viáveis são permitidas. Primeiro, uma heurística gulosa cria uma solução viável. Em seguida, a heurística de Minimização de Rotas é executada para diminuir o número de rotas nesta solução. Então, essa solução é utilizada como a solução inicial da heurística VNS. As três etapas, assim como os operadores de vizinhança que elas utilizam, são descritas a seguir.

Os operadores de vizinhança usados na heurística VNS foram aplicados pela primeira vez ao VRPMTW em [Belhaiza et al., 2014]. Eles são divididos em operadores de rota única e operadores de rota múltipla, de modo que o primeiro altera uma única rota, enquanto o último modifica ao menos duas rotas.

Existem dois operadores de rota única. O primeiro é o operador *single-route Or-opt*, que consiste em remover um cliente de sua posição atual e inseri-lo em outra posição na mesma rota. A Figura 2.1 mostra a rota $(0, 1, 2, 3, 4, 5, 0)$ de um veículo k e a Figura 2.2 ilustra a nova rota que pode ser obtida por meio de uma operação de *single-route Or-opt* do cliente 4 para a posição entre os clientes 2 e 3. Os arcos pontilhados representam os arcos eliminados da rota. O segundo é o operador *single-route 2-opt*, que consiste em trocar a posição de dois clientes em uma única rota. A Figura 2.3 ilustra a rota $(0, 1, 4, 3, 2, 5, 0)$ que pode ser obtida através de uma operação *single-route 2-opt* dos clientes 2 e 4 na rota exibida na Figura 2.1.

Existem cinco operadores multi-rota. O operador *multi-route shift(1,0)* consiste em remover um cliente de sua posição atual e inseri-lo em uma nova posição em outra rota. A Figura 2.4 ilustra as rotas $(0, 1, 2, 4, 5, 0)$ e $(0, 6, 7, 3, 8, 9, 10, 0)$ dos veículos k_1 e k_2, respectivamente, que podem ser obtidas por meio de uma operação *multi-route*

shift(1,0) do cliente 3 para a posição entre os clientes 7 e 8 nas rotas $(0, 1, 2, 3, 4, 5, 0)$ e $(0, 6, 7, 8, 9, 10, 0)$. O operador *multi-route swap(1,1)* consiste em trocar a posição de dois clientes de duas rotas distintas. A Figura 2.5 ilustra as rotas $(0, 1, 2, 8, 4, 5, 0)$ e $(0, 6, 7, 3, 9, 10, 0)$ dos veículos k_1 e k_2, respectivamente, que podem ser obtidas por meio de uma operação de *multi-route swap(1,1)* dos clientes 3 e 8 nas rotas $(0, 1, 2, 3, 4, 5, 0)$ e $(0, 6, 7, 8, 9, 10, 0)$. O operador *multi-route swap(1,1,1)* consiste em trocar a posição de três clientes $\langle i_1, i_2, i_3\rangle$ de três rotas distintas, de modo que i_2 assume a posição de i_1, i_3 assume a posição de i_2, e i_1 assume a posição de i_3. A Figura 2.6 ilustra as rotas $(0, 1, 2, 8, 4, 5, 0)$, $(0, 6, 7, 13, 9, 10, 0)$ e $(0, 11, 12, 3, 14, 15, 0)$ de veículos k_1, k_2 e k_3, respectivamente, que podem ser obtidas através de uma operação *multi-route swap(1,1,1)* dos clientes 3, 8 e 13 nas rotas $(0, 1, 2, 3, 4, 5, 0)$, $(0, 6, 7, 8, 9, 10, 0)$ e $(0, 11, 12, 13, 14, 15, 0)$.

O operador *multi-route swap(2,2)* consiste em trocar as posições de dois pares de clientes consecutivos $\langle i_1, j_1\rangle$ e $\langle i_2, j_2\rangle$ de duas rotas distintas, de modo que i_1 troca de posição com i_2 e j_1 troca de posição com j_2. A Figura 2.7 ilustra as rotas $(0, 1, 2, 8, 9, 5, 0)$ e $(0, 6, 7, 3, 4, 10, 0)$ dos veículos k_1 e k_2, respectivamente, que podem ser obtidas através de uma operação *multi-route swap(2,2)* dos arcos $(3, 4)$ e $(8, 9)$ nas rotas $(0, 1, 2, 3, 4, 5, 0)$ e $(0, 6, 7, 8, 9, 10, 0)$.

O operador *multi-route swap(2,2,2)* consiste em trocar as posições de três pares de clientes $\langle i_1, j_1\rangle$, $\langle i_2, j_2\rangle$ e $\langle i_3, j_3\rangle$, de três rotas distintas, de modo que $\langle i_2, j_2\rangle$ assume a posição de $\langle i_1, j_1\rangle$, $\langle i_3, j_3\rangle$ assume a posição de $\langle i_2, j_2\rangle$, e $\langle i_1, j_1\rangle$ assume a posição de $\langle i_3, j_3\rangle$. A Figura 2.8 ilustra as rotas $(0, 1, 2, 8, 9, 5, 0)$, $(0, 6, 7, 13, 14, 10, 0)$ e $(0, 11, 12, 3, 4, 15, 0)$ de veículos k_1, k_2 e k_3, respectivamente, que podem ser obtidas por meio de uma operação *multi-route swap(2,2,2)* dos arcos $(3, 4)$, $(8, 9)$ e $(13, 14)$ nas rotas $(0, 1, 2, 3, 4, 5, 0)$, $(0, 6, 7, 8, 9, 10, 0)$ e $(0, 11, 12, 13, 14, 15, 0)$.

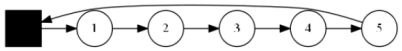

Figura 2.1: Exemplo de uma rota $(0, 1, 2, 3, 4, 5, 0)$

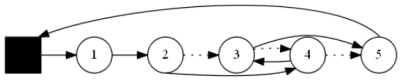

Figura 2.2: Exemplo de uma rota obtida pelo operador Single-route Or-opt aplicado a rota da Figura 2.1.

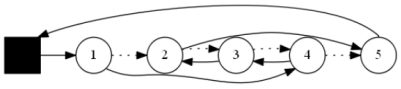

Figura 2.3: Exemplo de uma rota obtida pelo operador Single-route 2-opt aplicado a rota da Figura 2.1.

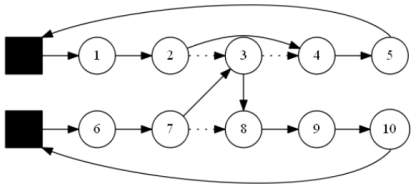

Figura 2.4: Operador Multi-route shift(1,0).

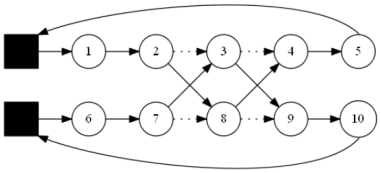

Figura 2.5: Operador Multi-route swap(1,1).

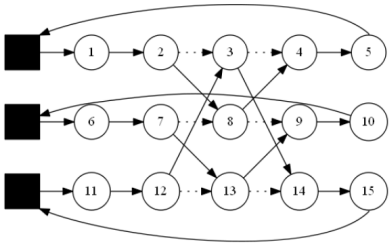

Figura 2.6: Operador Multi-route swap(1,1,1).

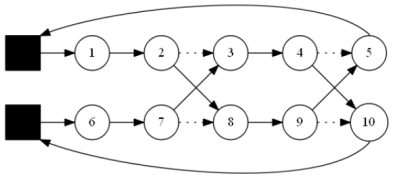

Figura 2.7: Operador Multi-route swap(2,2).

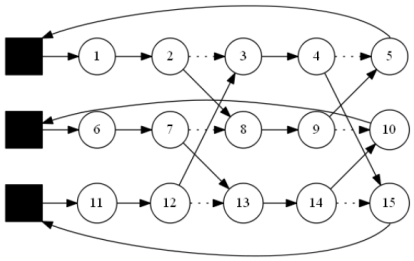

Figura 2.8: Operador Multi-route swap(2,2,2).

A heurística construtiva é baseada na heurística *Insertion* para VRPTW [Solomon, 1987]. A heurística começa com uma solução parcial em que a rota de todos os veículos em R está vazia. A cada iteração, os clientes que ainda não estão presentes na solução parcial são identificados e o custo incremental de inserir cada um deles em todas as posições possíveis nesta solução é avaliado. Em seguida, o cliente com o menor custo incremental é inserido na sua melhor posição possível. Esse procedimento para quando todos os clientes são inseridos na solução. Como $|R| \geq |C|$, as soluções fornecidas por essa heurística são viáveis para o VRPMTW.

A heurística de Minimização de Rotas visa minimizar o número de rotas na solução inicial S. Esse algoritmo é composto de um procedimento *Route Elimination* e um procedimento *Perturbation*. O procedimento $RouteElimination(k, S)$ tenta remover todos os clientes da rota de um veículo k em uma solução S aplicando sistematicamente os operadores *multi-route shift(1,0)* e *swap(1,1)* aos clientes na rota de k. Seja R^S o conjunto de veículos com rotas não vazias em S. Se o procedimento de eliminação de rota não puder esvaziar a rota de k, o procedimento $Perturbation(k, S)$ será aplicado para alterar aleatoriamente as rotas dos veículos em $R^S \setminus \{k\}$, esperando que os clientes na rota de k possam ser realocados na solução resultante. Esse procedimento consiste em selecionar aleatoriamente um dos cinco operadores de vizinhança de várias rotas descritas acima e aplicá-lo α vezes a clientes selecionados aleatoriamente.

O pseudocódigo da heurística de Minimização de Rotas $RouteMinimization(S)$ é apresentado no Algoritmo 2. O conjunto R^S é inicializado na linha 1 e o veículo $k \in R^S$ cuja rota tem o menor número de clientes em S é identificado na linha 2, onde $size(k, S)$ indica o número de clientes na rota de k. Em seguida, o procedimento *Route Elimination* é aplicado primeiro a k e S na linha 3. O loop das linhas 4 a 11 alterna entre o procedimento *Perturbation* e o procedimento *Route Elimination*. O primeiro é aplicado a k e S na linha 5 caso a rota de k não esteja vazia, e o último é aplicado a k e S na linha 6. Se a rota de k está vazia ou uma condição de estagnação é atendida, a heurística altera a rota sendo eliminada. Ela remove k de R^S na linha 8 e um novo veículo cuja rota tem o menor número de clientes em R^S é selecionado na linha 9. A condição de estagnação é acionada após β iterações sem esvaziar a rota de k. Seja $T = \frac{1}{Q}\sum_{i \in N} q_i$ um limite inferior ao número mínimo de veículos necessários para atender a todas as demandas dos clientes. A condição de parada é atendida quando o número de rotas não vazias em S é igual a T ou após as θ iterações do loop das linhas 4 a 11. No final desse loop, a solução corrente S é retornada na linha 12. Nesta tese, o valor de β e θ foi definido como 200 e 700, respectivamente, e o valor de α varia aleatoriamente de 5 a 20.

A heurística VNS visa minimizar a soma das durações das rotas em todos os veículos na solução retornada pela heurística de minimização de rotas, sem aumentar o número de rotas não vazias. Esse algoritmo é composto de um procedimento de perturbação *(Shaking)* e *Variable Neighborhood Descent (VND)*. O procedimento $Shaking(S, n)$ escolhe aleatoriamente entre os operadores *multi-route shift(1,0)* e *multi-route swap(1,1)*, e aplica o operador escolhido n vezes aos clientes escolhidos aleatoriamente em S. O procedimento $VND(S')$ é aplicado à solução S' retornada pelo procedimento de *Shaking*. O procedimento consiste em uma heurística clássica de *variable neighborhood descent*, composta por sete buscas locais, baseadas nos sete ope-

Algoritmo 2 *RouteMinimization*(S)

1: Seja R^S o conjunto de veículos com rotas não vazias em S.
2: $k \leftarrow argmin_{k' \in R^S} size(k', S)$
3: $RouteElimination(k, S)$
4: **while** Condição de parada não é atingida **do**
5: **if** $size(k, S) > 0$ **then** $Perturbation(k, S)$
6: $RouteElimination(k, S)$
7: **if** $size(k, S) = 0$ **or** condição de estagnação é atingida **then**
8: $R^S \leftarrow R^S \setminus \{k\}$
9: $k \leftarrow argmin_{k' \in R^S} size(k', S)$
10: **end if**
11: **end while**
12: **return** S

radores de vizinhança descritos acima. As buscas locais são ordenadas no VND na mesma ordem em que são definidas acima, de modo que esta ordem foi definida em função da complexidade dos tipos de movimentos utilizados, partindo dos movimentos de menor complexidade para maior complexidade. Além disso, depois que a última busca local é realizada, é aplicada uma versão restrita do procedimento de Eliminação de Rota, caso o número de rotas na solução corrente for maior que o limite inferior de T para o número mínimo de veículos necessários para transportar todas as demandas dos clientes.

O pseudocódigo da heurística VNS é apresentado no Algoritmo 3. A melhor solução conhecida S^* é inicializada na linha 1, enquanto o valor do tamanho da perturbação n é atribuído ao seu valor mínimo η_{min} na linha 2. A cada iteração do loop das linhas 3 a 18, um novo ótimo local é gerado aplicando o procedimento *Shaking* à solução atual S na linha 4 e, em seguida, aplicando o procedimento VND à solução resultante S' na linha 5. Se a nova solução S' for melhor que S (linha 6), a solução corrente e a melhor solução, bem como o tamanho da perturbação, são atualizados na linha 7. Caso contrário, o valor de n é incrementado na linha 9 se n for menor que seu valor máximo η_{max}. Se uma condição de estagnação é atingida na linha 11, ocorrerá uma reinicialização nas linhas 12 e 13 e a solução corrente será substituída por uma solução escolhida aleatoriamente entre as melhores soluções γ encontradas até o momento. Nesse caso, o tamanho da perturbação n também é redefinido para o seu menor valor eta_{min} na linha 14. A condição de estagnação é atingida após λ iterações sem melhorar a melhor solução encontrada. Já a condição de parada é definida em δ iterações do loop nas linhas 3 a 16. No final desse loop, a melhor solução encontrada S^* é retornada na linha 17. Nesta tese, os valores de γ, λ e δ foram definidos como 10, 60 e 500, respectivamente, e os valores de η_{min} e η_{max} foram definidos como 5 e 20,

respectivamente.

Algoritmo 3 VNS(S)

```
1: S* ← S
2: n ← η_min
3: while Condição de parada não é atingida do
4:     S' = Shaking(S, n)
5:     S' ← VND(S')
6:     if f(S') < f(S) then
7:         S ← S'
8:         S* ← best(S, S*)
9:         n ← η_min
10:    else if n < η_max then
11:        n ← n + 1
12:    end if
13:    if Condição de estagnação é atingida then
14:        Seja Γ o conjunto das melhores γ soluções encontradas até o momento.
15:        Seja S uma solução aleatória de Γ.
16:        n ← η_min
17:    end if
18: end while
19: return S*
```

2.5 Heurística POH

Os limites inferiores obtidos pela CG mostraram que os resultados da heurística VNS podem ser melhorados. Portanto, estendemos ao VRPMTW um *framework* heurístico de pós-otimização que foi aplicado com sucesso a outros problemas de otimização combinatória [Monaci & Toth, 2006; Morais et al., 2014; Sinclair et al., 2016; Taillard, 1999; Tempelmeier, 2011]. Embora exista alguma variação, todas essas abordagens consistem em resolver (exatamente ou heuristicamente) uma formulação WSCP do problema original com um subconjunto de colunas que são selecionadas de acordo com uma estratégia específica. É importante notar que, se pudéssemos gerar todas as colunas (ou seja, todas as rotas viáveis) e resolver a instância WSCP resultante, obteríamos uma solução ótima para o VRPMTW. No entanto, como o número de rotas cresce exponencialmente com o número de clientes, essa abordagem seria impraticável.

A Heurística de Pós-Otimização (do inglês, *Post Optimization Heuristic* - POH) para VRPMTW possui duas fases. Na primeira fase, um conjunto $\overline{N}$ de rotas é construído com (i) todas as rotas da melhor solução retornada pela heurística VNS, e (ii) todas as rotas obtidas resolvendo os subproblemas de *pricing* da CG. Além disso, para

cada rota $r \in \overline{N}$, aplicamos no grafo induzido pelos vértices presentes na rota r o algoritmo de programação dinâmica descrito na Seção 2.5.1 para o Problema do Caixeiro Viajante com Múltiplas Janelas de Tempo, com o objetivo de reduzir os custos de cada rota presente em $\overline{N}$.

Na segunda fase do algoritmo, para combinar as rotas fornecidas pelo VNS e pela CG em uma solução viável (possivelmente melhor), uma instância do WSCP é resolvida em uma matriz $\{a_{i\ell}\}$ com $\overline{N}$ colunas e M linhas, com cada linha em M representando um cliente. Como $\overline{N}$ é viável para VRPMTW, é garantido que essa heurística retorne uma solução viável.

2.5.1 Algoritmo de Programação Dinâmica para TSPMTW

O Problema do Caixeiro Viajante com Múltiplas Janelas de Tempo (do inglês, *Traveling Salesman Problem with Multiple Time Windows* - TSPMTW) consiste em encontrar um caminho que visita um conjunto de clientes, começando e terminando no mesmo cliente, a um custo mínimo. Cada cliente deve ser visitado exatamente uma vez e está associado a um tempo de serviço, de modo que o serviço deve começar dentro de uma das respectivas janelas de tempo de cada cliente.

TSPMTW pode ser definido formalmente da seguinte forma. Seja $G = (V, A)$ um grafo completo, onde $V = \{1, \ldots, n\}$ é o conjunto de nós e A é o conjunto de arcos, com um tempo de viagem $t_{ij} \in \mathbb{R}^+$ associado a cada arco $(i, j) \in A$ e tempo de serviço s_i associado a cada nó. Ademais, cada nó em V possui um conjunto de janelas de tempo J_i, em que cada janela $p = [l_i^p, u_i^p] \in J_i$ define um período de tempo em que um veículo pode chegar a i. As rotas devem começar e terminar no nó 0, e cada nó $i \in V$ deve ser visitado uma vez. A função objetivo do problema consiste em minimizar o tempo total de viagem. Esta função é declarada na Equação (2.19), em que $y_{ij} = 1$ se o arco $(i, j) \in A$ estiver na rota e $y_{ij} = 0$ caso contrário.

$$f(y) = \sum_{i \in V, j \in V \setminus \{i\}} t_{ij} y_{ij} \tag{2.19}$$

Como o TSPMTW é uma generalização do Problema do Caixeiro Viante (do inglês, *Traveling Salesman Problem* - TSP) [Gary & Johnson, 1979], ele também é *NP-Difícil*. O TSPMTW possui diversas aplicações de transporte e logística, como por exemplo o problema de organizar rotas de ônibus escolares para buscar crianças em um distrito escolar. Diferentes abordagens para resolver o TSP e suas variantes surgiram nas últimas décadas. Métodos exatos como algoritmos de planos de corte, algoritmos de *branch-and-bound* [Padberg & Rinaldi, 1987] e algoritmos de programação dinâmica

[Dumas et al., 1995], em geral conseguem apenas resolver instâncias pequenas, enquanto métodos heurísticos, tais como 2-opt e 3-opt [Lin & Kernighan, 1973], Cadeias de Markov [Martin, 1992] e Busca Tabu [Carlton & Barnes, 1996] são mais adequados para instâncias maiores.

Nesta tese, um algoritmo exato de programação dinâmica é proposto para o TSPMTW, onde utilizamos a abordagem *top-down* para computar a solução ótima do problema. Na abordagem de programação dinâmica *top-down*, quando uma solução ótima é calculada usando uma relação de recorrência, inicia-se com todo o problema, depois são calculadas recursivamente as soluções ótimas de seus subproblemas relacionados, até que os subproblemas se tornem casos base do problema. É importante observar que um estado não pode ser calculado até que todos os estados precedentes tenham sido computados, e que após o estado ser calculado, seu valor é armazenado em uma tabela para possíveis consultas futuras. Outra abordagem consiste em começar com soluções para os subproblemas relacionados aos casos base e expandi-las para soluções de subproblemas maiores, até a solução ótima de todo o problema ser computada. Esta abordagem é chamada de *bottom-up*.

O algoritmo desenvolvido é baseado no algorito de programação dinâmica proposto em [Dumas et al., 1995] para o Problema do Caixeiro Viajante com Janelas de Tempo. Seja $W_{i,t}$ o maior valor dentro do período de tempo de uma janela $p \in J_i$, tal que $t \geq W_{i,t}$. Caso $t \leq l_i^p$, para todo $p \in J_i$, então $W_{i,t} = -\infty$. Definimos $F(S, i, t)$ como o custo do caminho mais curto começando pelo nó 0 e terminando em i, de modo a visitar todos os nós do conjunto S exatamente uma vez e iniciar o serviço de i a partir do tempo t. Computa-se $F(S, i, t)$ resolvendo as equações de recorrência descritas em (2.20) - (2.22).

$$F(S,j,t) = \begin{cases} \infty & \text{se } \ t < 0 \qquad (2.20) \\ 0 & \text{se } \ |S| = 0 \qquad (2.21) \\ F(S - \{j\}, 0, W_{0,t-t_{0j}-s_0}) + t_{0j} & \text{se } \ |S| = 1 \qquad (2.22) \\ \min\limits_{(i,j)\in A} F(S - \{j\}, i, W_{i,t-t_{ij}-s_i}) + t_{ij} & \text{caso contrário} \qquad (2.23) \end{cases}$$

A solução ótima do problema pode ser calculada por $\min_{(j,0)\in A}(F(S -$

$\{0\}, j, W_{j,t'-t_{j0}-s_j}) + t_{j0})$, onde $t' = max_{p \in J_0}(u_0^p)$. Os casos base ocorrem quando $t < 0$, que ocorre quando o serviço do nó j não pode ser realizado dentro de suas respectivas janelas de tempo (2.20), e quando $|S| = 0$, indicando que todos os nós já foram visitados (2.21). O primeiro caso acontece quando $|S| = 1$, que ocorre quando devemos ligar o nó 0 ao próximo nó do caminho (2.22). Caso contrário precisamos selecionar um nó intermediário $i \in S$, onde o tempo de viagem $F(S - \{j\}, i, W_{i,t-t_{ij}-s_i})$ do caminho começando em 0 e terminando em i é adicionado ao tempo de viagem t_{ij}. Como o problema possui $2^n \times n \times t'$ estados, e cada estado pode ser computado em tempo computacional $O(n)$, a complexidade assintótica do algoritmo é $O(2^n n^2 t')$.

2.6 Experimentos Computacionais

Experimentos computacionais foram realizados em uma máquina Intel Xeon com 2.00 GHz de velocidade de clock e 16 GB de memória RAM. O algoritmo CG, a heurística VNS e a heurística POH foram implementados em C++ e compilados com o GNU *gcc* versão 6.3. As formulações de PLI do algoritmo CG e da heurística de pós-otimização foram resolvidas pelo IBM CPLEX versão 12.6 utilizando suas configurações de parâmetro padrões. O algoritmo CG foi interrompido após 3600 segundos de tempo de execução, enquanto a heurística POH foi executada até que uma solução ótima fosse encontrada. Além disso, a heurística VNS foi executada com os parâmetros padrões sugeridos em [Ferreira et al., 2018].

As 594 instâncias utilizadas nos experimentos são baseadas em um conjunto de 56 instâncias com 100 clientes proposto por Solomon [Solomon, 1987] para VRPTW. Essas instâncias diferem uma da outra pela janela de tempo de cada cliente, mas a posição geográfica de cada um dos 100 clientes é a mesma. Com base nessa característica, geramos três conjuntos de instâncias reproduzíveis, de modo a mesclar instâncias VRPTW e transformá-las em instâncias VRPMTW, nas quais cada cliente possui uma ou mais janelas de tempo.

O primeiro conjunto contém instâncias com uma janela de tempo por cliente e é dividido em três subconjuntos. O primeiro subconjunto $A13$ possui duas instâncias para cada instância VRPTW original. Ambas as instâncias são geradas selecionando os 13 primeiros clientes de cada instância original. No entanto, a primeira instância tem seu custo de frete definido como $F = 0$, ou seja, apenas o tempo total da viagem é considerado. Por outro lado, a segunda instância tem seu custo de frete definido como $F = \sum_{i \in V, j \in V} t_{ij}$, ou seja, nesse caso é sempre mais vantajoso utilizar uma menor frota de veículos. Os subconjuntos $A15$ e $A17$ são gerados da mesma maneira que $A13$, mas

Tabela 2.1: Resultados do algoritmo CG e das heurísticas nos nove conjuntos de instâncias.

Instâncias	$\|C\|$	$\|J_i\|$	lb (%)	opt. (%)	gap (%)	imp (%)	CG t(s)	POH t(s)
$A13$	13	1	70.53	51.78	6.28	28.12	254.48	12.24
$A15$	15	1	50.00	28.57	5.77	29.21	474.13	13.49
$A17$	17	1	51.78	33.03	5.80	30.18	335.31	12.22
$B13$	13	2	94.44	61.11	5.62	28.11	151.91	9.54
$B15$	15	2	96.29	64.81	2.35	30.87	105.11	7.71
$B17$	17	2	72.22	48.14	4.81	29.28	360.88	8.50
$C13$	13	3	81.25	46.87	10.11	29.26	634.22	13.10
$C15$	15	3	53.12	18.75	8.94	28.05	685.44	12.52
$C17$	17	3	31.25	6.25	4.41	27.55	678.45	12.36
		média:	66.76	39.92	6.01	28.95		

as instâncias nesses conjuntos são geradas selecionando os 15 e 17 primeiros clientes de cada instância original, respectivamente.

O segundo conjunto possui instâncias com duas janelas de tempo por cliente e também é dividido em três subconjuntos. Nesse caso, as instâncias originais são primeiro emparelhadas arbitrariamente. O primeiro subconjunto $B13$ possui duas instâncias para cada par de instância VRPTW original. As duas instâncias são geradas selecionando os 13 primeiros clientes de cada par de instâncias originais e mesclando as respectivas janelas de tempo. Como em $A13$, a primeira instância tem seu custo de frete definido como $F = 0$ e a segunda como $F = \sum_{i\ inV, j \in V} t_{ij}$. Os subconjuntos $B15$ e $B17$ são gerados da mesma forma que $B13$, mas as instâncias nesses conjuntos são feitas selecionando os 15 e 17 primeiros clientes de cada par de instâncias originais, respectivamente.

O terceiro conjunto consiste em instâncias com três janelas de tempo por cliente e também é dividido em três subconjuntos: $C13$, $C15$ e $C17$. Eles são gerados da mesma maneira que os conjuntos $B13$, $B15$ e $B17$, respectivamente, mas nesse caso as instâncias originais são arbitrariamente organizadas em grupos de três.

Os resultados são exibidos na Tabela 2.1. A primeira coluna identifica cada conjunto de instâncias. A segunda e a terceira coluna exibem o número de clientes e o número de janelas de tempo por cliente em cada instância, respectivamente. A quarta coluna mostra a porcentagem de execuções bem-sucedidas da CG, ou seja, o número de instâncias em que a CG terminou antes dos 3600 segundos e para o qual retornou um limite inferior válido. Para alguns desses casos, a solução ótima do relaxamento linear de 2.2 - 2.4 foi inteira, e portanto ótima. A porcentagem das 594 instâncias para as quais a CG encontrou soluções ótimas é relatada na quinta coluna. A sexta coluna

exibe o *gap* de otimalidade médio relativo ($\frac{\text{H}-\text{LP}}{\text{H}}$) entre o custo H da solução fornecida por POH e o limite inferior LP obtido pela CG. A sétima coluna relata a melhora média relativa ($\frac{\text{H}^i-\text{H}}{\text{H}^i}$) do custo H da solução fornecida pelo POH sobre o custo H^i da solução fornecida pela heurística VNS. Os tempos médios de execução das execuções bem-sucedidas da CG são apresentados na oitava coluna, enquanto os tempos de execução do VNS + POH são mostrados na última coluna. Pode-se observar que o algoritmo de geração de colunas conseguiu produzir limites inferiores para 66.76% das instâncias. Além disso, encontrou soluções ótimas para 39.92% das 594 instâncias. A heurística de pós-otimização também foi capaz de melhorar a solução fornecida pela heurística VNS em 28.95% em média. Por fim, para os casos em que os limites inferiores são conhecidos, o *gap* de otimalidade é de 6.01% em média, o que demonstra o bom desempenho da heurística de pós-otimização nas instâncias avaliadas.

Os tempos de execução das execuções do algoritmo da CG onde foi possível encontrar um limitante inferior também são relatados nos *box plots* [Spitzer et al., 2014] das Figuras 2.9, 2.10 e 2.11. As instâncias são agrupadas pelo número de janelas de tempo na Figura 2.9, pelo número de clientes na Figura 2.10, e pelo valor do custo do frete F na Figura 2.11. Esses gráficos exibem o primeiro quartil ($q1$), que equivale ao 25% maior tempo de execução, o segundo quartil ($q2$), que é a mediana dos tempos de execução, e o terceiro quartil ($q3$), que é o 75% maior tempo de execução. Além disso, eles também mostram as linhas *whisker*, que se estendem verticalmente a partir das caixas. O final do *whisker* superior é definido como $q_3 + 1.5 \times (q_3 - q_1)$, enquanto o final do *whisker* inferior é definido como $q_1 - 1.5 \times (q_3 - q_1)$. Os pontos não incluídos entre os *whiskers* superior e inferior são considerados *outliers* e são desenhados como pequenos círculos. A partir dos *box plots*, pode-se ver que a mediana dos tempos de execução é significativamente menor que o quartil superior, o que mostra que a maioria das execuções bem-sucedidas do GC leva menos de 200 segundos.

2.7 Considerações Finais

Para fornecer limites inferiores para o VRPMTW, foi proposto nesta tese um algoritmo Geração de Colunas, cujo conjunto inicial de colunas é inicializado por uma heurística VNS. Ademais, para melhorar os limites superiores obtidos por uma das melhores heurísticas para esse problema, um procedimento de pós-otimização é proposto baseado na mesma heurística VNS. O algoritmo de CG é o primeiro da literatura a fornecer limites inferiores para esse problema, enquanto a heurística de pós-otimização melhorou os resultados da heurística VNS em 28.95% em média para as instâncias avaliadas.

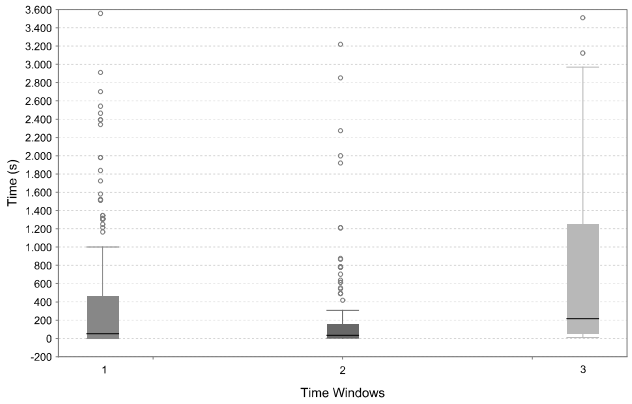

Figura 2.9: Tempos de execução da CG em instâncias com 1, 2 e 3 janelas de tempo.

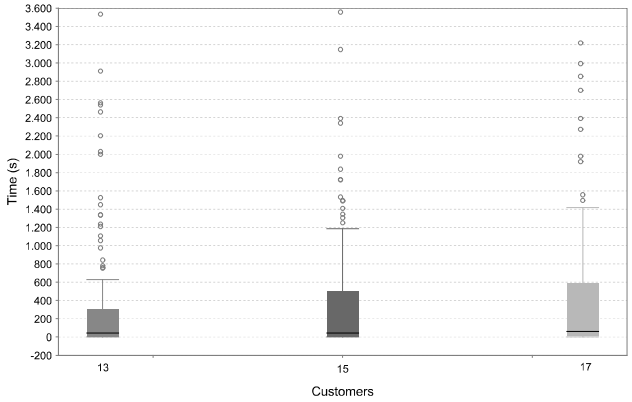

Figura 2.10: Tempos de execução da CG em instâncias com 13, 15 e 17 clientes.

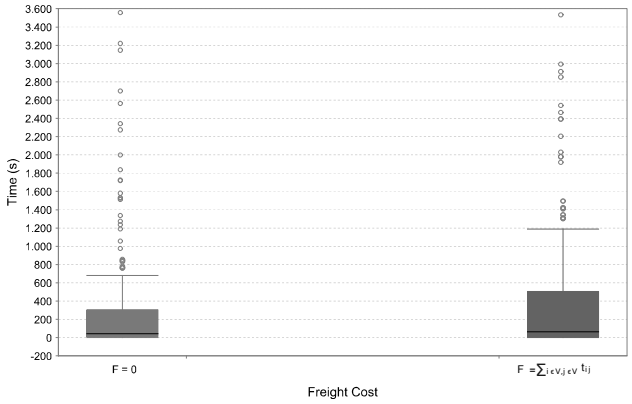

Figura 2.11: Tempos de execução da CG em instâncias com custo de frete $F = 0$ e $F = \sum_{i \in V, j \in V} t_{ij}$.

Os resultados mostraram que esses algoritmos podem ser aplicados com eficiência em instâncias com até 17 clientes.

Trabalhos futuros podem realizar um planejamento de experimentos para definir os valores mais adequados dos parâmetros da heurística VNS, visando obter melhores soluções, além de investigar métodos alternativos para gerar o primeiro conjunto de colunas N^0, com o intuito de começar com uma base melhor e diminuir o número de iterações do algoritmo de geração de colunas. Outra possibilidade consiste no desenvolvimento de métodos heurísticos para resolver mais rapidamente o subproblema de *pricing*. Além disso, como é sabido que algoritmos de geração de colunas sofrem de problemas de instabilidade quando executados em grandes problemas de LP, o que afeta a qualidade das colunas a serem geradas e, portanto, a velocidade geral de convergência do algoritmo, métodos de estabilização também podem ser aplicados [Amor et al., 2009].

Capítulo 3

O Problema da Configuração de Produtos

Uma Linha de Produção de Sofware (SPL) representa um conjunto de aplicativos de software (também conhecido como produtos) que compartilham um conjunto comum de componentes que atendem às necessidades específicas de um segmento de mercado específico [Pohl et al., 2005]. Um componente é um incremento na funcionalidade ou uma propriedade do sistema relevante para alguns clientes [Batory, 2005; Kang et al., 1990]. Todos os produtos que podem ser obtidos na SPL são especificados por um modelo de componentes e precisam atender a todas as condições impostas por ele.

Um modelo de componentes define (i) os componentes opcionais, que permitem a instanciação de diferentes produtos na SPL, e (ii) os componentes obrigatórios, que devem estar em todas as diferentes configurações de produtos [Pohl et al., 2005; Kang et al., 1990; Goedicke et al., 2004; Czarnecki & Eisenecker, 2007]. Eles são representados como uma árvore enraizada [Kang et al., 2002], em que os nós denotam os componentes do produto e os arcos descrevem os relacionamentos entre pares de componentes [Batory, 2005; Czarnecki & Wasowsky, 2007]. Esses relacionamentos definem como os componentes podem ser combinados para obter um produto válido.

O problema de selecionar um subconjunto de componentes que melhor atenda aos requisitos e preferências do cliente é chamado de Problema de Configuração do Produto (PCP). Existem muitas abordagens para o PCP na literatura [Asadi et al., 2014; Clements & Northrop, 2001; Pereira et al., 2015, 2017]. Nesta tese, vamos nos concentrar na abordagem de [Pereira et al., 2017], onde o PCP é definido como uma generalização do Problema da Mochila em Árvore.

O Problema de Configuração de Produtos é uma generalização do Problema da Mochila em Árvore (do inglês, *Tree Knapsack Problem* - TKP) [Johnson & Niemi,

1983], que é definido sobre uma árvore direcionada $T = (V, A)$ enraizada no nó $r \in R$, onde cada nó $i \in V$ é associado a um custo $c_i \in \mathbb{N}$ e um benefício $b_i \in \mathbb{N}$. TKP computa uma subárvore $T' \subseteq T$, com raiz em r, sujeito à soma dos custos dos nós em T' não excederem a capacidade $H \in \mathbb{N}$. O objetivo é encontrar a subárvore T^*, de forma que a soma dos benefícios dos nós em T^* seja a máxima. TKP é NP-difícil pois é uma generalização do problema da Mochila [Cho & Shaw, 1997]. Suas aplicações advém de problemas de *scheduling* e *network design* [Cho & Shaw, 1997]. A Figura 3.1 exibe a árvore de uma instância do TKP, onde cada nó está associado a um custo e um benefício, respectivamente.

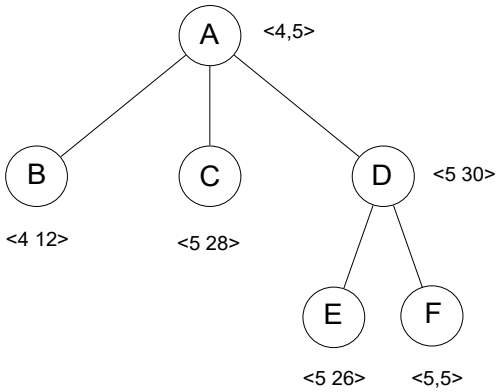

Figura 3.1: Exemplo de uma árvore de uma instância do Problema da Mochila em Árvore, onde cada nó está associado a um custo e um benefício.

No PCP, temos que os nós em V representam os componentes, e os arcos $(i, j) \in A$ indicam que o componente $j \in V$ pode ser selecionado apenas para o produto final se o componente $i \in V$ também estiver selecionado. Por exemplo, para que o produto final ofereça suporte ao componente j de *mensagens criptografadas de ponta a ponta*, o componente i da *troca de mensagens* também deve ser suportado. Portanto, neste caso, temos $(i, j) \in A$. Além disso, $c_i \in \mathbb{N}$ representa quanto o cliente paga pelo recurso i, $b_i \in \mathbb{N}$ expressa o grau de preferência do cliente por i e $H \in \mathbb{N}$ indica o orçamento do cliente.

O PCP difere do TKP pois está sujeito a restrições adicionais do modelo de componentes definidas pelos conjuntos dos componentes mandatórios $M \subseteq V$, dos componentes OR $O \subseteq V$, e dos componentes XOR $X \subseteq V$. As restrições *mandatórias* impõem que todo componente $i \in M$ deve ser necessariamente selecionado se seu componente pai $p_i \in V$ for selecionado para o produto final. Além disso, as restrições *OR* forçam que, se um componente $i \in O$ for selecionado, pelo menos um componente $j \in \delta(i)$ também deverá ser selecionado, onde $\delta(i) = \{j : (i, j) \in A\}$ é o *forward star* de

i. Ademais, as restrições XOR garantem que, se um componente $i \in X$ for selecionado, exatamente um componente $j \in \delta(i)$ também deverá ser selecionado. A Figura 3.2 exibe um exemplo gráfico de um modelo de componentes com $M = \{a, c, l\}$, $O = \{c, i\}$, $X = \{f, o\}$. Os componentes obrigatórios são representados por nós identificados por círculos tracejados, os componentes OR por nós caracterizados por arcos preenchidos e os componentes XOR por nós representados por arcos não preenchidos.

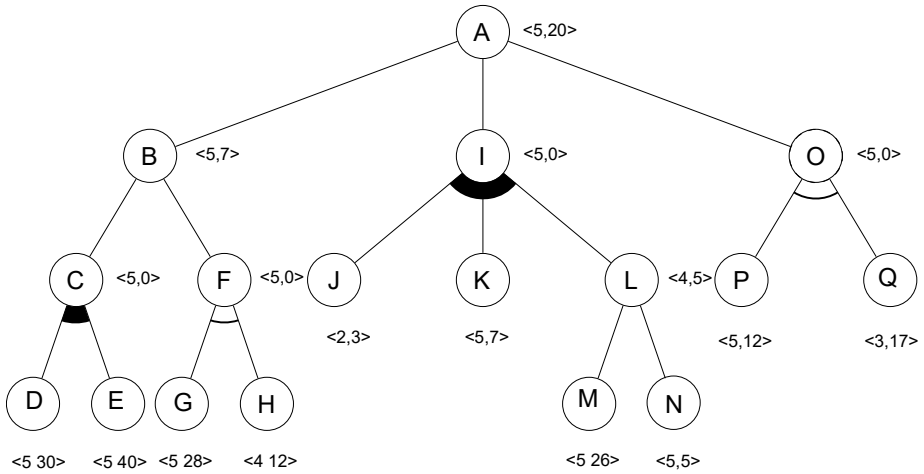

Figura 3.2: Exemplo de uma árvore de modelo de componentes. Os componentes XOR são indicados por arcos não preenchidos e os componentes OR com arcos preenchidos.

O PCP consiste em encontrar uma subárvore $T' \subseteq T$ com raiz em r, de forma que a soma dos benefícios sobre T' seja o máximo e as restrições de orçamento, mandatórias, OR e XOR sejam atendidas. Como o TKP é um caso particular de PCP quando $M = O = X = \emptyset$, o PCP é NP-difícil. Como não há uma abordagem conhecida para projetar algoritmos exatos de tempo polinomial para problemas nessa classe, propomos uma formulação de programação linear inteira para o PCP, que é resolvida pelo algoritmo de *branch-and-bound* do CPLEX, e um algoritmo exato de tempo pseudo-polinomial que resolve o PCP em $O(nH)$, onde $n = |V|$ é o número de componentes e H é o orçamento do cliente.

O restante deste capítulo está organizado da seguinte forma. Trabalhos relacionados são discutidos na Seção 3.1. Os algoritmos propostos nesta tese estão descritos nas Seções 3.2 e 3.3. Os experimentos computacionais são relatados na Seção 3.4, enquanto as observações e trabalhos futuros são apresentados na última seção.

3.1 Trabalhos Relacionados

Algoritmos exatos de complexidade exponencial foram propostos para diferentes variantes do PCP. Algoritmos de *Constraint Satisfaction* foram propostos em [Benavides et al., 2005]. Essa abordagem resolveu modelos de componentes com até 25 componentes. *Hierarchical Task Networks* foram aplicadas em [Asadi et al., 2014]. Essa abordagem encontrou configurações ótimas de produtos para modelos com até 200 componentes. Olaechea et al. [R. Olaechea & Rayside, 2012] aplicou o resolvedor Moolloy [Jackson et al., 2009] a outra variante do PCP. Seus experimentos mostraram que essa abordagem pode lidar com modelos de componentes com até uma dúzia de componentes. No entanto, como todas as outras abordagens listadas acima, ela não escala de forma eficiente para modelos com centenas de componentes.

Também existem trabalhos na literatura que se baseiam em algoritmos heurísticos de complexidade polinomial para diferentes variantes do PCP. *Filtered Cartesian Flattening* foi usado em [White et al., 2009] para encontrar configurações de produtos para modelos com até 10.000 componentes. Guo et al. [Guo et al., 2011] aplicou algoritmos genéticos (GA) ao PCP. Este último parte de uma configuração de produto inviável gerada aleatoriamente e a transforma em uma configuração de produto válida que está em conformidade com as restrições do modelo de componentes e que satisfaz os requisitos do cliente.

Pereira et al. [Pereira et al., 2017] propuseram a variante do PCP estudada nesta tese. No entanto, eles também mencionaram as chamadas restrições *cross-tree*. Essas restrições não foram incluídas neste trabalho porque são conhecidas por serem facilmente solucionáveis em modelos de componentes realistas [Pereira et al., 2017; Mendonca et al., 2009]. Os autores propõem um algoritmo genético de chave aleatória polarizada (BRKGA) e um algoritmo exato de *backtracking*. Experimentos computacionais, em instâncias com até 10.000 componentes, mostraram que o algoritmo de *backtracking* foi capaz de resolver apenas as 60 instâncias menores com até 43 componentes. Eles também mostraram que o BRKGA é capaz de encontrar soluções ótimas para essas 60 instâncias. No entanto, o tempo computacional médio dessa heurística foi de 2083.8 segundos nas instâncias com 10.000 componentes.

3.2 Formulação de Programação Linear Inteira

Nesta seção, apresentamos uma formulação de Programação Linear Inteira para o PCP. A formulação é baseada na formulação de PLI para o TKP apresentada em [Shaw & Cho, 1994]. Sejam as variáveis binárias $x_i \in \{0, 1\}$, de modo que $x_i = 1$ se o nó $i \in V$

estiver selecionado e $x_i = 0$ caso contrário. Podemos formular PCP como um problema de programação linear inteira definido por (3.1) - (3.7).

$$\text{Maximize} \quad \sum_{i \in I} b_i x_i \tag{3.1}$$

$$\text{s.t.} \quad \sum_{i \in I} c_i x_i \leq H \tag{3.2}$$

$$x_{p_i} \geq x_i, \quad \forall i \in V \tag{3.3}$$

$$x_{p_i} = x_i, \quad \forall i \in M \tag{3.4}$$

$$\sum_{j \in \delta(i)} x_j \geq x_i, \quad \forall i \in O \tag{3.5}$$

$$\sum_{j \in \delta(i)} x_j = x_i, \quad \forall i \in X \tag{3.6}$$

$$x_i \in \{0, 1\} \quad \forall i \in V \tag{3.7}$$

A função objetivo (3.1) maximiza a soma do benefício dos componentes selecionados. A restrição orçamentária é imposta pelas desigualdades em (3.2). As restrições de precedência em (3.3) garantem que um componente não possa ser selecionado se seu pai não estiver selecionado. As restrições mandatórias em (3.4) asseguram que um componente mandatório seja selecionado se seu pai for selecionado. As restrições OR em (3.5) garantem que, se $i \in O$, pelo menos um componente em $\delta(i)$ seja selecionado se i estiver selecionado. Além disso, as restrições XOR em (3.6) garantem que, se $i \in X$, exatamente um componente em $\delta(i)$ esteja selecionado se i estiver selecionado. Essa formulação é resolvida pelo algoritmo de *branch-and-bound* do CPLEX.

3.3 Algoritmo Exato de Programação Dinâmica

Como a complexidade de pior caso dos algoritmos de *branch-and-bound* baseados na formulação PLI da Seção 3.2 cresce exponencialmente com o número de componentes, nesta seção é proposto um algoritmo de tempo pseudo-polinomial de programação dinâmica. O algoritmo exato de programação dinâmica é proposto na Seção 3.3.2 e é uma generalização para o PCP do algoritmo de exato de programação dinâmica desenvolvido para o TKP descrito na Seção 3.3.1.

3.3.1 Algoritmo Exato de Programação Dinâmica para o TKP

Em Johnson e Niemi [Johnson & Niemi, 1983] foi introduzido o primeiro algoritmo de programação dinâmica para o TKP. A complexidade de tempo desse algoritmo é $\Theta(nC^*)$, onde C^* é a soma do benefício dos nós na solução ótima. Mais tarde, Cho e Shaw [Cho & Shaw, 1997] resolveram o Problema da Mochila Árvore em tempo $\Theta(nH)$, onde n é o número de nós em T e H é a capacidade da mochila. Experimentos computacionais mostraram que ele supera o procedimento de *branch-and-bound* proposto em [Shaw & Cho, 1994].

O algoritmo de programação dinâmica de Cho e Shaw [Cho & Shaw, 1997] é baseado nas seguintes equações de recorrência. Seja $L_k = \{0, 1, 2, \ldots, k\}$ um subconjunto de nós visitados em pré-ordem e $T_{L_k} = (L_k, A_{L_k})$ uma subárvore induzida por L_k em T. Para uma determinada capacidade h e um nó $v \in L_k$, $P_{L_k} = (v, h)$ é definido como a solução ótima para o TKP em uma árvore T_{L_k} com capacidade h, sujeito ao nó v esteja selecionado. A solução ótima do TKP é o máximo entre zero, se nenhum nó estiver selecionado, e $P_{L_n}(0, H)$). O último pode ser calculado pelas equações de recorrência (3.8) - (3.12).

$$P_{L_0}(0,h) = \begin{cases} b_0 & \text{se } \ c_0 \leq h \leq H \quad (3.8) \\ -\infty & \text{caso contrário} \quad (3.9) \end{cases}$$

$$P_{L_k}(k,h) = \begin{cases} P_{L_{k-1}}(p_k, h - c_k) + b_k & \text{se } \ \sum_{j \in P[0,k]} c_j \leq h \quad (3.10) \\ -\infty & \text{caso contrário} \quad (3.11) \end{cases}$$

$$Q_L(p_v, h) = \max(P_{L_{v-1}}(p_v, h), Q_L(v, h)) \quad (3.12)$$

Para todo $h = 0, 1, 2, \ldots, H$, $P_{L_0}(0, h)$ é inicializado usando as equações de recorrência em (3.8) - (3.9). Sempre que um novo nó é visitado na ordem de um caminho transversal de pré-ordem, ele é incluído como membro de L e $P_L(v, h)$ é avaliado para todo $h = 0, 1, 2, \ldots, H$ utilizando as equações de recorrência em (3.10) - (3.11). Se um nó folha ou um nó v, de modo que todos os seus sucessores foram visitados, for visitado, p_v é revisitado e $P_L(p_v, h)$ é avaliado, para todo $h = 0, 1, 2, \ldots, H$, usando a equação de recorrência em (3.12). Caso contrário, o primeiro sucessor não marcado do nó p_v é visitado. O procedimento acima continua até que o nó raiz seja visitado a partir do seu último sucessor, obtendo assim a solução ótima $\max(P_V(0, H), 0)$.

Descrevemos abaixo uma variante mais simples do algoritmo de programação

dinâmica proposto por [Cho & Shaw, 1997]. Essa variante será extendida em 3.3.2 para PCP. Seja $\Delta(v)$ a quantidade de nós da subárvore com raiz no nó v e π um vetor de nós com base em um caminho transversal de pré-ordem de T. O caminho em pré-ordem se faz necessário devido ao nó pai ser processado antes de quaisquer de seus filhos, sendo seguido de todos os nós da subárvore esquerda e posteriormente da subárvore direita. Definimos $F(k, H)$ como a soma de benefícios máxima começando no k-ésimo nó do vetor π, denotado por π_k, e tendo um orçamento H. Calculamos $F(k, H)$ resolvendo as equações de recorrência descritas em (3.13) - (3.15).

$$F(k,H) = \begin{cases} -\infty & \text{se } H < 0 \quad (3.13) \\ 0 & \text{se } k = |V| + 1 \quad (3.14) \\ \max\left(F(k + |\Delta(\pi_k)|, H),\ b_{\pi_k} + F(k+1, H - c_{\pi_k})\right) & \text{caso contrário} \quad (3.15) \end{cases}$$

A solução ótima do problema pode ser calculada por $F(0, H)$. Os casos base ocorrem quando $H < 0$, que indica que a solução é inviável pois o orçamento foi excedido (3.13), e quando $k = |V| + 1$, indicando que todos os nós já foram considerados(3.13).. No caso geral (3.15), existem duas possibilidades: (i) decidir entre selecionar o nó π_k ou (ii) não selecioná-lo. Se o nó π_k for selecionado, calculamos seu benefício e o adicionamos ao valor do próximo subproblema $F(k+1, K - c_{\pi_k})$. Caso contrário, a resposta é dada por $F(k + \Delta(k), H)$, onde $k + \Delta(k)$ é o índice do próximo nó que não está na subárvore de k, pois não precisamos avaliar a subárvore enraizada no nó k. Como o problema possui $n \times H$ estados e cada estado pode ser computado em tempo $\Theta(1)$, a complexidade assintótica do algoritmo é $O(nH)$.

3.3.2 Algoritmo Exato de Programação Dinâmica para o PCP

Nesta seção, um algoritmo exato de programação dinâmica para o PCP é apresentado. Conforme mencionado anteriormente, o algoritmo de programação dinâmica desenvolvido para o PCP é uma generalização do algoritmo de programação dinâmica elaborado para o TKP. Contudo, a realização de um pré-processamento é necessário para algumas instâncias do PCP. Para todo nó $i \in X \cup O$, sendo i o nó pai de um nó $j \in X \cup O$, precisamos criar um nó artificial l interligando i a j, de forma que $V = V \cup \{l\}$ e $A = A \cup \{(i, l)\} \cup \{(l, j)\} - \{(i, j)\}$, onde $b_l = c_l = 0$. Esse pré-processamento é necessário visto que caso nó $i \in X \cup O$ seja selecionado, o algoritmo de programação dinâmica imediatamente seleciona também um nó $j \in \delta(i)$, de modo que caso $j \in X \cup O$, um nó $p \in \delta(j)$ também deve ser selecionado.

Definimos $F(k, H)$ como a soma de benefícios máxima do cliente, começando

pela componente representada pelo nó $\pi_v \in V$ e tendo um orçamento H. $F(k, H)$ pode ser computado resolvendo as equações de recorrência descritas em (3.16) - (3.21).

$$F(k,H) = \begin{cases} -\infty & \text{se } H < 0 \quad (3.16) \\ 0 & \text{se } k = n+1 \quad (3.17) \\ F(k + |\Delta(\pi_k)|, H) & \text{se } p_{\pi_k} \in X \quad (3.18) \\ \max\left(F(k + |\Delta(\pi_k)|, H),\ b_{\pi_k} + F(k+1, H - c_{\pi_k})\right) & \text{se } \pi_k \notin M \wedge \pi_k \notin \{O \cup X\} \quad (3.19) \\ b_{\pi_k} + F(k+1, H - c_{\pi_k}) & \text{se } \pi_k \in M \wedge \pi_k \notin \{O \cup X\} \quad (3.20) \\ \max_{(\pi_y \in \delta(\pi_k)}(b_{\pi_k} + b_{\pi_y} + F(y+1, H - c_{\pi_k} - c_{\pi_y})) & \text{se } \pi_k \in M \wedge \pi_k \in \{O \cup X\} \quad (3.21) \\ \max(F(k + |\Delta(\pi_k)|, H), \max_{(\pi_y \in \delta(\pi_k)}(b_{\pi_k} + b_{\pi_y} + F(y+1, H - c_{\pi_k} - c_{\pi_y})) & \text{se } \pi_k \notin M \wedge \pi_k \in \{O \cup X\} \quad (3.22) \end{cases}$$

O custo da solução ótima do problema pode ser calculado por $F(0, H)$. Os casos base ocorrem quando $H < 0$, que indica que a solução é inviável pois o orçamento foi excedido (3.16), quando $k = n + 1$, indicando que todos os nós já foram considerados (3.17), e quando $p_{\pi_k} \in X$ (3.18), já que π_k já foi considerado nas equações de recorrência (3.21) e (3.22). O caso geral agora está dividido em 4 casos, descritos pelas equações de recorrência (3.19) - (3.22). O primeiro caso ocorre quando o nó não é mandatório e nem XOR ou OR (3.19), enquanto no segundo caso o nó é mandatório, mas não é XOR ou OR (3.20). No terceiro caso, o nó é mandatório e XOR ou OR (3.21). Finalmente, no quarto caso, o nó não é mandatório, mas é XOR ou OR (3.22).

No primeiro caso, como $\pi_k \notin M$, existem duas possibilidades: (i) decidir entre selecionar o nó π_k ou (ii) não selecioná-lo. Se o nó π_k for selecionado, calculamos seu benefício e o adicionamos ao valor do próximo subproblema $F(k + 1, K - c_{\pi_k})$. Caso contrário, a resposta é dada por $F(k + \Delta(k), H)$, pois não precisamos avaliar a subárvore enraizada no nó k. Como $\pi_k \in M$, o nó π_k deve ser selecionado no segundo caso. O terceiro caso ocorre quando o nó π_k é XOR ou OR e também é mandatório, portanto, devemos selecionar o nó π_k e um nó $\pi_y \in \delta(\pi_k)$. Finalmente, o quarto caso acontece quando o nó π_k é XOR ou OR, mas não é mandatório, portanto, devemos decidir entre não selecionar o nó ou selecionar o nó e um nó $\pi_y \in \delta(\pi_k)$.

Como o problema possui $n \times H$ estados e cada estado pode ser computado em tempo computacional $O(|\delta(v)|)$, para um valor fixo $h = \{1, \ldots, H\}$, a complexidade é igual a $O(\sum_{v \in V} |\delta(v)|)$. Sabe-se que em um grafo a soma dos graus dos nós é igual a duas vezes o número de arestas e, em uma árvore T, o número de arestas $|A| = n - 1$. Portanto, para um valor fixo h, o problema pode ser resolvido em tempo $O(n)$. Como $1 \leq h \leq H$, a complexidade do algoritmo é $O(nH)$.

O pseudocódigo do algoritmo de programação dinâmica top-down é apresentado no Algoritmo 4. Na linha 2 é verificado se o solução f_k^H referente ao subproblema $F(k, H)$ já foi computada, ou seja, se f_k^H pertence ao conjunto Ψ de subproblemas já computados, de modo que o valor da solução é retornada na linha 3. As linhas 5 a 10 calculam os casos base do problema, onde a linha 5 retrata o caso em que $H < 0$, que indica que a solução é inviável, a linha 7 o caso onde $k = n + 1$, ou seja, todos os nós já foram considerados, enquanto a linha 9 o caso no qual $p_{\pi_k} \in X$. Já as linhas 11 a 19 computam os quatro casos gerais, de modo que a linha 11 representa o caso em que o nó não é mandatório e nem XOR ou OR; a linha 13 o caso onde o nó é mandatório, mas não é XOR ou OR; a linha 15 o caso em que o nó é mandatório e XOR ou OR e, finalmente, a linha 17 o caso no qual o nó não é mandatório, mas é XOR ou OR. A solução f_k^H computada é então adicionada ao conjunto Ψ de soluções na linha 20 e por fim seu valor é retornado na linha 21.

Algoritmo 4 Pseudocódigo: $F(k, H)$

```
1: function F(k,H)
2:     if ∃⟨k, H, f_k^H⟩ ∈ Ψ then
3:         return f_k^H
4:     end if
5:     if H < 0 then
6:         f_k^H ← −∞
7:     else if k = n + 1 then
8:         f_k^H ← 0
9:     else if p_{π_k} ∈ X then
10:        f_k^H ← F(k + |Δ(π_k)|, H)
11:    else if π_k ∉ M ∧ π_k ∉ {O ∪ X} then
12:        f_k^H ← max(F(k + |Δ(π_k)|, H), b_{π_k} + F(k + 1, H − c_{π_k}))
13:    else if π_k ∈ M ∧ π_k ∉ {O ∪ X} then
14:        f_k^H ← b_{π_k} + F(k + 1, H − c_{π_k})
15:    else if π_k ∈ M ∧ π_k ∈ {O ∪ X} then
16:        f_k^H ← max_{(π_y∈Δ(π_k)}(b_{π_k} + b_{π_y} + F(y + 1, H − c_{π_k} − c_{π_y}))
17:    else if π_k ∉ M ∧ π_k ∈ {O ∪ X} then
18:        f_k^H ← max(F(k+|Δ(π_k)|, H), max_{(π_y∈δ(π_k)}(b_{π_k}+b_{π_y}+F(y+1, H−c_{π_k}−c_{π_y}))
19:    end if
20:    Ψ ← Ψ ∪ {⟨k, H, f_k^H⟩}
21:    return f_k^H
22: end function
```

3.4 Experimentos Computacionais

Os experimentos computacionais relatados nesta seção avaliam o desempenho do algoritmo de *branch-and-bound* do CPLEX com base na formulação (3.1) - (3.7) e o algoritmo de programação dinâmica da Seção 3.3. Nos referimos a esses algoritmos como B&B e DP, respectivamente. Esses algoritmos foram implementados em C++ e compilados com o GNU *gcc* versão 6.3. A formulação de PLI foi resolvida pelo IBM CPLEX versão 12.6, com configurações de parâmetro padrão. Todas os experimentos foram realizadas em um único núcleo de uma máquina Intel Xeon com 2.00 GHz de velocidade de clock e 16 GB de memória RAM.

O conjunto de instâncias utilizado nos experimentos foi proposto em Pereira et al. [2017]. Ele é dividido em 12 conjuntos de instâncias com o número de componentes variando de 200 a 10.000 nós. Cada conjunto de instâncias é baseado em um único modelo de componentes, que foi gerado usando o método de Thüm Thüm et al. [2014] para se parecer com modelos de componentes realistas. As características dessas instâncias são apresentadas na Tabela 3.1. A primeira coluna identifica o conjunto de

instâncias, enquanto a segunda exibe o número de nós ($|V|$) no modelo de componentes. A terceira coluna mostra o número de nós mandatórios ($|M|$), enquanto a quarta e a quinta coluna exibem o número de nós XOR ($|X|$) e OR ($|O|$), respectivamente. Cada conjunto contém 10 instâncias que diferem entre si pelo valor de b_i e c_i, que foram gerados aleatoriamente no intervalo $[0, 5]$ e $[0, 1000]$, respectivamente. Essa referência possui os maiores modelos de componentes da literatura sobre PCP, e o algoritmo de *backtracking* de [Pereira et al., 2017] não foi capaz de encontrar a solução ótima para nenhuma dessas instâncias dentro de 3600 segundos de tempo de execução.

Tabela 3.1: Características das instâncias utilizadas nos experimentos.

conjunto	|V|	|M|	|X|	|O|
FM11	200	60	7	9
FM12	200	49	10	13
FM13	500	147	18	29
FM14	500	152	21	22
FM15	1000	306	38	52
FM16	1000	319	46	37
FM17	2000	604	81	93
FM18	2000	586	90	101
FM19	5000	1466	213	231
FM20	5000	1486	252	216
FM21	10000	2918	446	464
FM22	10000	3038	463	440

No primeiro experimento, avaliamos o desempenho dos algoritmos B&B e DP para três valores diferentes de H: $C \times 1\%$, $C \times 5\%$ e $C \times 10\%$, onde $C = \sum_{i \in V} c_i$ é a soma do custo de todos os componentes no modelo de componentes. Os resultados são relatados na Tabela 3.2. A primeira coluna identifica o conjunto de instâncias. A segunda, terceira e quarta coluna exibem o tempo médio de execução, nas 10 instâncias de cada conjunto, e o respectivo desvio padrão (entre parênteses) do algoritmo de *branch-and-bound* do CPLEX para as instâncias com $H = C \times 1\%$, $H = C \times 5\%$ e $H = C \times 10\%$, respectivamente. Os mesmos dados são relatados para o algoritmo de programação dinâmica nas últimas três colunas.

Pode ser observado na Tabela 3.2 que ambos os algoritmos foram capazes de resolver de maneira ótima todas as instâncias, independentemente do valor de H. O tempo médio de execução máximo do algoritmo de B&B foi de 197.06 segundos para as instâncias no conjunto FM21 com $H = C \times 5\%$, enquanto o do DP foi de 306.52 segundos para as instâncias no conjunto FM22 com $H = C \times 5\%$. Pode-se observar que os tempos médios de execução do DP sempre foram menores ou iguais aos do B&B nas

instâncias com $H = C \times 1\%$ e $H = C \times 5\%$ e que acontece o contrário para aqueles com $H = C \times 10\%$. Além disso, os desvios padrão dos tempos de execução do algoritmo DP são significativamente menores que os do B&B, independentemente do valor de H.

Tabela 3.2: Tempo de execução e o respectivo desvio padrão (entre parênteses) do algoritmo de B&B e do algoritmo de programação dinâmica nos doze conjuntos de instâncias

conjunto	Branch-and-bound (B&B)			Programação Dinâmica (DP)		
instâncias	1%	5%	10%	1%	5%	10%
FM11	0.01 (0.00)	0.01 (0.00)	0.01 (0.00)	0.01 (0.00)	0.00 (0.00)	0.01 (0.00)
FM12	0.01 (0.00)	0.07 (0.03)	0.11 (0.03)	0.01 (0.00)	0.01 (0.00)	0.04 (0.00)
FM13	0.01 (0.00)	0.02 (0.01)	0.03 (0.01)	0.01 (0.00)	0.02 (0.00)	0.04 (0.00)
FM14	0.01 (0.00)	0.02 (0.00)	0.02 (0.00)	0.01 (0.00)	0.02 (0.00)	0.04 (0.00)
FM15	0.50 (0.29)	1.44 (0.74)	1.61 (0.82)	0.02 (0.00)	0.83 (0.04)	1.93 (0.07)
FM16	0.01 (0.00)	1.52 (0.48)	1.49 (0.82)	0.01 (0.00)	0.74 (0.04)	1.79 (0.05)
FM17	2.27 (0.62)	7.00 (4.08)	4.86 (3.77)	0.25 (0.05)	3.89 (0.13)	8.62 (0.20)
FM18	0.03 (0.00)	4.79 (3.32)	6.84 (4.81)	0.37 (0.02)	4.14 (0.10)	9.38 (0.16)
FM19	0.10 (0.00)	42.26 (33.13)	44.52 (31.28)	0.50 (0.02)	23.40 (0.76)	59.49 (0.65)
FM20	4.02 (1.11)	38.58 (16.34)	27.90 (17.95)	1.85 (0.19)	29.76 (0.83)	65.32 (1.77)
FM21	6.89 (5.94)	197.06 (122.45)	145.08 (156.58	6.71 (0.58)	127.80 (4.84)	304.18 (9.69)
FM22	34.11 (9.61)	191.05 (112.38)	143.59 (125.52)	16.82 (0.69)	142.98 (2.78)	306.52 (6.02)

O objetivo do próximo experimento é identificar para qual valor de H os tempos de execução do algoritmo DP são menores que os do B&B. Como a complexidade computacional do algoritmo DP é $O(nH)$, espera-se que seu tempo de execução cresça linearmente com o valor de H, o que não é necessariamente o caso do B&B. Para esse fim, no próximo experimento, fixamos o tamanho do modelo de componentes e variamos o valor de H de $C \times 1\%$ a $C \times 10\%$ em incrementos de $C \times 1\%$. Os resultados para as instâncias no conjunto FM22 são apresentados na Figura 3.3, que mostra o tempo médio de execução, nas 10 instâncias do conjunto, para cada valor de H. Pode-se ver nesta figura que o tempo médio de execução do DP foi mais rápido que o do B&B para as instâncias com $H \leq C \times 5\%$. Para valores maiores de H, B&B foi o algoritmo mais eficiente.

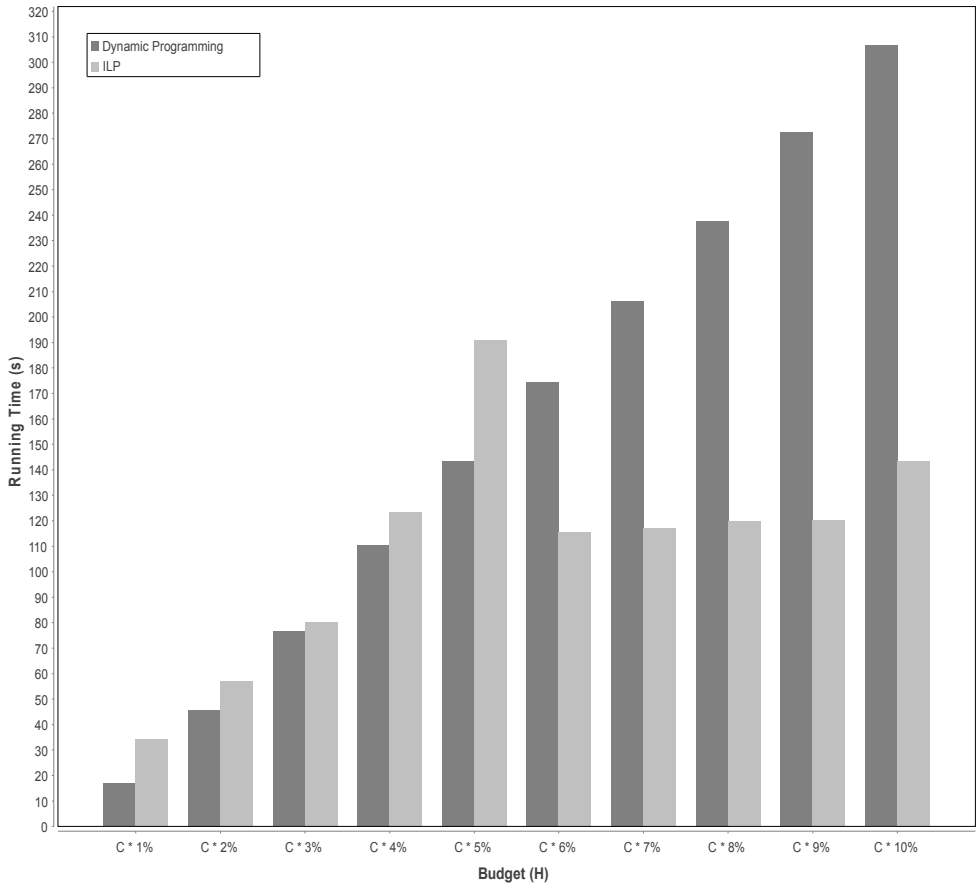

Figura 3.3: Tempos de execução do B &B e DP para as instâncias no conjunto FM22 com H variando de $C \times 1\%$ a $C \times 10\%$ em incrementos de $C \times 1\%$, em que $C = \sum_{i \in V} c_i$.

3.5 Considerações Finais

Neste capítulo, apresentamos uma variante do algoritmo de programação dinâmica de Cho & Shaw [1997] para o TKP, cuja complexidade computacional é $O(nH)$. Em seguida, generalizamos esse novo algoritmo para o PCP. O último tem a mesma complexidade de pior caso de $O(nH)$. Também fornecemos uma formulação de programação linear inteira para o PCP, que foi resolvida pelo algoritmo de *branch-and-bound* do CPLEX. Experimentos computacionais realizados mostraram que os algoritmos B&B e DP apresentaram resultados superiores à heurística BRKGA de Pereira et al. [2017], sendo capazes de encontrar soluções ótimas para todas as instâncias avaliadas com até 10.000 componentes em um tempo médio inferior a 306.52 segundos, independentemente do valor de H, enquanto o tempo computacional médio da heurística BRKGA foi de 2083.8 segundos. Já o algoritmo exato de *backtracking* de [Pereira et al., 2017] não foi capaz de encontrar a solução ótima para nenhuma dessas instâncias dentro de 3600 segundos de tempo de execução. Ademais, os resultados numéricos também mostraram que quanto menor é o valor de H, mais eficiente é o algoritmo DP em relação a B&B e vice-versa. Trabalhos futuros podem investigar o politopo da formulação com o intuito de descobrir desigualdades que definem facetas. Outra possibilidade seria desenvolver algoritmos exatos que não são baseados em técnicas de programação linear inteira, como Programação por Restrições.

Capítulo 4

O Problema do Corte Guilhotinado Bidimensional em Três Estágios com Restrições de Precedência em Lote

Muitas indústrias enfrentam o desafio de encontrar as soluções mais econômicas para o problema de cortar objetos maiores para produzir objetos menores específicos. Esses problemas de corte são frequentemente encontrados em diferentes processos industriais, como cortes de chapas de aço e de vidro, entre outros.

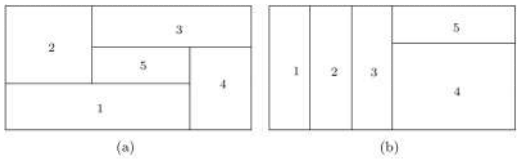

Figura 4.1: Exemplo de um corte não guilhotinado (a) e guilhotinado (b).

Nesta tese, nos referimos aos objetos maiores como placas e aos objetos menores como itens, e assumimos que eles são bidimensionais e têm uma forma retangular. Além disso, uma restrição comum para problemas de corte é que os itens devem ser cortados apenas com cortes guilhotinados, ou seja, cortes paralelos a um dos lados do objeto e que vão de um lado para o lado oposto. A Figura 4.1 mostra um exemplo de (a) cortes não guilhotinados e (b) guilhotinados.

Outra restrição frequente para esse tipo de problema é a necessidade de usar cortes em etapas. Um estágio de corte consiste em um conjunto de cortes guilhotinados

paralelos, e um corte em k estágios é uma sequência de no máximo k estágios de corte. Em cada etapa, os cortes são feitos na direção perpendicular à etapa anterior, sendo executados inicialmente verticalmente. A notação α-cut é usada para indicar o número do estágio em que o corte foi feito. A Figura 4.2(b) mostra um exemplo de um corte em 2 estágios.

Um padrão de corte é uma descrição de como uma placa é cortada para separar um subconjunto de itens. O padrão de corte pode ser representado como uma árvore, onde o nó raiz corresponde à placa inicial e os outros nós representam partes da placa que foram separadas com cortes guilhotinados. Portanto, as folhas representam os itens e os pedaços de placa individuais não utilizados. Cada nível da árvore corresponde a um estágio, ou seja, todos os cortes no mesmo nível têm a mesma orientação. Na prática, os cortes são executados em estágios para evitar a alteração do segmento da placa que está na guilhotina. Por exemplo, a representação em árvore do padrão de corte em 3 estágios da Figura 4.2 é fornecida na Figura 4.3. Primeiro, é aplicado um corte 1-cut vertical para separar um pedaço de placa não utilizado (destacado com linhas diagonais). Em seguida, é realizado um corte horizontal 2-cut para extrair o item 1. Por fim, três cortes verticais sucessivos 3-cuts são executados para obter os itens 2, 3 e 4, além de outro pedaço de placa não utilizado.

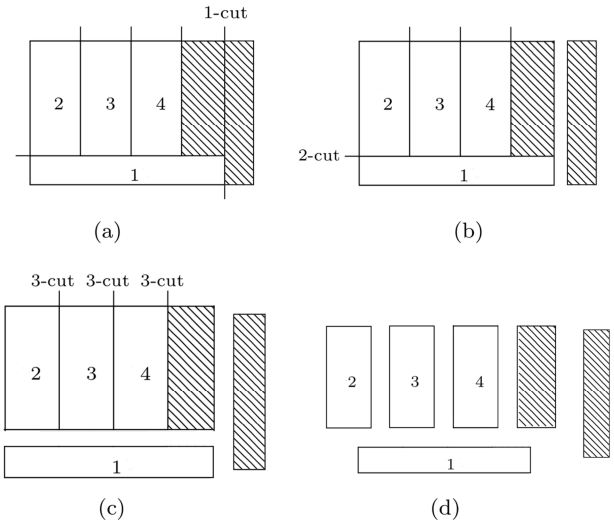

Figura 4.2: Um padrão de corte (a), seguido pelo primeiro (b), segundo (c) e terceiro (d) estágios de corte.

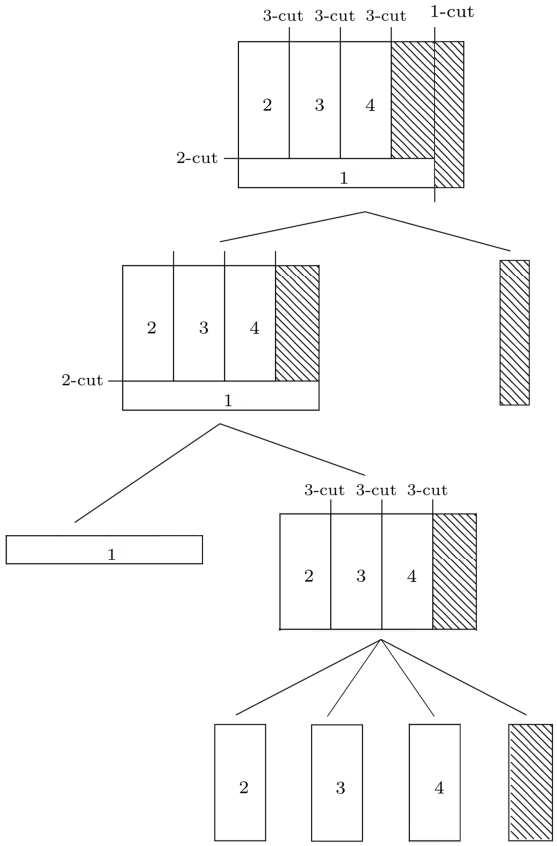

Figura 4.3: Exemplo do padrão de corte da Figura 4.2.

A diferença entre o problema abordado nesta tese e outros na literatura é que os itens são organizados em lotes. Cada lote representa um pedido do cliente e define a ordem parcial na qual os itens devem ser cortados. Ou seja, se o item i precede o item j em um lote, i deve ser cortado antes de j. Não há restrição de precedência entre itens em lotes distintos. Essa restrição vem de uma aplicação real do setor referente a um corte guilhotinado de três estágios em placas de vidro Lydia & Quentin [2018]. Como as peças de vidro são frágeis, elas devem ser empilhadas e enviadas na ordem exata em que são usadas pelo cliente, para evitar rachaduras ao acessar as peças de vidro no meio da pilha. A mesma restrição de precedência pode ser encontrada no gerenciamento da cadeia de suprimentos de peças metálicas, devido ao grande peso dessas peças.

Nesta tese, trabalhamos com o problema do corte guilhotinado bidimensional sujeito a restrições de guilhotina, estágio e precedência, denominado o Problema do Corte Guilhotinado Bidimensional em Três Estágios com Restrições de Precedência em Lote (do inglês, *2-Dimensional Cutting Stock Problem with Batch Constraints* - 2DCSP-BC). Seja W e H a largura e a altura das placas disponíveis para corte, e I o

conjunto de itens a serem cortados, onde cada item $i \in I$ tem largura w_i e altura h_i. Seja também S o conjunto de pilhas que representam os pedidos dos clientes, onde cada pilha $s = (\pi_1^s, \pi_2^s, \ldots, \pi_{n_s}^s)$ descreve a ordem na qual os itens devem ser cortados, de modo que o item $\pi_k^s \in I$ seja cortado antes do item $\pi_{k+1}^s \in I$, para todo $k = 1, \ldots, n_s - 1$, onde n_s é o número de itens em s. Uma solução para o 2DCSP-BC consiste em uma sequência de padrões de corte P que descreve como e em que ordem as placas devem ser cortadas. Esta solução deve atender a todas as seguintes restrições: (i) a placa não pode ser girada; (ii) os itens só podem ser rotacionados em 90°; (iii) todos os itens em I devem ser cortados exatamente uma vez; (iv) se $i \in I$ preceder $j \in I$ em sua pilha, i deve ser completamente separado de uma placa antes de j; (v) apenas cortes guilhotinados são permitidos; e (vi) o número de estágios de corte é no máximo 3. No entanto, como em [Hifi & Roucairol, 2001; Yanasse & Morabito, 2006; Alvarez-Valdes et al., 2007; Andrade et al., 2016], é permitido um único corte 4-cut adicional apenas para separar um único item de um pedaço de placa não utilizado. Isso é conhecido na literatura como *trimming*. A Figura 4.4 mostra um exemplo de *trimming*, onde o item 3 é separado de um pedaço de placa não utilizado por um único 4-cut, enquanto a Figura 4.4b dá um exemplo de um corte 4-cut inválido usado para separar o item 3 do item 4.

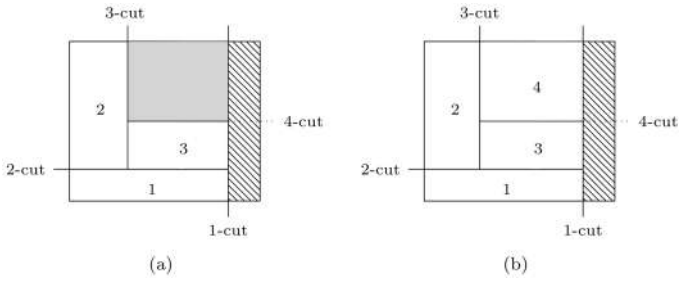

Figura 4.4: Exemplo de um corte 4-cut permitido (a) e não permitido (b).

O custo de uma solução de P do 2DCSP-BC é definido como a quantidade de material usado para cortar todos os itens. Como em [Cherri et al., 2009; Chen et al., 2015; Andrade et al., 2016; Cui et al., 2017], os pedaços de placa não utilizados que resultam dos padrões de corte em P são divididos em dois tipos. O chamado *resíduo* é o material à direita do último corte do tipo *1-cut* aplicado na última placa $|P|$. Supõe-se que este pedaço de placa possa ser reutilizado e não seja considerado no custo de P. Todos os outros pedaços de placa não utilizados são considerados *desperdícios*, porque é assumido que eles não podem ser reutilizados. Por exemplo, na Figura 4.4a, o pedaço de placa não utilizado destacado em cinza é considerado desperdício, enquanto

o destacado com linhas diagonais é considerado resíduo. Portanto, o custo $f(P)$ de uma solução P é definido de acordo com a equação (4.1),

$$f(P) = \quad H \times W \times (|P| - 1) + H \times r(P) \tag{4.1}$$

onde $|P|$ indica o número de placas usadas, $H \times W \times (|P| - 1)$ é a área total das primeiras $|P| - 1$ placas, $r(P)$ é a posição do último *1-cut* na última placa de P, e $H \times r(P)$ representa a área usada da última placa.

2DCSP-BC consiste então em encontrar uma solução $P^* = \text{argmin}_{P \in \Delta} f(P)$ que usa a menor quantidade de material para cortar todos os itens em I, onde Δ é o conjunto de soluções viáveis. Quando existe apenas um item por pilha, esse problema é equivalente ao Problema do Corte Guilhotinado Bidimensional (do inglês, *2-Dimensional Cutting Stock Problem* - 2DCSP) [Lai & Chan, 1997]. Portanto, ele também é NP-difícil.

Como não existe uma técnica conhecida para projetar um algoritmo exato de tempo polinomial para problemas NP-difíceis, esta tese se concentra em algoritmos heurísticos. No entanto, tanto quanto podemos dizer, a nova restrição de precedência introduzida em [Lydia & Quentin, 2018] impede o uso da maioria dos algoritmos na literatura do 2DCSP, visto que eles não foram projetados para considerar a precedência de itens. Além disso, exceto nas heurísticas construtivas FFF [Puchinger et al., 2004] e SHP [Suliman, 2006], não é evidente como adaptar os algoritmos heurísticos e exatos mais eficientes para levar em consideração a precedência. Portanto, nesta tese, estendemos FFF e SHP para 2DCSP-BC e propomos uma heurística baseada no algoritmo de programação dinâmica desenvolvido para o Problema da Mochila Retangular com Restrições em Lote (do inglês, *Rectangular Knapsack Problem with Batch Constraints* - RKP-BC).

O restante deste capítulo está organizado da seguinte forma. Na seção 4.1 apresentamos os trabalho relacionados. Na Seção 4.2 descrevemos as duas heurísticas construtivas extendidas para o 2DCSP-BC. Na Seção 4.3 apresentamos o algoritmo exato de programação dinâmica desenvolvido para o RKP-BC, e em seguida descrevemos na Seção 4.4 a heurística para o 2DCSP-BC que utiliza como subrotina o algoritmo exato de programação dinâmica desenvolvido para o RKP-BC. Por fim, na Seção 4.5 exibimos os experimentos computacionais com instâncias obtidas no *ROADEF Challenge 2018* e apontamos possíveis trabalhos futuros.

4.1 Trabalhos Relacionados

Até o presente momento, pelo nosso conhecimento nenhum trabalho na literatura trata especificamente o problema definido no Capítulo 4. Dessa forma, a revisão bibliográfica realizada considera problemas de corte bidimensional guilhotinado em estágios tradicionais. Em [Puchinger et al., 2004], uma heurística *Finite First-Fit* (FFF) foi apresentada para o Problema do Corte Guilhotinado Bidimensional em Três Estágios. Além disso, duas estratégias baseadas em *Branch-and-Bound* foram propostas para resolver o problema. Algoritmos genéticos que usam técnicas de recombinação e mutação para gerar soluções viáveis para o problema também foram propostos. Os experimentos realizados mostraram que os algoritmos genéticos obtiveram melhores resultados.

Um *Sequential Heuristic Procedure* (SHP) foi proposto em [Suliman, 2006] para resolver o 2DCSP, sendo este procedimento composto de 3 etapas. Inicialmente a primeira etapa seleciona a altura do corte e, em seguida, na segunda etapa o comprimento do corte é determinado. Por fim, na terceira etapa é decidido o número de vezes que o padrão de corte gerado será utilizado. Os autores argumentam que o desempenho da heurística proposta é melhor do que das heurísticas que utilizam padrões de corte pré-determinados.

Uma variante de 2DCSP na qual as placas podem conter defeitos e variar em tamanho foi abordada em [Jin et al., 2015]. Como não há restrições de precedência, a heurística proposta atribui pesos baseados na largura de cada item a ser cortado, de forma que os itens com maior largura tem maior prioridade na fila de corte. O algoritmo tenta posicionar os itens com maior largura nas placas menores e, após todos os itens serem posicionados, verifica se algum item foi colocado em alguma área defeituosa. Nesse caso, o item é removido e a possibilidade de ser adicionado a qualquer uma das placas já utilizadas é verificada. Experimentos computacionais mostraram que essaa heurística obteve melhores resultados do que os apresentados em [Vassiliadis, 2005].

O problema de corte bidimensional guilhotinado sem limitação de estágios foi estudado em [Furini et al., 2016] e uma formulação em Programação Linear Inteira foi proposta para resolvê-lo. O modelo proposto possui um número pseudo-polinomial de variáveis e restrições. A abordagem proposta foi capaz de resolver de forma ótima instâncias com até 30 itens.

Nenhum trabalho relacionado descrito acima lida com a restrição de precedência. Além disso, exceto nas heurísticas construtivas FFF [Puchinger et al., 2004] e SHP [Suliman, 2006], não é evidente como adaptar os algoritmos heurísticos e exatos mais avançados para levar em consideração a precedência da forma que definimos no 2DCSP-BC. Portanto, nas seções seguintes, projetamos algoritmos heurísticos que lidam com

a precedência de itens.

4.2 Heurísticas Construtivas

Nesta seção, apresentamos duas heurísticas construtivas para 2DCSP-BC, que são baseadas nas heurísticas FFF [Puchinger et al., 2004] e SHP [Suliman, 2006]. Todavia, em vez de descrever essas heurísticas utilizando a representação em árvore usual de uma solução, adotamos uma nova representação com base no que chamamos de representação *k-box*. Uma *0-box* por definição é uma placa. Portanto, sempre tem altura H e largura W. Essa *box* é dividida em uma sequência de *1-boxes*. Portanto, uma *1-box* sempre tem altura H, mas pode ter largura variável. Supõe-se que as *1-boxes* sejam colocadas contiguamente da esquerda para a direita da respectiva *0-box*. Logo, toda a área da *0-box* não coberta por uma *1-box* é uma área não utilizada (resíduos ou desperdícios). Analogamente, uma *1-box* é dividida em uma sequência de *2-boxes*. Desse modo, uma *2-box* tem sempre a mesma largura de sua respectiva *1-box*, mas pode ter altura variável. Supõe-se que as *2-boxes* sejam colocadas contiguamente de baixo para cima na respectiva *1-box*. Logo, toda a área do respectivo 1-cut não coberto por uma *2-box* é um desperdício. Da mesma forma, uma *2-box* é dividida em uma sequência de *3-boxes*, cada uma associada a exatamente um item. Como 2DCSP-BC é definido em 3 estágios, cada *3-box* corresponde a um item. Portanto, uma *3-box* tem sempre a mesma altura da respectiva *2-box*, mas sua largura é exatamente igual à do item correspondente. Vale ressaltar que, a partir da largura da *3-box*, pode-se inferir se o item correspondente foi rotacionado ou não. Além disso, está implícito que, se a altura de uma *3-box* for maior que a do item correspondente, ocorrerá um corte *trimming*. Assume-se também que as *3-boxes* são colocadas contiguamente da esquerda para a direita de suas respectivas *2-boxes*. Portanto, toda a área do respectivo 2-cut não coberto por uma *3-box* é um desperdício. Ressaltamos o fato de que existe uma correspondência direta entre uma *k-box* e um nó de nível k na árvore de padrões de corte. A vantagem da representação *k-box* de uma solução do 2DCSP é que ela é mais intuitiva e não é necessário levar em conta a posição exata dos cortes, mas apenas o tamanho das *k-boxes*.

4.2.1 Heurística *Finite First-Fit* (FFF)

Nesta seção, estendemos a heurística FFF de [Puchinger et al., 2004] para lidar com restrições de precedência. Primeiro, conforme sugerido em Puchinger et al. [2004], FFF gira os itens para que eles sejam orientados horizontalmente, de modo que sua largura

seja maior ou igual à sua altura, e nenhuma outra rotação é realizada. Em seguida, a cada iteração, FFF remove um item do topo de uma pilha e o insere em uma solução parcial, até que todos os itens sejam inseridos. Os itens são selecionados de acordo com um critério guloso c_i. Três valores para c_i são avaliados: (i) a largura do item, (ii) a área do item e (iii) um número gerado aleatoriamente. Nos referimos a essas variantes de FFF como FFF-W, FFF-A, e FFF-R. A abordagem usada para decidir onde colocar o item na solução é descrita pelo Algoritmo 5.

A entrada do Algoritmo 5 é uma tupla $\langle W, H, I, S\rangle$ contendo as dimensões das placas e o conjunto de pilhas de itens a serem cortados. A saída correspondente é uma sequência de padrões de corte P que descreve como cortar todos os itens em I. Denotamos por $T(S) \subseteq I$ como o conjunto de itens que estão no topo das pilhas não vazias de S e por $s(i) \in S$ como a pilha que contém o item $i \in I$. Além disso, representamos uma *k-box* b^k como uma lista de $(k+1) - boxes$ e denotamos $w(b^k)$ e $h(b^k)$, respectivamente, como a largura e a altura de b^k. Vale salientar que para um valor par de k, $w(b^k)$ é a soma da largura de todas as $(k+1) - boxes$ em b^k e $h(b^k)$ é a altura do item mais alto em b^k, enquanto para um valor ímpar de k, $h(b^k)$ é a soma da altura de todas as caixas $(k+1) - boxes$ em b^k e $w(b^k)$ é a largura do item mais largo em b^k.

Na primeira linha do Algoritmo 5, uma solução parcial vazia P é inicializada. A cada iteração do loop das linhas 2 a 16, o padrão de corte de uma placa é decidido. Esse loop é repetido até que todos os itens estejam na solução P. Uma *0-box* vazia b^0 é inicializada na linha 3 e, a cada iteração do loop das linhas 4 a 14, uma nova *1-box* é adicionada a b^0. Esse loop é repetido até que nenhum item em $T(S)$ caiba no espaço restante em b^0. O item i com o maior valor de c_i, que cabe no espaço restante de b^0, é identificado na linha 4. Conforme sugerido por [Puchinger et al., 2004], a largura deste primeiro item define a largura da *1-box* b^1, inicializada na linha 5 com uma única *2-box* contendo apenas i. Na mesma linha, i é retirado da pilha, pois já está na solução. A cada iteração do loop das linhas 6 a 12, uma nova *2-box* é adicionada a b^1. Esse loop é repetido até que nenhum item em $T(S)$ caiba no espaço restante em b^1. O item j com o maior valor de c_j, que cabe no espaço restante de b^1, é identificado na linha 6. Como também sugerido por [Puchinger et al., 2004], a altura desse item define o altura da *2-box* b^2, inicializada na linha 7 com uma única *3-box* contendo apenas j. Na mesma linha, j é retirado de sua pilha, pois já encontra-se na solução. A cada iteração do loop das linhas 8 a 10, uma nova *3-box* (contendo um único item por definição) é adicionada a b^2. Esse loop é repetido até que nenhum item em $T(S)$ caiba no espaço restante em b^2. O item k com o maior valor de c_k, que cabe no espaço restante de b^2, é identificado na linha 8 e sua respectiva *3-box* é adicionada a b^2 na linha 9. Na

mesma linha, k é retirado da pilha, pois já está na solução. Quando não for possível adicionar mais itens a b^2, este é adicionado a b^1 na linha 11. Analogamente, quando não for possível adicionar mais *2-boxes* a b^1, este é adicionado a b^0 na linha 13. Da mesma forma, quando não for possível adicionar mais *1-boxes* a b^0, este é adicionado à solução P na linha 15. Em seguida, quando todos os itens estiverem em P, a solução P é retornada na linha 17.

Algoritmo 5 *Finite First-Fit* (FFF)

Entrada: $\langle W, H, I, S\rangle$
Saída: P

1: Seja $P \leftarrow [\,]$
2: **while** $T(S) \neq \emptyset$ **do**
3: Seja $b^0 \leftarrow [\,]$
4: **while** $i \neq \emptyset$, onde $i \leftarrow \text{argmax}_{i \in T(S): w_i \leq W - w(b^0) \wedge h_i \leq H} c_i$ **do**
▷ `Inicialize a 1-box` b^1.
5: Seja $b^1 \leftarrow [[i]]$, $pop(s(i))$
▷ `Insere uma sequência de 2-box em` b^1.
6: **while** $j \neq \emptyset$, onde $j \leftarrow \text{argmax}_{j \in T(S): w_j \leq w(b^1) \wedge h_j \leq H - h(b^1)} c_j$ **do**
7: Seja $b^2 \leftarrow [j]$, $pop(s(j))$
8: **while** k $\neq \emptyset$, onde $k \leftarrow \text{argmax}_{k \in T(S): w_k \leq w(b^1) - w(b^2) \wedge h_k \leq h(b^2)} c_k$ **do**
9: $b^2 \leftarrow b^2 : k$, $pop(s(k))$
10: **end while**
11: $b^1 \leftarrow b^1 : b^2$
12: **end while**
13: $b^0 \leftarrow b^0 : b^1$
14: **end while**
15: $P \leftarrow P : b^0$
16: **end while**
17: **return** P

4.2.2 *Sequential Heuristic Procedure* (SHP)

A heurística SHP de [Suliman, 2006] também foi extendida para lidar com restrições de precedência. Essa heurística é semelhante à FFF e também rotaciona os itens para que eles sejam orientados horizontalmente. Além disso, os itens também são selecionados de acordo com um critério guloso c_i, e novamente avaliamos três valores para c_i: (i) a largura do item, (ii) a área do item e (iii) um número gerado aleatoriamente. Nos referimos a essas variantes de SHP como SHP-W, SHP-A, e SHP-R. No entanto, em vez de definir a largura das *1-boxes* como a largura do primeiro item adicionado, a heurística utiliza uma abordagem diferente, conforme descrita no Algoritmo 6.

A entrada do Algoritmo 6 é a tupla $\langle W, H, I, S\rangle$ contendo as dimensões das placas e o conjunto de pilhas com os itens a serem cortados. A saída correspondente é uma sequência de padrões de corte P que descreve como cortar todos os itens em I.

Na primeira linha do Algoritmo 6, uma solução parcial vazia P é inicializada. A cada iteração do loop das linhas 2 a 20, o padrão de corte de uma placa é decidido. Esse loop é repetido até que todos os itens estejam na solução P. Uma *0-box* vazia b^0 é inicializada na linha 3 e, a cada iteração do loop das linhas 4 a 18, uma nova *1-box* é adicionada a b^0. Esse loop é repetido até que nenhum item em $T(S)$ caiba no espaço restante em b^0. O item i com o maior valor de c_i, que cabe no espaço restante em b^0, é identificado na linha 4. Como no FFF, uma *2-box* (aqui chamada $\bar{b}^2$) é inicializada na linha 5, contendo apenas i, sendo i posteriormente retirado de sua pilha. No entanto, o loop das linhas 6 a 8 adiciona itens adicionais a $\bar{b}^2$, enquanto $W > (w(b^0) + w(\bar{b}^2))$. Conforme sugerido por [Suliman, 2006], a largura de $\bar{b}^2$ define a largura de b^1, que é inicializada na linha 9 com apenas $\bar{b}^2$. A cada iteração do loop das linhas 10 a 16, uma nova *2-box* é adicionada a b^1. Quando não for possível adicionar mais *2-boxes* a b^1, este é adicionado a b^0 na linha 17. Da mesma forma, quando não for possível adicionar mais *1-boxes* a b^0, este é adicionado para a solução P na linha 19. Por fim, quando todos os itens estiverem sido cortados, a solução P é retornada na linha 21.

Algoritmo 6 *Sequential Heuristic Procedure* (SHP)

Entrada: $\langle W, H, I, S\rangle$
Saída: P

1: Seja $P \leftarrow [\,]$
2: **while** $S \neq \emptyset$ **do**
3: Seja $b^0 \leftarrow [\,]$
4: **while** $i \neq \emptyset$, onde $i \leftarrow \text{argmax}_{i \in T(S): w_i \leq W - w(b^0) \wedge h_i \leq H} c_i$ **do**
▷ `Inicialize a 1-box` b^1.
5: Seja $\bar{b}^2 \leftarrow [i]$, $pop(s(i))$
6: **while** $\ell \neq \emptyset$, onde $\ell \leftarrow \text{argmax}_{\ell \in T(S): w_\ell \leq W - (w(b^0) + w(\bar{b}^2)) \wedge h_\ell \leq H} c_\ell$ **do**
7: $\bar{b}^2 \leftarrow \bar{b}^2 : \ell$, $pop(s(\ell))$
8: **end while**
9: Seja $b^1 \leftarrow [\bar{b}^2]$
▷ `Insere uma sequência de 2-box em` b^1.
10: **while** $j \neq \emptyset$, onde $j \leftarrow \text{argmax}_{j \in T(S): w_j \leq w(b^1) \wedge h_j \leq H - h(b^1)} c_j$ **do**
11: Seja $b^2 \leftarrow [j]$, $pop(s(j))$
12: **while** $k \neq \emptyset$, onde $k \leftarrow \text{argmax}_{k \in T(S): w_k \leq w(b^1) - w(b^2) \wedge h_k \leq h(b^2)} c_k$ **do**
13: $b^2 \leftarrow b^2 : k$, $pop(s(k))$
14: **end while**
15: $b^1 \leftarrow b^1 : b^2$
16: **end while**
17: $b^0 \leftarrow b^0 : b^1$
18: **end while**
19: $P \leftarrow P : b^0$
20: **end while**
21: **return** P

4.3 Algoritmo Exato de Programação Dinâmica para o RKP-BC

Nessa seção, propõe-se o Problema da Mochila Retangular com Restrições de Precedência em Lote e um um algoritmo exato pseudo-polinomial de programação dinâmica para este problema. Esse algoritmo é utilizado como uma subrotina da heurística para o Problema do Corte Guilhotinado Bidimensional em Três Estágios com Restrições de Precedência em Lote descrita na Seção 4.4.

O Problema da Mochila Retangular (do inglês, *Rectangular Knapsack Problem* - RKP) é definido sobre um conjunto de itens I, onde cada item representa um retângulo de largura $w_i \in \mathbb{N}$ e altura $h_i \in \mathbb{N}$, e está associado um benefício $b_i \in \mathbb{R}$. Entretanto, os itens podem ser rotacionados em exatamente 90°, caso desejado. Dada uma placa de largura W e altura H, o problema consiste em selecionar um subconjunto de itens $I' \subseteq I$ e posiciona-los lado a lado, da direita para a esquerda, na base da placa, de forma que a altura dos itens em I' não exceda a altura da placa e a soma dos comprimentos dos itens em I' não exceda a largura da placa. O objetivo é encontrar o subconjunto $I^* = \text{argmax}_{I' \in \Omega} \sum_{i \in I'} b_i$ cuja soma dos benefícios dos itens é máximo, onde Ω é o conjunto das soluções viáveis para o problema. Um exemplo de uma solução viável com 5 itens e benefício total igual a 100 é mostrado na Figura 4.5. Quando $w_i = h_i$ para todo $i \in I$, o RKP se reduz a um Problema da Mochila [Karp, 1972]. Portanto, ele também é NP-Difícil.

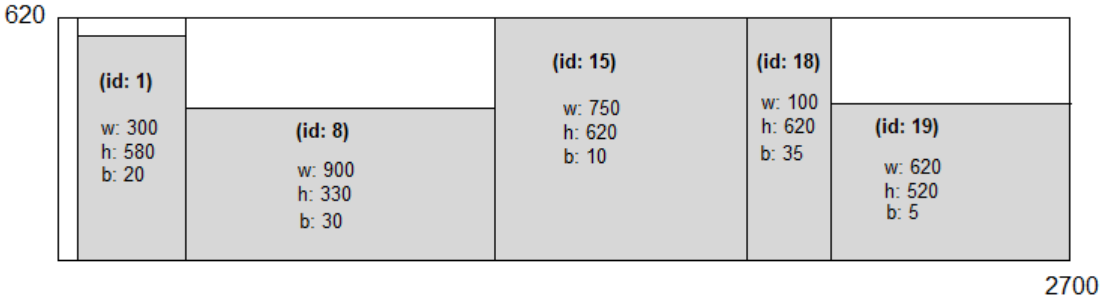

Figura 4.5: Exemplo de uma placa de uma instância do Problema da Mochila Retangular.

O Problema da Mochila Retangular com Restrições de Precedência em Lote é uma generalização do RKP onde o conjunto de itens está particionado em lotes, os quais representam o pedido de um cliente. Seja S um conjunto de pilhas que representam os lotes, onde cada pilha $s = (\pi_1^s, \pi_2^s, \ldots, \pi_{n_s}^s)$ descreve a precedência entre os itens no

lote da seguinte forma. Se um item π_k^s de uma pilha $s \in S$ for selecionado, todos os itens π_ℓ^s para $\ell = 1, \ldots, k-1$ devem ser selecionados e posicionados a esquerda de π_k^s. Novamente, o objetivo é encontrar o subconjunto $I^* \subseteq I$ cuja soma dos benefícios dos itens é máximo. Quando $n_s = 1$ para todo $s \in S$, o RKP-BC se reduz a um RKP. Portanto, ele também é NP-Difícil.

O RKP-BC pode ser resolvido da seguinte forma. Seja L um vetor resultante da concatenação de todos os lotes em S, i.e., L contém todos os itens de I, de forma que se $i \in I$ antecede $j \in I$ em algum lote, i antecede j em L. Seja também $\beta_i \in \mathbb{B}$ para todo $i \in I$, tal que $\beta_i = 1$ caso o item i é precedido por algum item em seu respectivo lote. Caso contrário, $\beta_i = 0$, i.e, i é o primeiro elemento do seu respectivo lote. Com isto, define-se a função $F(k, W, H, B)$, para $B \in \{0, 1\}$, de tal forma que $F(k, W, H, 0)$ é um subproblema de RKP-BC onde são considerados apenas os k primeiros itens do vetor L e uma placa de largura W e altura H, e $F(k, W, H, 1)$ se diferencia do subproblema anterior por uma única restrição adicional que obriga que o item $L[k]$ seja selecionado (independentemente de ser rotacionado ou não). Sendo assim, a solução ótima do RKP-BC pode ser obtida através da chamada de $F(|I|, W, H, 0)$.

$F(k, W, H, B)$ pode ser computada resolvendo-se a equação de recorrência (4.2). Para fins de uma notação mais clara, utiliza-se uma função auxiliar $feasible(e, t)$, tal que $feasible(e, t) = e$ se a expressão booleana t é verdadeira, e $feasible(e, t) = -\infty$ caso contrário. Esta função é utilizada para verificar as condições necessárias para a viabilidade dos subproblemas do caso geral de (4.2). O caso base de (4.2) ocorre quando $k = 0$, o que indica que não há mais itens a serem considerados, e consequentemente o benefício total deste subproblema é igual a zero. Já o caso geral consiste no máximo dentre benefício total da solução ótima dos três subproblemas de $F(k, W, H, B)$, descritos a seguir. Quando $B = 0$, o primeiro subproblema consiste em não selecionar o item $L[k]$ e resolver o problema original para os $k-1$ primeiros itens de L. Neste caso, a chamada recursiva é feita para $B = 0$, uma vez que o item $L[k-1]$ não precisa ser necessariamente selecionado já que $L[k]$ não foi selecionado. Quando $W - w_{L[k]} \geq 0$, o segundo subproblema consiste em posicionar o item $L[k]$, sem rotaciona-lo, a direita da solução ótima de um RKP-BC para os $k-1$ primeiros itens de L numa placa de largura $W - w_{L[k]}$ e altura H. Quando $W - h_{L[k]} \geq 0$ e $H - w_{L[k]} \geq 0$, o terceiro subproblema consiste em rotacionar o item $L[k]$ e posicioná-lo a direita da solução ótima de um RKP-BC para os $k-1$ primeiros itens de L numa placa de largura $W - h_{L[k]}$ e altura H. Nos dois últimos subproblemas, onde o item $L[k]$ é selecionado, a chamada recursiva é feita para $B = \beta_{L[k]}$, uma vez que neste caso, se $\beta_{L[k]} = 1$, $L[k-1]$ deve ser necessariamente selecionado. O mesmo não ocorre quando $\beta_{L[k]} = 0$, pois neste caso $L[k]$ é o primeiro item em seu lote e não é precedido por

nenhum outro item.

$$F(k,W,H,B) = \begin{cases} & (4.2) \\ 0 & \text{se } k = 0 \\ \max(feasible(F(k-1,W,H,0), \quad B=0), & \\ \quad feasible(b_{L[k]} + F(k-1, W-w_{L[k]}, H, \beta_{L[k]}), \ w_{L[k]} \leq W), & \\ \quad feasible(b_{L[k]} + F(k-1, W-h_{L[k]}, H, \beta_{L[k]}), \ h_{L[k]} \leq W \wedge w_{L[k]} \leq H)) & \text{c.c.} \end{cases}$$

O pseudocódigo do algoritmo de programação dinâmica top-down é apresentado no Algoritmo 7. Seja Ψ um conjunto que guarda o benefício total das soluções ótimas dos subproblemas já resolvidos pelo algoritmo. Na linha 2, verifica-se se o benefício total $f_k^{W,B}$ da solução ótima do subproblema $F(k,W,H,B)$ já foi computado. Neste caso, o valor da solução é retornado na linha 3. O *if* das linhas 5 a 7 retrata o caso base do problema, que ocorre quando $k = 0$. Neste caso, $f_k^{W,B} = 0$. Caso contrário, as linhas 8 a 16 computam o caso geral do problema, de modo que o valor de $f_k^{W,B}$ é inicializado com $-\infty$ na linha 8. O *if* das linhas 9 e 11 tratam o caso em que o item $L[k]$ não é selecionado, as linhas 12 a 14 tratam o caso em que $L[k]$ é selecionado sem rotacioná-lo, e as linha 15 a 17 tratam o caso onde $L[k]$ é selecionado de maneira rotacionada. O valor máximo de $f_k^{W,B}$ é adicionado ao conjunto Ψ na linha 19 e retornado na linha 20. Como o número de estados do problema é igual a $|I| \times W \times 2$, e cada estado pode ser computado em tempo $O(1)$, a complexidade assintótica do algoritmo é igual a $O(|I|W)$.

Algoritmo 7 Pseudocódigo: RKP-BC(k, W, H, B)

```
1: function F(k, W, H, B)
2:     if ∃⟨k, W, B, f_k^{W,B}⟩ ∈ Ψ then
3:         return f_k^{W,B}
4:     end if
5:     if k = 0 then
6:         f_k^{W,B} ← 0
7:     else
8:         f_k^{W,B} = −∞
9:         if B = 0 then
10:            f_k^{W,B} ← max(f_k^{W,B}, F(k − 1, W, H, 0))
11:        end if
12:        if W − w_k ≥ 0 then
13:            f_k^{W,B} ← max(f_k^{W,B}, b_{L[k]} + F(k − 1, W − w_{L[k]}, H, β_{L[k]}))
14:        end if
15:        if h_{L[k]} ≤ W ∧ w_{L[k]} ≤ H then
16:            f_k^{W,B} ← max(f_k^{W,B}, b_{L[k]} + F(k − 1, W − h_{L[k]}, H, β_{L[k]}))
17:        end if
18:    end if
19:    Ψ ← Ψ ∪ {⟨k, W, B, f_k^{W,B}⟩}
20:    return f_k^{W,B}
21: end function
```

4.4 Heurística baseada em PD para o 2DCSP-BC

Nesta seção é proposta uma nova heurística para 2DCSP-BC, chamada DPH. Ela é semelhante a FFF, mas uma vez definidas a largura e a altura de uma *2-box*, ela utiliza o Algoritmo 7 (descrito na seção anterior) para computar de forma ótima o subconjunto de itens que minimiza o desperdício desta *2-box*, satisfazendo todas as restrições de 2DCSP-BC.

DPH é descrita no Algoritmo 8. Ela recebe como entrada uma tupla $\langle W, H, I, S\rangle$ contendo as dimensões das placas e o conjunto de pilhas de itens a serem cortados. A saída correspondente é uma sequência de padrões de corte P que descreve como cortar todos os itens em I. A medida que os itens no topo das pilhas são posicionados nas placas, eles são removidos de sua respectiva pilha. Sendo assim, define-se $I(S) \subseteq I$ como o conjunto de itens que ainda estão em alguma pilha de S, e $H(S)$ como o conjunto das alturas dos itens de $I(S)$. Além disso, detona-se por $T(S) \subseteq I$ o conjunto de itens que estão no topo das pilhas não vazias de S.

Na primeira linha do Algoritmo 8, uma solução parcial vazia P é inicializada. A cada iteração do loop das linhas 2 a 14, o padrão de corte de uma placa é decidido. Esse loop é repetido até que todos os itens estejam na solução P. Uma *0-box* vazia b^0 é inicializada na linha 3 e, a cada iteração do loop das linhas 4 a 12, uma nova *1-box* é adicionada a b^0. Esse loop é repetido até que nenhum item em $L(S)$ caiba no espaço

restante em b^0. Conforme sugerido por [Puchinger et al., 2004], a largura do maior item em $T(S)$ define o largura da *1-box* b^1, inicializada na linha 5. A cada iteração do loop das linhas 6 a 10, uma única *2-box* é inserida em b^1. Para isso, são construídas $|H(S)|$ 2-*boxes* resolvendo-se um RKP-BC para os itens em S (com $b_i = w_i \times h_i$) numa placa de largura $w(b^1)$, e cada valor de altura $h' \in H(S)$. Na linha 7 , identifica-se a 2-*box* $b^2 = [I']$ com o maior aproveitamento de área ($\sum_{i \in I'} \frac{w_i \times h_i}{w(b^1) \times h'}$), i.e., àquela com o menor desperdício. Todos itens em I' são removidos de suas respectivas pilhas na linha 8, e b^2 é a adicionada a b^1 na linha 9. Esse loop é repetido até que nenhum item em $T(S)$ caiba no espaço restante em b^1. Quando não for possível adicionar mais *2-boxes* a b^1, este é adicionado a b^0 na linha 11. Da mesma forma, quando não for possível adicionar mais *1-boxes* a b^0, este é adicionado à solução P na linha 13. Em seguida, quando todos os itens estiverem em P, esta solução é retornada na linha 15.

Algoritmo 8 *Heurística de Programação Dinâmica* (DPH)

Entrada: $\langle W, H, I, S \rangle$
Saída: P

1: Seja $P \leftarrow [\,]$
2: **while** $T(S) \neq \emptyset$ **do**
3: Seja $b^0 \leftarrow [\,]$
4: **while** $i \neq \emptyset$, onde $i \leftarrow \text{argmax}_{i \in T(S): w_i \leq W - w(b^0) \wedge h_i \leq H} w_i$ **do**
▷ `Inicialize a 1-box` b^1.
5: Seja $b^1 \leftarrow [[i]]$, $pop(s(i))$
▷ `Insere uma sequência de 2-box em` b^1.
6: **while** $h' \neq \emptyset$, onde $h' \leftarrow \text{argmax}_{h' \in H(S)} \frac{F(|I(S)|, w(b^1), h', 0)}{w(b^1) \times h'}$ **do**
7: Seja $b^2 = [I']$, onde I' é a solução ótima de $F(|I(S)|, w(b^1), h', 0)$
8: pop(s(j)) $\forall j \in I'$
9: $b^1 \leftarrow b^1 : b^2$
10: **end while**
11: $b^0 \leftarrow b^0 : b^1$
12: **end while**
13: $P \leftarrow P : b^0$
14: **end while**
15: **return** P

4.5 Experimentos Computacionais

Nesta seção, são exibidos os resultados dos experimentos computacionais para avaliar o desempenho das heurística FFF, SHP e DPH. Todos as heurísticas desenvolvidas foram implementadas em C++ e compiladas com o GNU *gcc* versão 6.3. Os experimentos foram realizados em um único núcleo de uma máquina Intel Xeon com 2.00 GHz de velocidade de clock e 16 GB de memória RAM.

Três conjuntos de instâncias foram usados nos experimentos, os conjuntos de *Ins-*

tâncias A, *Instâncias B* e *Instâncias X*. Esses conjuntos de instâncias foram empregados no ROADEF Challenge 2018 e podem ser obtidos no site `http://www.roadef.org/challenge/2018/en/instances.php`. Adaptamos essas instâncias para o 2DCSP-BC ignorando os defeitos nas placas.

No experimento, avaliamos e comparamos a qualidade das soluções fornecidas pelas heurísticas FFF, SHP e DPH para as instâncias nos conjuntos A, B e X. Avaliamos três variantes de FFF: (*i*) FFF-Width (FFF -W); (*ii*) Área FFF (FFF-A); e (*iii*) FFF aleatório (FFF-R). Cada variante classifica os itens de acordo com uma regra diferente. O FFF-W classifica os itens de acordo com sua dimensão máxima decrescente, o FFF-A classifica os itens de acordo com a área decrescente e o FFF-R considera os itens em uma ordem aleatória. Também avaliamos três variantes para SHP: (*i*) SHP-Width (SHP-W); (*ii*) Área SHP (SHP-A); e (*iii*) SHP-Aleatório (SHP-R). Cada variante classifica os itens de acordo com uma regra diferente, que é igual aos de FFF-W, FFF-A e FFF-R, respectivamente. Como as heurísticas FFF-R, SHP-R e DPH usam escolhas aleatórias durante suas execuções, executamos essas heurísticas 5 vezes para cada instância utilizando sementes diferentes utilizando um gerador de números pseudo-aleatório. Para cada heurística, calculamos o *gap* de otimalidade médio relativo ($\frac{\text{H}-\text{LP}}{\text{H}}$) entre o custo H da solução fornecida por cada heurística e o limite inferior $\text{LB} = \sum_{i \in I} w_i h_i$, que representa a soma da área de todos os itens em I. As Tabelas 4.1, 4.2 e 4.3 fornecem, para cada instância dos conjuntos A, B e X, respectivamente, o nome da instância, além dos *gaps* de otimalidade obtidos pelas heurísticas FFF-W, FFF-A, FFF-R, SHP-W, SHP-A, SHP-R e DPH, seguido do tempo de execucação (em segundos) da heurística DPH. Todas as execuções das demais heurísticas tiveram um tempo de execucação inferior a 1 segundo.

Pode-se observar que entre as heurísticas construtivas FFF e SHP, as variantes FFF obtiveram intervalos de otimalidade relativos médios menores que as variantes SHP para todos os conjuntos de instâncias avaliados. Esses resultados podem dever-se ao fato de a SHP posicionar itens com a mesma largura máxima lado a lado. Assim, há uma pequena variação na largura dos itens nos estágios finais da heurística SHP, o que pode forçar um padrão de corte muito irregular. Entre as heurísticas da literatura FFF, os melhores resultados foram obtidos pela heurística FFF-W, onde seu *gap* de otimalidade médio relativo foi de 29,04 % para o conjunto de instâncias A, 20,77% para o conjunto de instâncias B e 23,34% para o conjunto de instâncias X. Já analisando apenas as heurísticas SHP, podemos observar que os melhores resultados foram também alcançados pela variante SHP-W, com um *gap* de otimalidade médio relativo de 35,45% para o conjunto de instâncias A, 28,87% para o conjunto de instâncias B e 32,70% para o conjunto de instâncias X. Quando comparamos as heurística FFF, SHP

e DPH, podemos observar que a heurística DPH obteve resultados superiores as demais heurísticas para os três conjuntos de instâncias, obtendo um *gap* de otimalidade médio relativo de 19,21% para o conjunto de instâncias A, 13,34% para o conjunto de instâncias B e 13,06% para o conjunto de instâncias X. Assim, a heurística DPH obteve um desperdício de área, com relação ao limitante inferior, 33,84% menor no conjunto de instâncias A, 35,77% menor no conjunto de instâncias B, e 44,04% menor no conjunto de instâncias X, quando comparada à heurística FFF-W. Todavia, o tempo de execucação da heurística DPH é superior às demais, com um tempo de execução máximo de 559.84 segundos na instância B13.

Tabela 4.1: Resultados das heurísticas FFF-W, FFF-A, FFF-R, SHP-W, SHP-A, SHP-R e DPH para o conjunto de instâncias A.

Instâncias	FFF-W	FFF-A	FFF-R	SHP-W	SHP-A	SHP-R	DPH	DPH t(s)
A1	10.87	10.87	10.87	10.87	10.87	10.87	8.61	1.48
A2	14.07	13.85	42.20	34.71	33.83	47.95	12.52	16.55
A3	27.39	26.80	40.16	38.09	38.43	42.24	15.41	20.88
A4	27.39	26.80	41.72	38.09	38.43	42.30	15.41	22.62
A5	31.29	23.90	38.42	36.25	37.24	42.33	11.63	30.59
A6	31.54	31.54	42.37	24.62	24.62	40.82	11.90	11.79
A7	31.74	31.97	42.46	40.61	40.61	46.00	14.94	21.33
A8	30.60	28.10	37.29	37.01	38.17	39.57	33.15	26.68
A9	40.42	40.42	46.35	36.61	37.66	47.43	16.48	19.82
A10	23.66	32.44	42.55	37.94	40.44	45.56	15.51	32.22
A11	23.89	29.60	39.30	32.13	32.88	38.99	11.85	30.41
A12	39.46	44.37	52.25	40.09	40.09	48.98	13.49	14.48
A13	16.97	16.97	32.03	32.21	32.21	36.75	39.10	81.49
A14	19.41	18.85	34.20	30.29	30.51	35.05	40.65	187.19
A15	21.27	22.25	33.46	34.34	33.25	36.16	34.43	204.36
A16	27.23	30.86	43.28	29.51	38.81	42.76	18.45	11.04
A17	46.17	54.83	52.14	46.17	57.21	51.21	23.30	7.79
A18	28.49	28.49	44.94	34.16	33.65	43.52	14.58	27.36
A19	34.88	34.88	44.99	36.64	36.64	45.62	18.45	15.46
A20	54.00	54.00	55.32	48.34	48.34	47.47	14.39	5.49
média:	29.04	30.09	40.81	34.93	36.19	41.58	19.21	39.45

Tabela 4.2: Resultados das heurísticas FFF-W, FFF-A, FFF-R, SHP-W, SHP-A, SHP-R e DPH para o conjunto de instâncias B.

Instâncias	FFF-W	FFF-A	FFF-R	SHP-W	SHP-A	SHP-R	DPH	DPH t(s)
B1	9.00	11.97	31.51	16.34	17.69	33.66	11.27	23.53
B2	21.23	21.23	34.06	42.79	43.38	43.37	11.17	201.81
B3	18.89	18.89	31.19	23.24	23.24	31.89	13.36	133.74
B4	21.88	21.98	33.79	27.19	31.68	34.90	11.84	76.93
B5	42.15	42.15	42.55	42.15	42.15	44.00	32.37	145.26
B6	23.85	23.85	33.48	30.02	29.23	34.52	13.49	59.78
B7	8.08	7.18	29.02	29.96	29.98	35.80	5.08	27.70
B8	19.18	19.18	30.35	22.96	22.96	29.50	12.81	229.02
B9	17.70	14.96	31.57	18.43	18.05	31.16	13.00	101.20
B10	22.67	22.67	30.12	27.97	27.97	33.20	15.18	66.23
B11	19.09	21.35	34.15	23.01	22.80	33.63	13.97	70.23
B12	21.96	21.96	34.22	35.70	35.65	38.01	10.04	226.93
B13	19.75	22.17	32.11	30.74	30.75	33.89	10.65	559.84
B14	26.52	23.72	35.76	38.86	38.76	43.84	11.73	56.43
B15	19.55	19.18	29.61	25.89	26.40	30.62	14.17	188.42
média:	20.77	20.83	32.90	29.02	29.38	35.47	13.34	144.47

Tabela 4.3: Resultados das heurísticas FFF-W, FFF-A, FFF-R, SHP-W, SHP-A, SHP-R e DPH para o conjunto de instâncias X.

Instâncias	FFF-W	FFF-A	FFF-R	SHP-W	SHP-A	SHP-R	DPH	DPH t(s)
X1	21.35	22.97	34.75	28.24	28.93	37.02	14.34	102.01
X2	7.10	7.07	29.58	33.82	34.12	38.81	5.68	33.52
X3	20.74	20.74	33.81	36.51	36.51	38.68	10.97	79.29
X4	23.69	22.42	33.54	33.61	34.82	39.15	11.07	149.67
X5	28.03	24.69	40.27	37.01	38.20	42.69	14.82	16.76
X6	20.99	20.99	33.08	33.88	33.88	35.94	9.37	191.98
X7	22.90	22.90	29.33	31.16	30.78	31.83	14.61	65.88
X8	52.31	52.31	55.41	45.08	45.08	50.32	32.06	114.30
X9	16.63	19.61	31.50	17.88	19.61	30.32	13.39	76.32
X10	22.13	22.13	31.79	28.52	29.93	32.13	11.27	188.63
X11	24.40	21.81	34.41	30.33	31.13	34.24	11.95	123.93
X12	22.76	22.81	33.61	29.85	28.98	35.91	11.61	129.72
X13	19.93	19.93	33.45	31.38	31.38	35.79	10.18	132.06
X14	23.36	23.36	35.30	33.24	33.24	37.10	12.43	55.40
X15	23.73	28.58	34.37	33.34	31.29	36.19	12.16	104.56
média:	23.34	23.49	34.95	32.26	32.53	37.08	13.06	104.26

4.6 Considerações Finais

Neste capítulo, apresentamos duas heurísticas construtivas estendidas da literatura para o 2DCSP-BC, além de uma heurística baseada em programação dinâmica que utiliza como subrotina o algoritmo exato de tempo pseudo-polinomial programação dinâmica desenvolvido para o Problema da Mochila Retangular com Restrições em Lote. Experimentos computacionais realizados mostraram que a heurística de programação dinâmica obteve um *gap* de otimalidade médio relativo inferior às heurísticas construtivas FFF e SHP, obtendo um desperdício de área, com relação ao limitante inferior, 33,84% menor no conjunto de instâncias A, 35,77% menor no conjunto de instâncias B, e 44,04% menor no conjunto de instâncias X, quando comparada à heurística FFF-W.

Trabalhos futuros podem investigar métodos para se obter melhores limites inferiores para o problema. Outras possíveis extensões deste trabalho incluem propor métodos exatos como Programação Linear Inteira para resolver o problema. Alternativamente, outros métodos heurísticos como algoritmos genéticos e heurísticas baseadas em programação por restrições podem ser desenvolvidos.

Capítulo 5

Conclusão

Nesta tese, métodos exatos e heurísticos foram desenvolvidos para três variantes de problemas de roteamento, empacotamento e corte guilhotinado: o Problema do Roteamento de Veículos com Múltiplas Janelas de Tempo, o Problema da Configuração de Produtos, e o Problema do Corte Guilhotinado Bidimensional em Três Estágios com Restrições de Precedência em Lote.

No Problema do Roteamento de Veículos com Múltiplas Janelas de Tempo, um algoritmo de geração de colunas que calcula a relaxação linear do problema é proposto. Ademais, uma heurística *Variable Neighborhood Search* e uma heurística de pós-otimização foram desenvolvidas para o problema. Os resultados mostraram os primeiros limites inferiores para o problema, além da heurística de pós-otimização ter melhorado os resultados da heurística VNS, de modo que esses algoritmos podem ser aplicados com eficiência em instâncias com até 17 clientes.

No Problema da Configuração de Produtos, uma formulação de programação linear inteira e um algoritmo exato de programação dinâmica de tempo pseudo-polinomial foram propostos. Experimentos computacionais mostraram que os algoritmos B&B e DP apresentaram resultados superiores à heurística BRKGA e ao algoritmo de *backtracking* de [Pereira et al., 2017], sendo capazes de encontrar soluções ótimas para todas as instâncias avaliadas com até 10.000 componentes, enquanto o algoritmo exato de *backtracking* não foi capaz de encontrar a solução ótima para nenhuma dessas instâncias.

Por fim, três heurísticas foram desenvolvidas para o Problema do Corte Guilhotinado Bidimensional em Três Estágios com Restrições de Precedência em Lote, sendo duas heurísticas construtivas estendidas da literatura e uma heurística baseada em um algoritmo de programação dinâmica desenvolvido para o Problema da Mochila Retangular com Restrições de Precedência em Lote. Os resultados mostraram que a

heurística de programação dinâmica obteve resultados superiores às demais, reduzindo o *gap* de otimalidade médio em até 44,04%, quando comparada à segunda melhor heurística desenvolvida.

Referências Bibliográficas

Alvarez-Valdes, R.; Martí, R.; Tamarit, J. M. & Parajón, A. (2007). Grasp and path relinking for the two-dimensional two-stage cutting-stock problem. *INFORMS Journal on Computing*, 19(2):261--272.

Amor, H. M. B.; Desrosiers, J. & Frangioni, A. (2009). On the choice of explicit stabilizing terms in column generation. *Discrete Applied Mathematics*, 157(6):1167--1184.

Andrade, R.; Birgin, E. G. & Morabito, R. (2016). Two-stage two-dimensional guillotine cutting stock problems with usable leftover. *International Transactions in Operational Research*, 23(1-2):121--145.

Asadi, M.; Soltani, S.; Gasevic, D.; Hatala, M. & Bagheri, E. (2014). Toward automated feature model configuration with optimizing non-functional requirements. *Information and Software Technology*, 56(9):1144–1165.

Barnhart, C.; Johnson, E. L.; Nemhauser, G. L.; Savelsbergh, M. W. & Vance, P. H. (1998). Branch-and-price: Column generation for solving huge integer programs. *Operations research*, 46(3):316--329.

Batory, D. S. (2005). Feature models, grammars and propositional formulas. pp. 7–20.

Belhaiza, S.; Hansen, P. & Laporte, G. (2014). A hybrid variable neighborhood tabu search heuristic for the vehicle routing problem with multiple time windows. *Computers and Operations Research*, 52:269--281.

Benavides, D.; Trinidad, P. & Ruiz-Cortés, A. (2005). Automated reasoning on feature models. *17th Conference on Advanced Information Systems Engineering (CAiSE)*, 1:491–503.

Carlton, W. B. & Barnes, J. W. (1996). Solving the traveling-salesman problem with time windows using tabu search. *IIE transactions*, 28(8):617--629.

Chabrier, A. (2006). Vehicle routing problem with elementary shortest path based column generation. *Computers and Operations Research*, 33(10):2972--2990.

Chen, Q.; Chen, Y.; Cui, Y.; Lu, X. & Li, L. (2015). A heuristic for the 3-staged 2d cutting stock problem with usable leftover. Em *2015 International Conference on Electrical, Automation and Mechanical Engineering*, pp. 776--779, Phuket. Atlantis Press. ISSN 2352-5401.

Cherri, A. C.; Arenales, M. N. & Yanasse, H. H. (2009). The one-dimensional cutting stock problem with usable leftover–a heuristic approach. *European Journal of Operational Research*, 196(3):897--908.

Cho, G. & Shaw, D. X. (1997). A depth-first dynamic programming algorithm for the tree knapsack problem. *INFORMS Journal on Computing*, 9(4):431–438.

Clements, P. & Northrop, L. (2001). *Software Product Lines: Practices and Patterns*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA.

Cui, Y.; Song, X.; Chen, Y. & Cui, Y. (2017). New model and heuristic solution approach for one-dimensional cutting stock problem with usable leftovers. *Journal of the Operational Research Society*, 68(3):269--280.

Czarnecki, K. & Eisenecker, U. (2007). *Generative Programming: Methods, Tools and Applications*. Addison-Wesley Longman Publising Co., Inc., Boston, MA.

Czarnecki, K. & Wasowsky, A. (2007). Feature diagrams and logics: There and back again. *11th International Software Product Lines Conference (SPLC)*, 1:23–24.

Dantzig, G. B. & Ramser, J. H. (1959). The truck dispatching problem. *Management science*, 6(1):80--91.

Desaulniers, G.; Desrosiers, J. & Solomon, M. M. (2006). *Column generation*, volume 5. Springer Science and Business Media.

Dumas, Y.; Desrosiers, J.; Gelinas, E. & Solomon, M. M. (1995). An optimal algorithm for the traveling salesman problem with time windows. *Operations research*, 43(2):367--371.

Favaretto, D.; Moretti, E. & Pellegrini, P. (2007). Ant colony system for a vrp with multiple time windows and multiple visits. *Journal of Interdisciplinary Mathematics*, 10(2):263--284.

Ferreira, H. S.; Bogue, E. T.; Noronha, T. F.; Belhaiza, S. & Prins, C. (2018). Variable neighborhood search for vehicle routing problem with multiple time windows. *Electronic Notes in Discrete Mathematics*, 66:207 – 214. ISSN 1571-0653. 5th International Conference on Variable Neighborhood Search.

Furini, F.; Malaguti, E. & Thomopulos, D. (2016). Modeling two-dimensional guillotine cutting problems via integer programming. *INFORMS Journal on Computing*, 28:736--751.

Gary, M. R. & Johnson, D. S. (1979). Computers and intractability: A guide to the theory of np-completeness.

Gianessi, P.; Alfandari, L.; Létocart, L. & Calvo, R. W. (2016). A column generation based heuristic for the multicommodity-ring vehicle routing problem. *Transportation Research Procedia*, 12:227--238.

Goedicke, M.; Köllmann, C. & Zdun, U. (2004). Designing runtime variation points in product line architectures: Three cases. *Science of Computer Programming*, 53(3):353–380.

Guo, J.; White, J.; Wang, G.; Li, J. & Wang, Y. (2011). A genetic algorithm for optimized feature selection with resource constraints in software product lines. *Journal System and Software*, 84(12):2208–2221.

Hifi, M. & Roucairol, C. (2001). Approximate and exact algorithms for constrained (un) weighted two-dimensional two-staged cutting stock problems. *Journal of combinatorial optimization*, 5(4):465--494.

Jackson, D.; Estler, H. & Rayside, D. (2009). A guided improvement algorithm for exact, general purpose, many-objective combinatorial optimization. Relatório técnico, MIT Computer Science and Artificial Intelligence Laboratory.

Jin, M.; Ge, P. & Ren, P. (2015). A new heuristic algorithm for two-dimensional defective stock guillotine cutting stock problem with multiple stock sizes. *Tehnicki vjesnik/Technical Gazette*, 22(5).

Johnson, D. S. & Niemi, K. (1983). On knapsacks, partitions, and a new dynamic programming technique for trees. *Mathematics of Operations Research*, 8(1):1--14.

Kang, K.; Cohen, S.; Hess, J.; Novak, W. & Peterson, S. (1990). Feature-oriented domain analysis (foda) feasibility study. Relatório técnico CMU/SEI-90-TR-021, Software Engineering Institute.

Kang, K.; Lee, J. & Donohoe, P. (2002). Feature-oriented product line engineering. *IEEE Software*, 19(4):58–65.

Karp, R. M. (1972). Reducibility Among Combinatorial Problems. Em Miller, R. E. & Thatcher, J. W., editores, *Complexity of Computer Computations*, pp. 85--103. Plenum Press.

Lai, K. K. & Chan, J. (1997). Developing a simulated annealing algorithm for the cutting stock problem. *Computers and Industrial Engineering*, 32:115–127.

Lin, S. & Kernighan, B. W. (1973). An effective heuristic algorithm for the traveling-salesman problem. *Operations research*, 21(2):498--516.

Lydia, T. & Quentin, V. (2018). Challenge roadef - euro 2018 cutting optimization problem description.

Martin, O. (1992). Large-step markov chains for the traveling salesman problem. *Operations Research Letters*, 11:219--224.

Mendonca, M.; Wasowski, A. & Czarnecki, K. (2009). Sat-based analysis of feature models is easy. Em *Proceedings of the 13th International Software Product Line Conference*, pp. 231--240. Carnegie Mellon University.

Monaci, M. & Toth, P. (2006). A set-covering-based heuristic approach for bin-packing problems. *INFORMS Journal on Computing*, 18:71–85.

Morais, V. W. C.; Mateus, G. R. & Noronha, T. F. (2014). Iterated local search heuristics for the vehicle routing problem with cross-docking. *Expert Systems with Applications*, 41(16):7495 – 7506. ISSN 0957-4174.

Padberg, M. & Rinaldi, G. (1987). Optimization of a 532-city symmetric traveling salesman problem by branch and cut. *Operations Research Letters*, 6(1):1--7.

Pereira, J. A.; Constantino, K. & Figueiredo, E. (2015). A systematic literature review of software product line management tools. *14th International Conference on Software Reuse (ICSR)*, pp. 73–89.

Pereira, J. A.; Maciel, L.; Noronha, T. F. & Figueiredo, E. (2017). Heuristic and exact algorithms for product configuration in software product lines. *International Transactions in Operational Research*, 24(6):1285--1306.

Pohl, K.; Böckle, G. & van Der Linden, F. (2005). *Software Product Line Engineering: Foundations, Principles and Techniques.* Springer-Verlag New York, Inc., Secaucus, NJ.

Puchinger, J.; Raidl, G. & Koller, G. (2004). Solving a real-world glass cutting problem. Em Gottlieb, J. & Raidl, G. R., editores, *Evolutionary Computation in Combinatorial Optimization*, pp. 165--176, Berlin, Heidelberg. Springer Berlin Heidelberg.

R. Olaechea, S. Stewart, K. C. & Rayside, D. (2012). Modelling and multi-objective optimization of quality attributes in variability-rich software. *4th International Workshop on Nonfunctional System Properties in Domain Specific Modeling Languages*, pp. 1–6.

Shaw, D. X. & Cho, G. (1994). *A branch-and-bound procedure for the tree knapsack problem.* School of Industrial Engineering, Purdue University.

Sinclair, K.; Cordeau, J.-F. & Laporte, G. (2016). A column generation post-optimization heuristic for the integrated aircraft and passenger recovery problem. *Computers and Operations Research*, 65:42--52.

Solomon, M. M. (1987). Algorithms for the vehicle routing and scheduling problems with time window constraints. *Operations research*, 35(2):254--265.

Spitzer, M.; Wildenhain, J.; Rappsilber, J. & Tyers, M. (2014). Boxplotr: a web tool for generation of box plots. *Nature methods*, 11(2):121.

Suliman, S. (2006). A sequential heuristic procedure for the two-dimensional cutting-stock problem. *International Journal of Production Economics*, 99(1-2):177--185.

Taillard, É. D. (1999). A heuristic column generation method for the heterogeneous fleet vrp. *RAIRO-Operations Research*, 33(1):1--14.

Taş, D.; Gendreau, M.; Dellaert, N.; Woensel, T. V. & Kok, A. D. (2014). Vehicle routing with soft time windows and stochastic travel times: A column generation and branch-and-price solution approach. *European Journal of Operational Research*, 236(3):789--799.

Tempelmeier, H. (2011). A column generation heuristic for dynamic capacitated lot sizing with random demand under a fill rate constraint. *Omega*, 39(6):627--633.

Thüm, T.; Kästner, C.; Benduhn, F.; Meinicke, J.; Saake, G. & Leich, T. (2014). Feature ide: An extensible framework for feature-oriented software development. *Science of Computer Programming*, 79(0):70–85.

Vassiliadis, V. S. (2005). Two-dimensional stock cutting and rectangle packing: binary tree model representation for local search optimization methods. *Journal of food engineering*, 70(3):257--268.

White, J.; Dougherty, B. & Schmidt, D. (2009). Selecting highly optimal architectural feature sets with filtered cartesian flattening. *Journal of Systems and Software*, 82(8):1268–1284.

Yanasse, H. H. & Morabito, R. (2006). Linear models for 1-group two-dimensional guillotine cutting problems. *International Journal of Production Research*, 44(17):3471--3491.