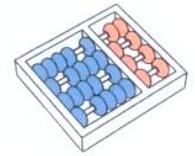


Jefferson Luiz Moisés da Silveira

**“Algoritmos para Problemas de Empacotamento e Roteamento.”**

CAMPINAS  
2013





Universidade Estadual de Campinas  
Instituto de Computação

Jefferson Luiz Moisés da Silveira

## “Algoritmos para Problemas de Empacotamento e Roteamento.”

Orientador(a): **Prof. Dr. Eduardo Candido Xavier**

Tese de Doutorado apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Computação da Universidade Estadual de Campinas para obtenção do título de Doutor em Ciência da Computação.

ESTE EXEMPLAR CORRESPONDE À VERSÃO FINAL DA TESE DEFENDIDA POR JEFFERSON LUIZ MOISÉS DA SILVEIRA, SOB ORIENTAÇÃO DE PROF. DR. EDUARDO CANDIDO XAVIER.

A handwritten signature in blue ink, appearing to read "Eduardo C. Xavier", written over a horizontal line.

Assinatura do Orientador(a)

CAMPINAS  
2013

Ficha catalográfica  
Universidade Estadual de Campinas  
Biblioteca do Instituto de Matemática, Estatística e Computação Científica  
Maria Fabiana Bezerra Muller - CRB 8/6162

Si39a Silveira, Jefferson Luiz Moisés da, 1986-  
Algoritmos para problemas de empacotamento e roteamento. / Jefferson Luiz Moisés da Silveira. – Campinas, SP : [s.n.], 2013.

Orientador: Eduardo Candido Xavier.  
Tese (doutorado) – Universidade Estadual de Campinas, Instituto de Computação.

1. Problema de empacotamento. 2. Problema de roteamento de veículos. 3. Algoritmos aproximados. 4. Metaheurística. I. Xavier, Eduardo Candido, 1979-. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Informações para Biblioteca Digital

**Título em outro idioma:** Algorithms for packing and routing problems.

**Palavras-chave em inglês:**

Packing Problem

Vehicle Routing Problem

Approximation Algorithms

Metaheuristic

**Área de concentração:** Ciência da Computação

**Titulação:** Doutor em Ciência da Computação

**Banca examinadora:**

Eduardo Candido Xavier [Orientador]

Cid Carvalho de Souza

Victor Fernandes Cavalcante

Reinaldo Morábito Neto

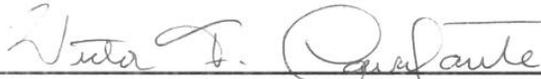
Horacio Hideki Yanasse

**Data de defesa:** 02-10-2013

**Programa de Pós-Graduação:** Ciência da Computação

# TERMO DE APROVAÇÃO

Tese Defendida e Aprovada em 02 de outubro de 2013, pela Banca  
examinadora composta pelos Professores Doutores:



---

**Prof. Dr. Victor Fernandes Cavalcante**  
**IBM Research Brasil**



---

**Prof. Dr. Horácio Hideki Yanasse**  
**CTE / INPE**



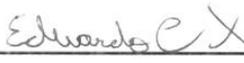
---

**Prof. Dr. Reinaldo Morábito Neto**  
**DEP / UFSCar**



---

**Prof. Dr. Cid Carvalho de Souza**  
**IC / UNICAMP**



---

**Prof. Dr. Eduardo Candido Xavier**  
**IC / UNICAMP**



# Algoritmos para Problemas de Empacotamento e Roteamento.

Jefferson Luiz Moisés da Silveira<sup>1</sup>

02 de outubro de 2013

## Banca Examinadora:

- Prof. Dr. Eduardo Candido Xavier (Orientador)
- Prof. Dr. Cid Carvalho de Souza  
Instituto de Computação - UNICAMP
- Dr. Victor Fernandes Cavalcante  
IBM Research - Brasil
- Prof. Dr. Reinaldo Morábito  
DEP - UFSCar
- Prof. Dr. Horacio Hideki Yanasse  
CTE - INPE
- Prof. Dr. Flávio Keidi Miyazawa  
Instituto de Computação - UNICAMP (Suplente)
- Prof. Dr. Luis Augusto Angelotti Meira  
FT - UNICAMP (Suplente)
- Prof. Dr. Yuri Abitbol de Menezes Frota  
IC - UFF (Suplente)

---

<sup>1</sup>Auxílio Financeiro: Capes e FAPESP (processo 2011/08563-9) 2011–2013



# Abstract

In this work we are interested in packing and routing problems. Assuming  $P \neq NP$ , we have that there are no efficient algorithms to deal with such problems. Besides exact algorithms, two approaches to solve such problems are Approximation Algorithms and Heuristics. In this thesis we show algorithms using these three approaches for both packing and routing problems.

The first two addressed problems are generalizations of classical packing problems: The Two Dimensional Knapsack problem and the Strip Packing problem. These problems were generalized by adding constraints on the way the items can be inserted/removed into/from the bin (These constraints appear in the context of routing problems). The third problem is combination of packing and routing problems. It is a generalization of the classical Pickup and Delivery problem.

We propose the first approximation results for some packing problems. Besides that, we present some practical algorithms for the third problem. The heuristics were assessed through computational experiments by comparing their results with exact algorithms.



# Resumo

Neste trabalho estamos interessados em problemas de empacotamento e roteamento. Assumindo a hipótese de que  $P \neq NP$ , sabemos que não existem algoritmos eficientes para resolver tais problemas. Além de algoritmos exatos, duas das abordagens para resolver tais problemas são Algoritmos Aproximados e Heurísticas. Nesta tese mostramos algoritmos baseados nestas três abordagens para ambos os problemas, de empacotamento e roteamento.

Os dois primeiros problemas atacados foram generalizações de problemas clássicos de empacotamento: O problema da mochila bidimensional e o problema de empacotamento em faixas. Estes foram generalizados adicionando restrições na forma de carregamento e descarregamento dos itens no recipiente (restrições estas, que aparecem no contexto de problemas de roteamento). O terceiro problema é uma combinação de problemas de empacotamento e roteamento. Neste caso, atacamos uma generalização do clássico *Pickup and Delivery Problem*.

Propomos os primeiros resultados de aproximação para algumas versões dos problemas de empacotamento supracitados. Além disto, apresentamos algumas abordagens práticas para o terceiro problema. As heurísticas foram avaliadas através de experimentos computacionais comparando os seus resultados com algoritmos exatos.



*À minha mãe, Geralda Silveira.*



*“... It is a tale, told by an idiot, full of  
sound and fury, signifying nothing.”*

William Shakespeare



# Sumário

<b>Abstract</b>	<b>ix</b>
<b>Resumo</b>	<b>xi</b>
<b>Dedicatória</b>	<b>xiii</b>
<b>Epígrafe</b>	<b>xv</b>
<b>1 Introdução</b>	<b>1</b>
1.1 Objetivos do Trabalho . . . . .	2
1.2 Organização do Trabalho . . . . .	2
<b>2 Preliminares</b>	<b>3</b>
2.1 Problemas de Roteamento . . . . .	3
2.2 Problemas de Empacotamento com Restrições de Carregamento e Descarregamento . . . . .	5
2.3 Algoritmos Aproximados . . . . .	9
2.4 Metaheurística <i>GRASP</i> . . . . .	11
<b>3 Resumo dos Resultados</b>	<b>15</b>
3.1 A Note on a Two Dimensional Knapsack Problem With Unloading Constraints . . . . .	15
3.2 Two Dimensional Strip Packing with Unloading Constraints . . . . .	16
3.3 On The Pickup and Delivery with Two Dimensional Loading/Unloading Constraints . . . . .	17
3.4 Outros Trabalhos Feitos Durante o Doutorado . . . . .	18
3.4.1 Artigo: <i>Heuristics for the strip packing problem with unloading constraints</i> . . . . .	18
3.4.2 Artigo: <i>On the Worst Case of Scheduling with Task Replication on Computational Grids</i> . . . . .	19



<b>4</b>	<b>Artigo: A Note on a Two Dimensional Knapsack Problem With Unloading Constraints</b>	<b>21</b>
4.1	Introduction . . . . .	21
4.2	A Hybrid Algorithm for the KU problem . . . . .	24
4.3	A 4-approximation Algorithm for the KU Problem . . . . .	27
4.4	Concluding Remarks . . . . .	30
4.5	References . . . . .	30
<b>5</b>	<b>Artigo: Two Dimensional Strip Packing with Unloading Constraints</b>	<b>33</b>
5.1	Introduction . . . . .	33
5.2	An algorithm for the SPU <sub>vh</sub> problem . . . . .	36
5.2.1	$\mathcal{A}_{vh}$ Analysis . . . . .	41
5.3	An Algorithm to a parametric oriented case of the SPU <sub>vh</sub> Problem . . . . .	42
5.3.1	$\mathcal{A}_{vh}^p$ Analysis . . . . .	43
5.4	An Algorithm to the SPU <sub>v</sub> Problem . . . . .	44
5.4.1	$\mathcal{A}_v$ Analysis . . . . .	46
5.5	An algorithm to a parametric version of the SPU <sub>v</sub> problem . . . . .	47
5.5.1	$\mathcal{A}_v^p$ Analysis . . . . .	47
5.6	Concluding Remarks . . . . .	48
5.7	References . . . . .	48
5.8	Appendix . . . . .	49
5.8.1	Minimizing function (5.1) . . . . .	49
5.8.2	Minimizing function (5.2) . . . . .	50
<b>6</b>	<b>On The Pickup and Delivery with Two Dimensional Loading/Unloading Constraints</b>	<b>53</b>
6.1	Introduction . . . . .	54
6.1.1	Our Results . . . . .	56
6.1.2	Paper Organization . . . . .	56
6.2	Definitions and Basic Notation . . . . .	57
6.3	Exact Algorithms . . . . .	58
6.3.1	A Backtracking algorithm for the 2KPLU . . . . .	58
6.3.2	A CP Model for the 2KPLU . . . . .	66
6.3.3	An ILP based algorithm for the PDPLU . . . . .	67
6.4	Heuristic Algorithms . . . . .	69
6.4.1	Bottom Left heuristics for the 2KPLU . . . . .	69
6.4.2	A simple heuristic for the PDPLU . . . . .	70
6.4.3	A <i>GRASP</i> heuristic for the PDPLU . . . . .	71
6.5	List of Routes . . . . .	75



6.6	Generated Instances . . . . .	77
6.7	Computational experiments . . . . .	78
6.8	Conclusions . . . . .	83
6.9	References . . . . .	84
6.10	Appendix: Proof of Lemma 1 . . . . .	87
<b>7</b>	<b>Considerações Finais</b>	<b>91</b>
7.1	Trabalhos Futuros . . . . .	92
	<b>Referências Bibliográficas</b>	<b>93</b>



# Lista de Tabelas

6.1	Parameters for the GRASP heuristic. . . . .	80
-----	---	----



# Lista de Figuras

2.1	Exemplo do 2L-CVRP. . . . .	5
2.2	Exemplo do PDPLU. . . . .	6
2.3	Exemplos de remoção de itens de um <i>bin</i> . . . . .	8
2.4	Exemplos de solução inviável (1) e viável (2) para o 2KPU. . . . .	9
2.5	Exemplos de soluções inviáveis (1 e 2) e viável (3) para o 2KPLU. . . . .	10
4.1	Suppose we have a squared bin $B$ and four items $a_i, i = 1, 2, 3, 4$ with $c(a_2) = c(a_3) = 1$ and $c(a_1) = c(a_4) = 2$ . Any 2-staged feasible solution for this problem uses at most 3 items, while the pattern above uses all 4 items and is feasible for the KU problem. If we had $c(a_2) > c(a_1)$ then this packing would be infeasible since $a_2$ would be blocking $a_1$ . . . . .	23
4.2	The algorithm uses the 4 <sup>th</sup> bin since two items did not fit into bins 1 and 2.	29
5.1	Suppose that $c(a_j) > c(a_k) > c(a_i)$ . (a): An infeasible solution to $\text{SPU}_{vh}$ , because $a_k$ and $a_j$ are blocking $a_i$ . (b): A feasible placement to $\text{SPU}_{vh}$ , where the items can be removed in order 1, 2 and 3. (c): This placement is not a feasible solution to $\text{SPU}_v$ , because $a_j$ is blocking $a_i$ . (d): A feasible placement to $\text{SPU}_v$ , where the items can be removed in order. . . . .	34
5.2	The packing generated by the <i>Base</i> algorithm. Items in each shelf are sorted by class in non-increasing order (i.e. $c(a_1) \geq c(a_2) \geq \dots \geq c(a_j)$ ). Furthermore, the shelves in $\mathbf{P}^F \cup \mathbf{T}^F$ are sorted by the class of the right most item (i.e. $c(a_j) \geq c(a_k) \geq c(a_l)$ ). Finally, the items in $\mathbf{M}$ are sorted by class in non-increasing order (i.e. $c(a_m) \geq c(a_{m+1})$ ). . . . .	39
5.3	The boundaries of the analyzed region: $y = x, y = 1, y = \frac{0.263422}{x}$ . . . . .	50
6.1	Consider the route defined above: Pickup items from A ( $p_a$ ), Pickup items from B ( $p_b$ ), delivery items from A ( $d_a$ ) and delivery items from B ( $d_b$ ) (Part (a)). In (b) we have an infeasible packing, since the removal of two items from A is blocked by an item from B. In (c) we have another infeasible packing, since there are two items from A blocking the insertion of an item from B. In (d) we have a feasible packing. . . . .	55



6.2	Suppose $\epsilon > 0$ sufficiently small and $a_i \prec a_j$ , then the bottom left corner of $a_j$ must be in (R1) (part a). Also $a_i$ 's bottom-left corner must be in (R2) (part b). . . . .	60
6.3	In this figure we represent the contour lines of the packed items ( $l(\mathcal{P})$ ) with dashed lines. The region below $l(\mathcal{P})$ is the $e(\mathcal{P})$ and above $e^-(\mathcal{P})$ . The white circles represent the set $corner(\mathcal{P})$ . Suppose that an item $a$ is being packed and that it will be removed after the items $A'$ , then we have the set $orderContour(a, \mathcal{P})$ represented by black circles. . . . .	61
6.4	Example of packings using different orderings of items. . . . .	65
6.5	In this case, the item $a$ do not need to be packed in $p$ . There is another order of the items that can produce a packing where item $a$ is packed on the point below $p$ . . . . .	66
6.6	Suppose $a_5$ is not packed yet, and that items $a_i$ $1 \leq i \leq 4$ belongs to the set $S$ , i.e, $a_5$ is going to be packed after them, and all of them are going to be removed before $a_5$ . There is no space to pack $a_5$ , thus this configuration is invalid. . . . .	67
6.7	In this figure we have an example for the $BL_{LU}$ where $b$ is the only item that will be removed before $a$ . The item $a$ can not be packed in the point $r$ since $a$ would not fit in the bin. Point $q$ is also infeasible for $a$ since the packing would not satisfy the unloading constraint. Points $p$ and $s$ are feasible and $p$ is chosen since it is the lowest one. . . . .	71
6.8	In this figure we have the same scenario from the Fig. 6.7. In this case, the point $s$ is chosen since it maximizes the touching perimeter of $a$ . . . . .	72
6.9	In this figure we have a similar scenario from the Figures 6.7 and 6.8. Suppose that items $a$ and $c$ do not share the bin. In this case, the point $p$ is chosen due to the overlap between $a$ and $c$ . . . . .	73
6.10	Performance profiles for the cost(left) and running time(right) of the exact algorithms for the PDPLU. . . . .	80
6.11	Cost(left) and time(right) factor for the heuristics in all instances. . . . .	81
6.12	Performance profiles for the solutions cost (left) and running time (right) for the heuristics. Instances with unitary items. . . . .	81
6.13	Performance profiles for the solutions cost (left) and running time (right) for the heuristics. Instances with "short and wide" items. . . . .	82
6.14	Performance profiles for the solutions cost (left) and running time (right) for the heuristics. Instances with "tall and thin" items. . . . .	82
6.15	Performance profiles for the solutions cost (left) and running time (right) for the heuristics. Instances with "homogeneous" items. . . . .	83



6.16	For each heuristic, the figure presents the average time in seconds used to solve an instance of a given size. . . . .	84
6.17	For each heuristic, and for each instance size, the figure presents the average proportion of packing sub-problems solved by the Routes List data structure.	85



# Lista de Notas nos Artigos

Each item has an associated profit. . . . .	21
Integer Linear Programming (ILP) . . . . .	22
Input list of rectangles . . . . .	24
Error note: Any $\beta$ approximation is sufficient. The term <i>absolute</i> was misused here.	25
With approximation factor of 2. . . . .	27
author's . . . . .	35
break . . . . .	40
Error: Total height of a <i>shelf</i> containing an item of type $\mathcal{T}$ . . . . .	41



# Capítulo 1

## Introdução

Neste trabalho estamos interessados em problemas de empacotamento bidimensionais e problemas de roteamento. Muitas das variações destes problemas clássicos de otimização pertencem à classe NP-difícil [25]. Nos problemas de otimização combinatória temos um conjunto, geralmente finito, de objetos e uma função objetivo definida sobre este conjunto. O objetivo consiste em encontrar um dos objetos que minimiza ou maximiza a função objetivo dada. Em geral, métodos exaustivos são inviáveis para se resolver tais problemas devido ao tamanho do conjunto de objetos.

Como trabalho de doutorado realizamos um estudo sobre problemas de roteamento e empacotamento bidimensionais com certas restrições relacionadas aos problemas de roteamento.

Nos problemas de roteamento temos um grafo com dois conjuntos de vértices identificados como: depósitos e clientes, além de um conjunto de requisições de cada cliente. Neste caso, uma requisição pode ser uma demanda de carga, cuja representação pode ser simplesmente um peso, ou formato de cada item da carga, que por sua vez pode ser uni, bi ou tridimensional. Desta forma, o problema consiste em encontrar um conjunto de rotas (percorridas por veículos) que minimizam uma função objetivo dada, atendam as requisições de cada cliente e respeitem as restrições do problema (Capacidades dos veículos, números de visitas aos clientes, etc). Como exemplo de problema de roteamento podemos citar o *Capacitated Vehicle Routing Problem* (CVRP) [48].

Nos problemas de empacotamento, por sua vez, temos um conjunto de objetos grandes, chamados de *bins*, e vários objetos menores, chamados de itens. O objetivo do problema é empacotar itens dentro de *bins*, de forma a maximizar ou minimizar uma dada função objetivo. Como exemplos clássicos de problemas de empacotamento bidimensionais podemos citar o *Two Dimensional Bin Packing Problem* e o *Strip Packing Problem* [36].

Neste trabalho, supomos a hipótese de que  $P \neq NP$ . Desta forma, tais problemas de empacotamento, roteamento e muitos outros problemas de otimização que são NP-

difíceis, não possuem algoritmos eficientes, ou seja, polinomiais [7], para resolvê-los de forma exata. Muitos destes problemas aparecem em aplicações práticas e há um forte apelo econômico para resolvê-los.

Este trabalho é focado no desenvolvimento e análise de algoritmos aproximados, exatos e heurísticas para versões dos problemas supracitados. De maneira geral, heurísticas são algoritmos que abdicam da garantia de otimalidade a fim de reduzir a sua complexidade de tempo. Os Algoritmos Aproximados, por sua vez, são aqueles para os quais é possível demonstrar que suas soluções sempre estão dentro de um certo fator (distância) do ótimo.

## 1.1 Objetivos do Trabalho

O objetivo principal do trabalho é apresentar novos algoritmos para versões de problemas de roteamento e empacotamento bidimensional. Além disto, buscamos abordar alguns problemas de maneira mais teórica, utilizando algoritmos aproximados e outros de maneira mais prática com a implementação de heurísticas e algoritmos exatos que são testados realizando-se experimentos computacionais.

## 1.2 Organização do Trabalho

Esta tese está organizada como uma coletânea de artigos. Apesar deste modelo induzir repetições em algumas definições, ele nos permite apresentar de maneira direta os resultados obtidos durante o doutorado.

O restante da tese foi organizada da seguinte forma. No Capítulo 2 introduzimos os problemas atacados durante o doutorado, além de apresentar algumas técnicas utilizadas no restante do trabalho. Um resumo dos resultados obtidos no doutorado é mostrado no Capítulo 3. Após isto, os artigos são apresentados nos capítulos seguintes (4, 5 e 6). Por fim, o Capítulo 7 contém as considerações finais do trabalho, além de discorrer sobre alguns possíveis trabalhos futuros.

# Capítulo 2

## Preliminares

Neste Capítulo discorremos sobre os problemas atacados na tese e algumas técnicas utilizadas para resolvê-los. Primeiramente apresentamos os problemas de roteamento considerados (Seção 2.1) e, em seguida, os de empacotamento (Seção 2.2). Por fim, resumimos alguns tópicos abordados nos capítulos seguintes: Algoritmos Aproximados (Seção 2.3) e a metaheurística *GRASP* (Seção 2.4).

### 2.1 Problemas de Roteamento

Problemas de roteamento de veículos consistem no atendimento de um conjunto de demandas de clientes, por meio de veículos que estão inicialmente localizados em depósitos. Em geral, cada veículo possui uma capacidade que não deve ser excedida pelas demandas atendidas por este veículo. Formalmente, temos um grafo completo  $G(V, E)$  com demandas  $d : V \rightarrow \mathbb{R}$  e custos  $c : E \rightarrow \mathbb{R}$ , e  $R$  veículos com capacidades  $p_i$ ,  $1 \leq i \leq R$ , inicialmente localizados em um vértice inicial  $d$  chamado de depósito. O objetivo é encontrar um conjunto de  $R$  rotas (ciclos)  $r_i$ , disjuntas (exceto por  $d$ ), onde cada ciclo contém o vértice  $d$ , todos os vértices são visitados e a soma das demandas em cada rota  $r_i$  não excede  $p_i$ . Este conjunto de rotas deve ter o menor custo possível (soma dos custos das arestas das rotas). De nosso conhecimento, os primeiros a estudar este problema foram Dantzig e Ramser em [14].

É fácil ver que este problema é NP-difícil, já que se fizermos  $R = 1$  e  $p_i = \sum_{v \in V} d(v)$ , então temos um método para se resolver o clássico problema *Traveling Salesman Problem* (TSP) [3].

Muitas variações desta versão básica do problema já foram propostas. Estas variam desde modificações da função objetivo (minimizar distância total, minimizar a utilização de veículos) até a adição de novas restrições (janelas de tempo, entregas particionadas) [33]. Em determinados casos, a demanda dos clientes pode ser representada como um

conjunto de produtos ou itens.

Transportadoras, por exemplo, devem organizar suas entregas em caminhões de forma a atender todos os pedidos de uma região minimizando os custos de distância e tempo para realizar esta tarefa. Neste cenário, obter bons arranjos de produtos nos caminhões desempenha um papel importante no problema como um todo, já que um arranjo mal feito pode gerar retrabalho nas tarefas de carregamento e descarregamento do caminhão.

Neste contexto, surge o problema *Capacitated Vehicle Routing Problem with Two Dimensional Loading Constraints* (2L-CVRP) [16, 26, 54, 23]. Nele o problema CVRP é combinado com problemas de empacotamento a fim de modelar uma situação onde se faz necessário entregar um conjunto de produtos ao longo de uma rota. Neste problema, busca-se minimizar o custo do transporte necessário para realizar entregas de produtos a clientes em diferentes localizações. Estes produtos (itens) são enviados em veículos (*bins*) que estão inicialmente situados no fornecedor dos produtos (todos os produtos são carregados antes de se percorrer a rota). Para acomodar os itens nos *bins* é utilizado um algoritmo de empacotamento com restrição de descarregamento, para garantir que, durante a entrega de produtos de cada cliente, não hajam produtos de outros clientes bloqueando a saída do *bin*.

Na Figura 2.1 temos um exemplo do 2L-CVRP. Os clientes A, B e C serão visitados em ordem (veja a parte (a)). O empacotamento apresentado em (b) é inviável pois há itens de B bloqueando a saída de um item de A. Por fim, o empacotamento em (c) é viável.

Em outros casos, além de descarregar produtos ao longo de uma rota, faz-se necessário carregar novos produtos no caminhão. Considere por exemplo o caso em que é necessário transportar demandas entre pares de clientes ao longo de uma rota. Se desconsiderarmos as demandas dos clientes e levarmos em consideração apenas a ordem de visita dos clientes então teremos o *Pickup and Delivery Problem* (PDP). Este problema é uma generalização do clássico TSP, descrito anteriormente. O PDP consiste em encontrar, no grafo de entrada, um ciclo hamiltoniano de custo mínimo que satisfaça todas as restrições de ordem de visita dos clientes (antes de entregar uma demanda é necessário pegá-la no seu respectivo cliente). Estas restrições são representadas através de pares de vértices do grafo (um de coleta e outro de entrega).

Se considerarmos que as demandas são produtos que serão carregados e descarregados ao longo da rota então teremos o problema referenciado na literatura como *Pickup and Delivery Problem with Two Dimensional Loading/Unloading Constraints* [38] (PDPLU). Neste caso o arranjo dos produtos (itens) no caminhão (*bin*) deve considerar que itens serão removidos e inseridos no *bin* em ordem definida pela rota considerada. Neste problema, temos o mesmo objetivo do PDP, porém, devemos considerar a inserção e remoção dos itens na rota. Desta forma, deve-se garantir que na inserção de um item, este não será

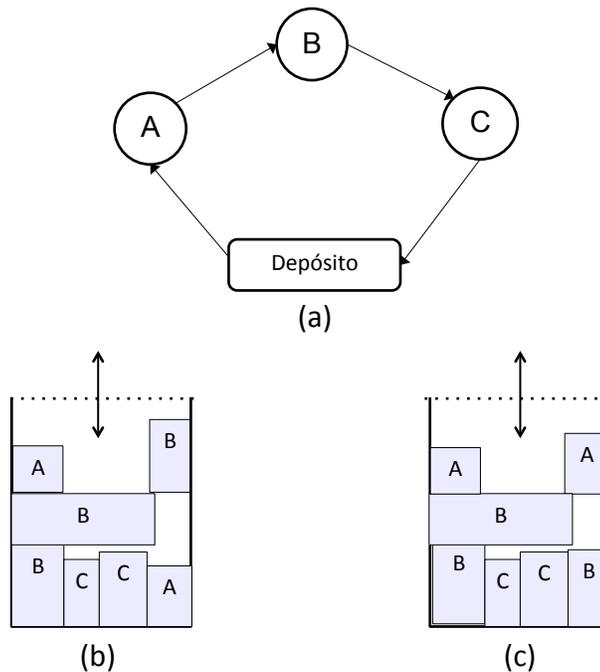


Figura 2.1: Exemplo do 2L-CVRP.

bloqueado até atingir a sua posição de empacotamento e, como no 2L-CVRP, o mesmo deve acontecer no seu descarregamento.

Na Figura 2.2 temos um exemplo do PDPLU. Considere que a rota é definida como “carregamento” de itens de A ( $p_a$ ), “carregamento” de itens de B ( $p_b$ ), “descarregamento” de itens de A ( $d_a$ ) e, por fim, “descarregamento” de itens de B ( $d_b$ ) (Parte (a)). Em (b) temos um empacotamento inviável, pois há um item de A cuja remoção é bloqueada pelo item de B. Em (c) temos um empacotamento inviável, já que há itens de A bloqueando o carregamento do item de B. Em (d) temos um empacotamento viável.

Na tese, estudamos o problema PDPLU e problemas de empacotamento relacionados ao 2L-CVRP.

## 2.2 Problemas de Empacotamento com Restrições de Carregamento e Descarregamento

Nos problemas de empacotamento, temos um ou mais objetos grandes  $n$ -dimensionais, os quais chamamos de *bins*, e vários objetos menores também  $n$ -dimensionais os quais chamamos de itens. O objetivo do problema é empacotar itens dentro de *bins*, de forma a

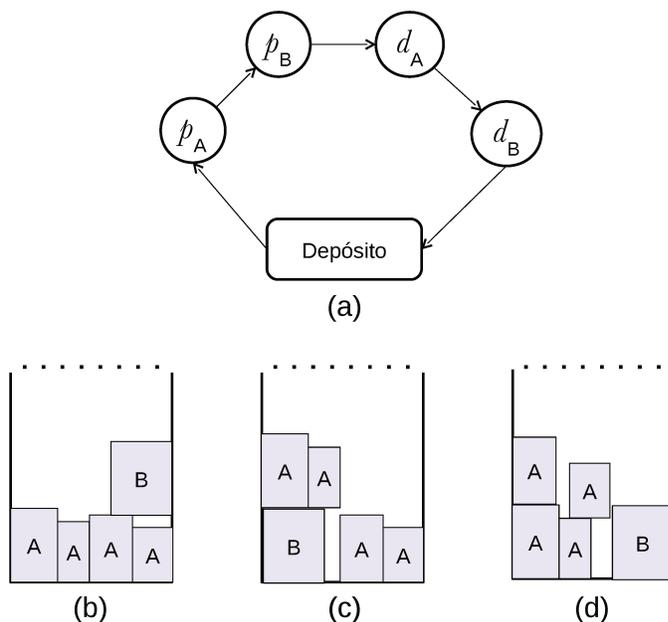


Figura 2.2: Exemplo do PDPLU.

maximizar ou minimizar uma dada função objetivo. Tanto os itens quanto os *bins* podem assumir formas regulares ou irregulares. Além disso, o empacotamento dos itens pode ser submetido à certas restrições (precedência, ordem de remoção, etc.) ou a rotações. O empacotamento deve ser feito de tal maneira que os itens não ocupem um mesmo espaço e que as restrições do *bins* e do problema sejam respeitadas.

Definiremos agora dois problemas de empacotamento básicos que são casos específicos dos problemas atacados na tese.

O primeiro problema é o *Strip Packing Problem* (SPP). Na versão bidimensional deste problema temos uma lista de itens bidimensionais retangulares  $I = (a_1, \dots, a_m)$ , cada item  $a_i$  com tamanho  $(h(a_i), w(a_i))$ , e uma faixa  $S$  de largura  $L$  e altura infinita. O objetivo do problema é empacotar todos os itens na faixa de tal maneira que seja minimizada a altura total utilizada para empacotar os itens. Versões multidimensionais podem ser consideradas.

O outro problema é conhecido como *Knapsack Problem* (KP), ou problema da mochila. Neste caso temos apenas um *bin*  $B$  de tamanho  $T$ , e uma lista de itens  $I = (a_1, \dots, a_m)$ , cada item  $a_i$  com tamanho  $s(a_i)$  e valor  $p(a_i)$ ,  $i = 1, \dots, m$ . O objetivo do problema é selecionar um conjunto  $I' \subseteq I$  de tal forma que  $\sum_{a_i \in I'} s(a_i) \leq T$  (os itens em  $I'$  caibam em  $B$ ) e que  $\sum_{a_i \in I'} p(a_i)$  seja maximizada. Também podemos considerar as versões multidimensionais deste problema, neste caso, tanto itens como o *bin* terão a mesma dimensão.

Diversas aplicações industriais podem ser modelados utilizando estes problemas de empacotamento. Dentre eles podemos citar: *problemas de corte de insumos* (aço, vidro, tecido, etc), *alocação de recursos*, *problemas de carregamento* (caminhões, vagões, contêineres, etc), entre outras [22, 8, 34, 52, 16].

Para uma visão mais abrangente da área de problemas de empacotamento, recomenda-se a leitura dos trabalhos [18] e [50], propostos por Dyckhoff (1990) e Wäscher *et al.* (2007).

Quando um problema de empacotamento é submetido à restrições que impõem a forma como itens (bi ou tridimensionais) devem ser inseridos/removidos do *bin* ou a ordem com que estas operações podem ser realizadas, então este é chamado de Problema de Empacotamento com Restrições de Carregamento e Descarregamento. Estes aparecem em áreas como Pesquisa Operacional e Logística.

Nos problemas considerados neste trabalho, exceto quando disposto contrário, será realizado apenas um movimento para o carregamento e outro para o descarregamento dos itens. Este movimento se dará a partir da posição de entrada do veículo, em linha reta, até a posição final de empacotamento, sem que haja sobreposição dos itens ao longo do caminho. Na Figura 2.3, apresentamos um exemplo de remoção e inserção de itens. Nas figuras 2.1, 2.2 e 2.3 os itens são inseridos e removidos pelo topo do *bin* utilizando-se apenas de um movimento. Ademais, nos problemas abordados, durante a remoção de um item não é permitida a movimentação dos itens já empacotados. Para fins de notação, nas figuras, o lado do *bin* que será utilizado para a inserção e remoção dos itens, estará tracejado, enquanto os outros três estarão em linha cheia (exceto no Capítulo 5, onde uma das dimensões do *bin* é ilimitada).

Na Figura 2.3 temos um exemplo de restrição de descarregamento. Suponha que descarregaremos os itens de A antes dos itens de B. Neste caso, o empacotamento apresentado em (a) é inviável, pois, mesmo havendo espaço para remover os itens de A pelo topo, isto exigiria mais de um movimento dos itens de A para remoção ou movimentar o item de B. Em (b) temos um empacotamento viável.

De maneira geral, na tese, abordamos três versões de problemas de empacotamento com restrições de carregamento e/ou descarregamento: *Two Dimensional Knapsack Problem with Unloading Constraints* (2KPU), *Strip Packing Problem with Unloading Constraints* (SPPU) e o *Two Dimensional Knapsack Problem with Loading and Unloading Constraints* (2KPLU).

O 2KPU pode ser definido da seguinte maneira. A entrada consiste em um *bin*  $B$  e uma lista de itens  $L = (a_1, \dots, a_m)$ , cada item  $a_i$  com altura e largura  $(h(a_i), w(a_i))$ , valor  $(v(a_i))$  e cliente ou classe  $(c(a_i))$ , além disso, o objetivo é empacotar um subconjunto de  $L$  de tal maneira que a soma dos valores dos itens empacotados seja maximizada e os itens podem ser removidos do empacotamento em ordem crescente de valores  $c(a_i)$ , sem alterar

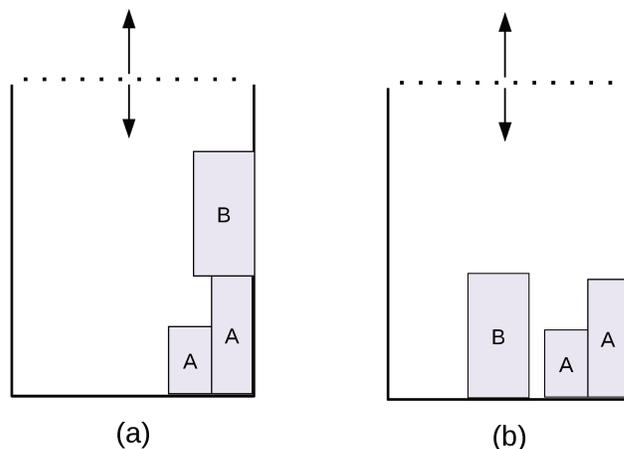


Figura 2.3: Exemplos de remoção de itens de um *bin*.

o posicionamento de demais itens ainda empacotados. Na Figura 2.4 temos um exemplo com 3 itens para o 2KPU. Supondo que  $c(A) < c(B)$  então temos que o empacotamento (1) é inviável enquanto que o (2) é viável e os itens podem ser removidos em ordem.

Por sua vez, o SPPU, na sua versão bidimensional, pode ser definido da seguinte forma. Como entrada do problema temos uma lista de itens bidimensionais  $L = (a_1, \dots, a_m)$ , cada item  $a_i$  definido pela sua altura ( $h(a_i)$ ), largura ( $w(a_i)$ ), classe  $c(a_i)$  e uma faixa  $S$  de largura  $W$  e altura infinita. O objetivo do problema é empacotar todos os itens na faixa de tal maneira que seja minimizada a altura total utilizada para empacotar os itens, respeitando a restrição de descarregamento (assim como no 2KPU). Como o SPPU possui a mesma restrição de descarregamento que o 2KPU, a Figura 2.4 também representa um exemplo para este problema. A única diferença é que, neste caso, a altura do *bin* seria ilimitada.

Por fim, o 2KPLU é similar ao 2KPU, com a substituição do valor  $c(a_i)$  (classe) por dois valores  $p(a_i)$  (carregamento) e  $d(a_i)$  (descarregamento). No 2KPU consideramos apenas o descarregamento dos itens do *bin*, enquanto que no 2KPLU consideramos também o carregamento dos itens, ou seja, no momento de inserção/remoção de um item no *bin*, deve existir um caminho livre entre a entrada do *bin* e a sua posição final de empacotamento, que utilize apenas um movimento vertical. Outra diferença, é que consideraremos a versão de decisão do 2KPLU ao invés da sua versão de otimização, portanto, o problema consiste em decidir se existe um empacotamento que satisfaz as restrições de carregamento

e descarregamento. Na Figura 2.5 apresentamos exemplos de empacotamento para o 2KPLU com 3 itens. Neste caso, considere a rota  $(v_1, v_2, v_3, v_4, v_1)$  e que os itens  $A$  e  $B$  vão do vértice  $v_1$  para  $v_3$  enquanto que  $C$  vai de  $v_2$  para  $v_4$ . Neste caso temos que (1) é inválido pois não há como carregar o item  $C$  com um movimento vertical apenas, (2) é inviável pois não há como descarregar  $B$  e, por fim, (3) é viável.

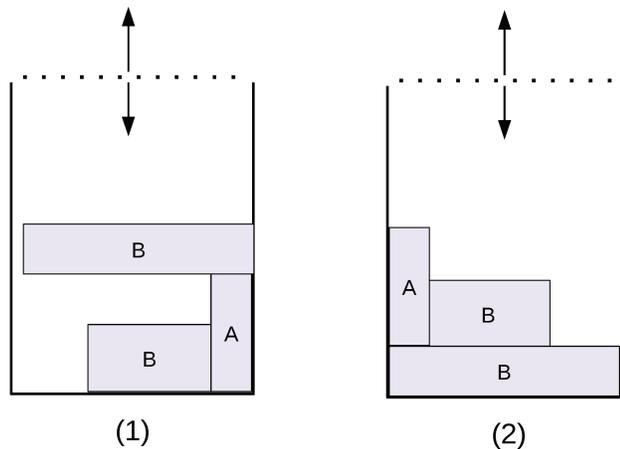


Figura 2.4: Exemplos de solução inviável (1) e viável (2) para o 2KPU.

## 2.3 Algoritmos Aproximados

Nesta Seção apresentamos de maneira resumida o tema Algoritmos de Aproximados, definindo a notação utilizada e também conceitos básicos sobre o tema. Nas definições a seguir, consideramos sempre problemas de minimização, que podem facilmente ser adaptadas para o caso de problemas de maximização.

Dado um algoritmo  $\mathcal{A}$  com complexidade de tempo polinomial e  $I$ , uma instância para este problema, denotaremos por  $\mathcal{A}(I)$  o custo da solução devolvida pelo algoritmo  $\mathcal{A}$  aplicado à  $I$  e  $OPT(I)$  o custo de uma solução ótima para  $I$ . Um algoritmo  $\mathcal{A}$  tem um fator de aproximação  $\alpha$  se  $\mathcal{A}(I) \leq \alpha OPT(I)$ , para qualquer  $I$ . Neste caso,  $\mathcal{A}$  é dito ser  $\alpha$ -aproximado ou uma  $\alpha$ -aproximação. Neste trabalho consideramos também algoritmos ditos assintoticamente  $\alpha$ -aproximados. Nestes casos, vale que  $\mathcal{A}(I) \leq \alpha OPT(I) + \beta$ , onde  $\alpha$  é o fator de aproximação e  $\beta$  é uma constante aditiva, para toda instância  $I$ . Assim, dize-

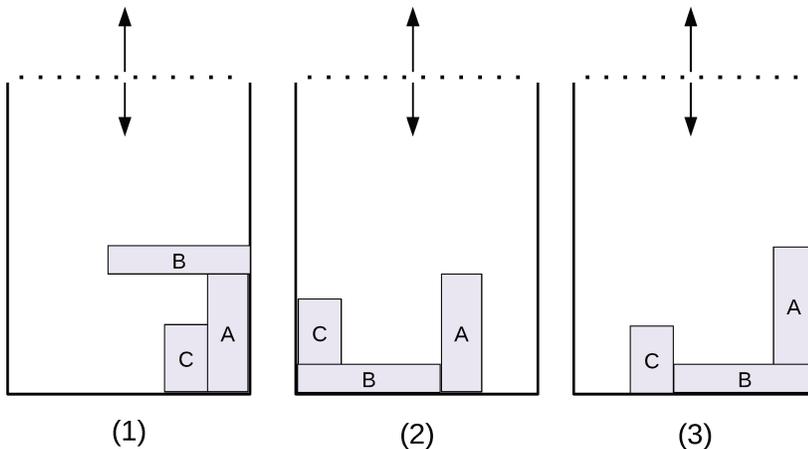


Figura 2.5: Exemplos de soluções inviáveis (1 e 2) e viável (3) para o 2KPLU.

mos que o algoritmo é assintoticamente  $\alpha$ -aproximado, pois  $\lim_{OPT(I) \rightarrow \infty} \sup_I \left( \frac{\mathcal{A}(I)}{OPT(I)} \right) \leq \alpha$ .

Um análise importante no desenvolvimento de algoritmos aproximados é verificar se o fator  $\alpha$  demonstrado é o melhor possível. Para tanto, devemos apresentar pelo menos uma instância onde vale que  $\mathcal{A}(I)/OPT(I) = \alpha + \epsilon$  (para  $\epsilon$  tão próximo de zero quanto se queira). Neste caso, o fator de aproximação do algoritmo  $\mathcal{A}$  não pode ser melhorado e é considerado justo.

Os algoritmos aproximados propostos na tese são combinatórios, ou seja, algoritmos que usam técnicas convencionais para o projeto de algoritmos, como algoritmos gulosos, programação dinâmica, etc. Nestes casos, não são empregadas técnicas específicas para o desenvolvimento de algoritmos de aproximação. Entretanto, nos últimos anos, surgiram várias técnicas para o desenvolvimento de algoritmos aproximados. Algumas delas são: *arredondamento de soluções via programação linear*, *dualidade em programação linear e método primal dual*, *algoritmos probabilísticos e sua desaleatorização*, *programação semidefinida*, *provas verificáveis probabilisticamente e a impossibilidade de aproximações*, dentre outras (veja [29, 51, 21, 49]).

Programação linear tem sido usada para a obtenção de algoritmos aproximados de diversas maneiras. Em geral os problemas são formulados utilizando-se programação linear inteira e a relaxação destes é resolvida, uma vez que isto pode ser feito em tempo polinomial. A partir disto, pode-se arredondar os valores fracionários da solução. Outra

técnica, é formular o dual do programa linear e obter soluções a partir das variáveis duais. No caso do uso da técnica primal-dual o algoritmo projetado é combinatório, porém fortemente baseado nas formulações primal e dual do problema.

No âmbito teórico, os algoritmos de aproximação mais desejados são os que possuem o menor fator de aproximação possível. Para alguns problemas, é possível mostrar que existem famílias de algoritmos com fatores de aproximação  $(1 + \epsilon)$ , no caso de problemas de minimização, e  $(1 - \epsilon)$ , no caso de problemas de maximização, onde  $\epsilon > 0$  é uma constante e pode ser tomada tão pequena quanto se deseje. Chamamos PTAS (*Polynomial Time Approximation Scheme*) uma família de algoritmos que têm tais fatores de aproximação e têm complexidade de tempo polinomial no tamanho da entrada. Se, além de serem polinomiais no tamanho da entrada, os algoritmos forem polinomiais em  $1/\epsilon$ , dizemos que são FPTAS (*Fully Polynomial Time Approximation Scheme*). Caso existam constantes aditivas nos fatores de aproximação destes algoritmos, chamaremos-los APTAS (*Asymptotic Polynomial Time Approximation Scheme*) e AFPTAS (*Asymptotic Fully Polynomial Time Approximation Scheme*), respectivamente. Nestes esquemas de aproximação temos uma família de algoritmos aproximados pois, dado um  $\epsilon > 0$  fixo, podemos construir um algoritmo com tal aproximação e tempo de execução polinomial. Dentre os esquemas supracitados, os FPTAS são os mais desejados.

Além de apresentar algoritmos de aproximação para problemas NP-difíceis, também podemos demonstrar que alguns deles não podem ser aproximados além de um certo fator  $\alpha$  (fator de inaproximabilidade). Nestes casos, dado um problema  $P$ , devemos demonstrar que não pode existir um algoritmo  $\alpha$ -aproximado para  $P$ . Uma das formas de fazê-lo é através de reduções, demonstrando que caso exista um algoritmo polinomial  $\alpha$ -aproximado para  $P$ , então podemos resolver algum outro problema NP-Difícil em tempo polinomial. Para mais detalhes em inaproximabilidade veja [1, 49].

Para determinados problemas ditos *online*, onde não é possível conhecer toda a instância  $I$  de entrada a priori, devemos projetar algoritmos que processem a entrada de maneira sequencial, à medida que ela lhe é apresentada, sem que seja possível modificar escolhas feitas em passos anteriores. A este tipo de algoritmos dá-se o nome de *algoritmos online*. Para mais detalhes sobre algoritmos online veja [5]. Um algoritmo  $\mathcal{A}$  é dito ser  $\alpha$ -competitivo se é *online* e é  $\alpha$ -aproximado. Nestes casos o valor  $\mathcal{A}(I)$  é comparado com o valor da solução ótima offline  $OPT(I)$ , a fim de obtermos uma aproximação.

## 2.4 Metaheurística GRASP

Nesta Seção descreveremos as principais ideias da metaheurística *GRASP*, as quais serão utilizadas em alguns dos algoritmos propostos nesta tese.

As metaheurísticas são procedimentos genéricos utilizados para guiar o desenvolvi-

mento de heurísticas para problemas específicos. Cada uma utiliza um mecanismo diferente para fugir de ótimos locais e tentam se aproximar ou encontrar alguma solução ótima global. Dentre as metaheurísticas mais conhecidas podemos citar os algoritmos genéticos [30], a Busca Tabu [27], *simulated annealing* [32], GRASP [19] e Otimização por Colônias de Formigas [9].

A metaheurística GRASP foi proposta por Feo *et al.* [19], mais formalmente em [20]. Esta metaheurística é guiada por um procedimento iterativo onde cada iteração é formada por duas fases: **Construção** e **Busca Local**. A fase de Construção ou fase construtiva cria uma solução  $s$  inicial viável, enquanto que a fase de *Busca Local*, por sua vez, busca soluções melhores na vizinhança de  $s$ . A melhor dentre todas as soluções encontradas nas iterações realizadas é devolvida como o resultado do algoritmo GRASP.

Na fase construtiva, é criada uma *Lista Restrita de Candidatos* (LRC) formada pelos elementos que, quando inseridos na solução parcial, levam a novas soluções de baixo custo. Após isto, é sorteado, aleatoriamente, um dos elementos da LRC e este é inserido na solução parcial. Este procedimento é repetido enquanto uma solução viável não é encontrada.

A *Busca Local* é um método simples usado para resolver problemas de otimização combinatória e, devido as suas limitações, na maioria dos casos, serve basicamente como apoio a outros algoritmos mais rebuscados como o GRASP. Seu processo básico é o seguinte. Dada uma solução inicial  $s$ , analisar a vizinhança de  $s$  (denotada por  $\mathcal{V}(s)$ ) em busca de uma solução de melhor valor. Se uma solução melhor for encontrada, então segue a busca pela vizinhança desta nova solução encontrada. Este processo é repetido até que a solução atual seja um ótimo local na vizinhança. As deficiências deste tipo simples de busca são amenizadas devido à característica de múltiplos pontos iniciais (*multistart*) do GRASP.

Nos algoritmos 1, 2 e 3 são apresentados pseudocódigos que sintetizam a metaheurística GRASP e suas fases.

---

**Algorithm 1** Fase-Construtiva
 

---

```

1: begin
2: input: A lista de elementos  $L$  do problema (candidatos).
3:  $Solução \leftarrow \emptyset$ 
4: while  $Solução$  não é viável do
5:   Calcule os custos incrementais de cada candidato.
6:   Construa a Lista Restrita de Candidatos  $LRC \subseteq L$ .
7:   Selecione aleatoriamente um elemento  $e \in LRC$ .
8:    $Solução \leftarrow Solução \cup \{e\}$ .
9:    $L \leftarrow L \setminus \{e\}$ .
10: return  $Solução$ .
11: end

```

---

---

**Algorithm 2** *Busca Local (Minimização)*

---

```

1: begin
2: input:  $s$  uma solução inicial.
3: while  $s$  não é mínimo local do
4:   Seja  $s' = \min(\mathcal{V}(s))$ .
5:    $s \leftarrow s'$ .
6: return  $s$ .
7: end

```

---



---

**Algorithm 3** GRASP

---

```

1: begin
2: input:  $MaxIter$ , o número máximo de iterações e  $L$  a lista de entrada do problema.
3:  $Melhor\_Solução \leftarrow \infty$ 
4: for  $i = 1$  to  $MaxIter$  do
5:    $Solução \leftarrow$  Fase-Construtiva( $L$ ).
6:    $Solução \leftarrow$  Busca-Local( $Solução$ ).
7:    $Melhor\_Solução \leftarrow \min(Melhor\_Solução, Solução)$ .
8: return  $Melhor\_Solução$ .
9: end

```

---

O algoritmo GRASP é executado por um número máximo de iterações ( $MaxIter$ ). Dentre as soluções geradas nas iterações do algoritmo é escolhida aquela de menor custo (no caso de problemas de minimização).

Heurísticas GRASP são consideradas gulosas pois sua fase construtiva gera soluções iterativamente com itens mais promissores. Por vezes são chamadas “semi-gulosas” pois esta escolha é feita de maneira aleatória a partir da LRC. Caso esta tenha apenas um elemento então o algoritmo seria puramente guloso. Este fator aleatório é muito importante para que soluções diferentes do espaço de soluções sejam encontradas e, portanto, ajuda na fuga de ótimos locais. Além disto, estas heurísticas são chamadas adaptativas pois os valores (custos) de cada candidato são recalculados no início de cada iteração da fase construtiva, tornando mais ou menos promissores dependendo da configuração atual do algoritmo.

As implementações GRASP geralmente usam valores fixos como parâmetros das escolhas aleatórias do algoritmo, entretanto isto pode ser melhorado com valores que se ajustem durante a execução da heurística. Nestes casos, ela é chamada GRASP *reativa* [43].

Outra técnica bastante utilizada em conjunto com o GRASP é o *Path-relinking*. Esta técnica foi proposta por Glover [28] para buscar melhores soluções no “caminho” entre boas soluções encontradas com *Busca Tabu* e *Scatter Search*. Esta técnica busca mes-

clar diferentes características das melhores soluções encontradas pelo GRASP gerando possivelmente, soluções melhores.

No trabalho [45] a metaheurística GRASP e suas melhorias são detalhadas. Neste trabalho, são apresentadas também as melhores formas de se aplicar cada uma destas técnicas (GRASP *reativa*, *Path-relinking*, etc).

# Capítulo 3

## Resumo dos Resultados

Nesta seção apresentamos os resultados obtidos durante o doutorado. Para tanto, descrevemos sucintamente os resultados apresentados em cada artigo dos capítulos seguintes. Além disto, discorreremos sobre dois trabalhos adicionais realizados durante o doutorado, porém que não estão inseridos como capítulos na tese (Seção 3.4).

### 3.1 A Note on a Two Dimensional Knapsack Problem With Unloading Constraints

No Capítulo 4 estamos interessados na versão bidimensional do *Two Dimensional Knapsack Problem with Unloading Constraints* (2KPU). Nele apresentamos os primeiros algoritmos aproximados para o problema em questão. Primeiramente, desenvolvemos um algoritmo híbrido, o qual combina dois tipos de algoritmos aproximados: Algoritmos para o 2KP bidimensional e algoritmos baseados em níveis para o *Two Dimensional Bin Packing Problem* bidimensional (2BPP). A partir deste algoritmo proposto e de sua análise, desenvolvemos algoritmos aproximados para 3 variantes do 2KPU ortogonal sem rotações. Uma  $(3 + \epsilon)$ -aproximação para o caso em que os itens e o *bin* são quadrados, um algoritmo  $(3 + \epsilon)$  aproximado para o caso em que o valor dos itens é igual a sua área e, por fim, uma  $(6 + \epsilon)$ -aproximação para o caso em que os itens são retangulares e o *bin* quadrado. Quando rotações são permitidas, estes fatores de aproximação foram reduzidos a  $2/3$  (Por exemplo, uma  $(4 + \epsilon)$ -aproximação para último caso citado, porém utilizando rotações). Por fim, apresentamos um algoritmo  $(4 + \epsilon)$  aproximado para o caso geral do problema sem rotações, utilizando o algoritmo *Next-Fit Decreasing Height*(NFDH) [31]. Ademais, mostramos que este fator de aproximação é justo.

Resultados preliminares foram publicados na conferência *Latin-American Algorithms, Graphs and Optimization Symposium - LAGOS* [12]. Uma versão completa aceita para pu-

blicação na revista *RAIRO - Theoretical Informatics and Applications* [47] é apresentada no Capítulo 4. Para facilitar a notação e a definição destes algoritmos, neste trabalho, utilizamos, excepcionalmente, o lado direito dos *bins* para a remoção dos itens. Isto ficará evidente na primeira figura Capítulo 4 (Figura 4.1).

## 3.2 Two Dimensional Strip Packing with Unloading Constraints

No Capítulo 5 desenvolvemos vários algoritmos aproximados para o *Strip Packing Problem with Unloading Constraints* (SPPU). Neste problema temos uma faixa  $F$  de largura um e altura infinita, e  $n$  itens divididos em  $C$  classes (ordens) diferentes, cada item  $a_i$  com altura  $h(a_i)$ , largura  $w(a_i)$  e classe  $c(a_i)$ . Assim como no SPP, desejamos empacotar os  $n$  itens minimizando a altura utilizada, porém agora consideramos a restrição de que itens de maior classe ( $c(a_i)$ ) não podem bloquear a saída de itens de menor classe. Supomos também que é permitida a rotação ortogonal dos itens.

De maneira geral, neste trabalho, atacamos duas versões do problema em questão. A primeira, como descrita acima e a segunda com uma restrição de descarregamento modificada (relaxada). Nesta segunda versão pode-se realizar dois movimentos para remover um item do *bin*, um horizontal e um vertical (ver a Figura 5.1 no Capítulo 5).

Para o caso em que podemos realizar movimentos horizontais e verticais para remover os itens, desenvolvemos um algoritmo com aproximação assintótica 3. Este algoritmo divide os itens em 4 tipos, baseado em seus tamanhos, e os empacota com uma versão modificada do NFDH. Desenvolvemos também um algoritmo para o caso paramétrico desta versão, onde os itens têm um largura limitada por um fator  $1/m$ ,  $m \geq 2$ . Este algoritmo é  $(\frac{m}{m-1} + \epsilon)$ -aproximado.

Para o caso onde só é permitido realizar movimentos verticais, apresentamos uma 5.745-aproximação assintótica. Este algoritmo utiliza um Teorema apresentado por Meir and Moser em [40] sobre a área dos itens empacotados com o NFDH. Por fim, também projetamos algoritmos para a versão paramétrica deste problema. Neste caso, supomos ambos, largura e altura, limitados por  $1/m$ ,  $m \geq 3$ , e então utilizamos o teorema proposto por Li e Cheng em [35] para desenvolver um algoritmo  $(\frac{m}{m-2})$ -aproximado.

Estes resultados melhoram aqueles obtidos pelo aluno e apresentados na sua dissertação de mestrado [11].

A versão apresentada no Capítulo 5 corresponde ao artigo completo aceito para publicação na revista *Discrete Applied Mathematics*. Uma versão preliminar do mesmo foi publicada na conferência *Latin-American Algorithms, Graphs and Optimization Symposium - LAGOS* [13].

### 3.3 On The Pickup and Delivery with Two Dimensional Loading/Unloading Constraints

No Capítulo 6 estudamos o *Pickup and Delivery Problem with Two Dimensional Loading/Unloading Constraints* (PDPLU), o qual surge como uma generalização do clássico *Pickup and Delivery Problem* (PDP) [41]. A entrada do PDPLU consiste em um grafo completo  $G$  com  $2n + 1$  vértices, sendo um depósito e  $n$  pares  $(p_i, d_i)$  (representando pontos de carregamento e descarregamento de um cliente  $i$ ), e uma lista de itens  $L_i$  associada a cada par de vértices  $(p_i, d_i)$ . O objetivo do problema é encontrar um ciclo hamiltoniano de custo mínimo onde cada vértice  $p_i$  é visitado antes do seu par  $d_i$  e existe um empacotamento válido para essa rota, respeitando as restrições do problema 2KPLU.

No 2KPLU temos basicamente a mesma instância do 2KPU porém ao invés de apenas um atributo de ordem ( $c(a_i)$ ) um item  $a_i$  terá dois atributos (carregamento  $p(a_i)$  e descarregamento  $d(a_i)$ ). Estes atributos indicam o momento em que um item será inserido (carregamento) e removido (descarregamento) do recipiente em questão. Neste caso abordaremos uma restrição a mais, a restrição de carregamento dos itens (modelando o problema de forma ideal para atacar o problema PDPLU).

Primeiramente, a fim de resolver o PDPLU de maneira exata, utilizamos uma versão simplificada do algoritmo *Branch-and-Cut* apresentado em [17] em conjunto com nossos algoritmos para o 2KPLU. Nossa abordagem foi testar, para cada rota gerada com o *Branch-and-Cut*, se existe um empacotamento válido para os itens, considerando a ordem definida pela rota, satisfazendo as restrições do 2KPLU. Testamos também algumas versões modificadas do algoritmo para o 2KPLU.

Além disto, continuamos os estudos com o desenvolvimento de heurísticas para o problema em questão. Primeiramente, adaptamos a melhor heurística da literatura [44] para o problema PDP, inserindo os algoritmos de empacotamento para o 2KPLU para validar as rotas geradas no algoritmo. Esta adaptação foi comparada com uma nova heurística GRASP que propusemos para o problema PDPLU. Esta é baseada na inserção de pares de vértices (clientes) na solução.

Para o 2KPLU, desenvolvemos dois algoritmos exatos. Um deles é uma extensão do algoritmo exato bidimensional que considera todos os pontos de canto para o empacotamento [39]. Na enumeração dos possíveis pontos onde pode-se empacotar um item deve-se remover alguns pontos inviáveis e adicionar outros baseado nas restrições de carregamento e descarregamento. Ademais, para definir os pontos de canto, devemos considerar apenas itens do empacotamento que certamente estarão juntos com o item sendo empacotado durante algum momento na rota considerada. Desta forma, veremos que em alguns empacotamentos teremos itens sobrepostos, mas somente em casos em que ambos não estarão empacotados ao mesmo tempo na rota. O outro algoritmo exato é baseado na formulação

com *Constraint Programming* (CP) apresentada por Malapert *et al.* [38].

Além disto, trabalhamos também em três heurísticas simples para o problema 2KPLU. Uma heurística baseada no algoritmo *Bottom Left* [2, 6], outra baseada no *Touching Perimeter* [37] e uma terceira que busca maximizar a sobreposição dos itens que não estarão juntos no *bin* ao mesmo tempo. A ideia básica de todas as heurísticas é perturbar a lista de itens de entrada e depois utilizar os algoritmos básicos considerando as restrições de carregamento e descarregamento. Ideias similares foram utilizadas no contexto do problema 2L-CVRP [26, 54].

Realizamos experimentos computacionais com várias versões destes algoritmos. Para isto, criamos novas instâncias para o PDPLU baseadas em instâncias do PDP já utilizadas na literatura [17]. Os resultados mostraram que o melhor algoritmo exato testado para o PDPLU foi o algoritmo que utiliza CP para o 2KPLU. Além disto, a heurística GRASP obteve consistentemente os melhores resultados dentre as heurísticas para o PDPLU.

O Capítulo 6 apresenta uma versão completa do artigo, que será submetido para publicação em uma revista.

## 3.4 Outros Trabalhos Feitos Durante o Doutorado

Ainda durante o doutorado participamos na elaboração de outros dois trabalhos aceitos que não estão incluídos como capítulos na tese por se tratarem de trabalhos que já apareceram em dissertações de mestrado. O primeiro foi a finalização do nosso trabalho de mestrado com resultados teóricos e práticos para o SPPU [10] (Seção 3.4.1). O segundo foi um trabalho na área de escalonamento em grades computacionais da dissertação de mestrado de Peixoto em [42] (Seção 3.4.2).

### 3.4.1 Artigo: *Heuristics for the strip packing problem with unloading constraints*

Neste trabalho estudamos o problema SPPU, já definido anteriormente. Nele apresentamos os primeiros resultados de aproximação para o SPPU. Do ponto de vista teórico, descrevemos uma 6.75 aproximação para o caso não orientado do SPPU e uma 1.75 aproximação para o caso orientado em que o número de classes dos itens é constante. Do ponto de vista prático, propusemos uma heurística GRASP para o problema e realizamos extensivos experimentos computacionais em novas instâncias criadas para o SPPU.

Durante o doutorado finalizamos uma pequena parte dos resultados teóricos do artigo e, além disto, implementamos uma grande parte dos experimentos, que foram solicitados para fins de aceitação do trabalho. Um artigo com estes resultados foi publicado na revista *Computers & Operations Research* [11].

### 3.4.2 Artigo: *On the Worst Case of Scheduling with Task Replication on Computational Grids*

Neste trabalho estudamos um problema de escalonamento que consiste em escalonar  $n$  tarefas em  $m$  máquinas onde, em geral, a velocidade das máquinas é desconhecida *a priori*. Este é um cenário típico em uma grade computacional, onde o poder de processamento varia de máquina para máquina e no decorrer do tempo.

Como resultados, apresentamos os primeiros algoritmos aproximados para a versão do problema que permite replicação de tarefas e que busca minimizar o *Total Processor Cycle Consumption* (TPCC) [24], além disto, mostramos que estes resultados são justos. Do ponto de vista prático, foram realizados extensivos experimentos computacionais com várias versões dos algoritmos analisados. Este estudo prático concluiu que a replicação induz um maior ganho em situações com grande variabilidade no poder de processamento das máquinas.

Durante o doutorado, tivemos que implementar e testar vários algoritmos estudados no artigo. Um artigo com estes resultados foi aceito para publicação na revista *Journal of Combinatorial Optimization* [53].



# Chapter 4

## *Artigo: A Note on a Two Dimensional Knapsack Problem With Unloading Constraints*<sup>1 2</sup>

J. L. M. da Silveira<sup>3</sup>   E. C. Xavier<sup>3</sup>   F. K. Miyazawa<sup>3</sup>

### Abstract

In this paper we address the two-dimensional knapsack problem with unloading constraints: we have a bin  $B$ , and a list  $L$  of  $n$  rectangular items, each item with a class value in  $\{1, \dots, C\}$ . The problem is to pack a subset of  $L$  into  $B$ , maximizing the total profit of packed items, where the packing must satisfy the unloading constraint: while removing one item  $a$ , items with higher class values can not block  $a$ .

Each item has an associated profit.

We present a  $(4 + \epsilon)$ -approximation algorithm when the bin is a square. We also present  $(3 + \epsilon)$ -approximation algorithms for two special cases of this problem.

**1991 Mathematics Subject Classification.** 68W25,05B40,90C27.

## 4.1 Introduction

In this paper we study the two-dimensional knapsack problem with unloading constraints (KU), that is a generalization of the well known NP-Hard two-dimensional knapsack

---

<sup>1</sup>*This research was supported by CNPQ and FAPESP*

<sup>2</sup>*Keywords and phrases: Knapsack Problem, Approximation Algorithms, Unloading/loading Constraints.*

<sup>3</sup>Institute of Computing, University of Campinas - UNICAMP, Av. Albert Einstein, 1251, Campinas, Brazil; jmoises@ic.unicamp.br & ecx@ic.unicamp.br & fkm@ic.unicamp.br

problem [1, 13]. This problem arises in operations research transportation problems, like the 2L-CVRP (Capacitated Vehicle Routing Problem with Two Dimensional Loading Constraints Problem) [6, 10]. In this problem we are given  $k$  identical vehicles, with a weight capacity and a rectangular surface that may be accessed only from one side. We are also given a graph that represents the customers, the distance between them and the starting point of the vehicles (a special vertex on the graph which represents the depot). Each customer has a demand given by a set of rectangular items. The goal is to find  $k$  routes that visits all clients such that the total cost of the routes is minimized. For each route it must be obtained a feasible packing of the items of the clients in the route: the unloading of items of a client must not be blocked by items of customers to be visited later along the route (unloading constraint).

The KU problem can be formally defined as follows: We are given a bin  $B$  of width and height 1, and  $n$  items of  $C$  different classes, each item  $a_i$  with height  $h(a_i)$ , width  $w(a_i)$ , profit  $p(a_i)$  and class  $c(a_i)$ . A packing is feasible if items do not *overlap*, all of them are packed inside bin  $B$ , and there is an *order* to unload the items, such that no item of a given class blocks the way out of other items of smaller classes. We consider that while removing one item of  $B$  only horizontal movements are allowed. The class values  $c$  represents the order in which items must be removed. While removing one item, only this item can be moved and only in the available free space of the bin. In this case, if an item  $a_i$  is packed in  $(x_i, y_i)$  (i.e. its bottom left corner is placed at this position) and  $a_j$  is packed in  $(x_j, y_j)$  and  $c(a_j) > c(a_i)$  then either  $y_j + h(a_j) \leq y_i$  or  $y_i + h(a_i) \leq y_j$  or  $x_j + w(a_j) \leq x_i$ . These constraints guarantee that item  $a_j$  is not blocking  $a_i$  during its removal while using only horizontal movements (see Fig. 4.1).

Since items are removed in non-decreasing order of values  $c$ , we can assume that, in a feasible packing, when removing item  $a_i$ , only items  $a_j$  with class  $c(a_j) \geq c(a_i)$  are still packed. The other items  $a_k$  with  $c(a_k) < c(a_i)$  should be removed previously. The value of a feasible solution is the sum of the profits of items packed in  $B$ .

The KU problem is a generalization of the classical two dimensional Knapsack Problem: a special case in which all items have the same class value  $c$ .

Denote by  $OPT(I)$  the cost of an optimal packing for the instance  $I$  and  $A(I)$  the cost of the solution computed by algorithm  $A$ . The proposed algorithms have polynomial-time complexity and satisfy  $\sup_I \frac{A(I)}{OPT(I)} \leq \alpha$ , where  $\alpha$  is the approximation ratio.

**Related Work.** An extended abstract of this work appeared in [3] and to our knowledge, there are no approximation algorithms for the KU problem. There are a few heuristics focused on the 2L-CVRP Problem [6, 10, 5, 16], and some heuristics for the strip packing version of the problem [4]. There is also an exact algorithm for the three dimensional version of the KU problem, based on an ILP formulation which take into account other practical constraints within the unloading constraint [14, 15].

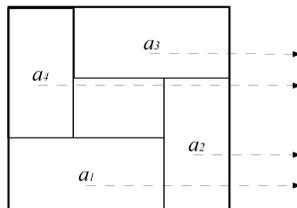


Figure 4.1: Suppose we have a squared bin  $B$  and four items  $a_i, i = 1, 2, 3, 4$  with  $c(a_2) = c(a_3) = 1$  and  $c(a_1) = c(a_4) = 2$ . Any 2-staged feasible solution for this problem uses at most 3 items, while the pattern above uses all 4 items and is feasible for the KU problem. If we had  $c(a_2) > c(a_1)$  then this packing would be infeasible since  $a_2$  would be blocking  $a_1$ .

#### Integer Linear Programming (ILP)

Furthermore, the recent advances in the two dimensional knapsack does not directly apply to the KU problem. In 2006, Harren [7] proposed a  $(5/4 + \epsilon)$ -approximation algorithm for the knapsack problem where the items are squares and have arbitrary profits and the bin is also a square. Jansen and Solis-Oba, proposed a PTAS for packing squares with independent profits into a *rectangle* [12]. There is a  $(2 + \epsilon)$ -approximation algorithm for packing rectangles with arbitrary profits [13]. In [11], Jansen and Prädél proposed a PTAS for the case in which the profits are proportional to the items surface. These results cannot be directly used in the KU problem because of the unloading constraints. It is worth noting that 2-staged knapsack solutions clearly satisfies the unloading constraints, but approximation algorithms for such problems do not lead directly to approximation algorithms for the KU problem. In [11], for instance, it is presented a PTAS for the geometrical 2-staged knapsack problem. But their result does not imply a PTAS to our problem, since optimal solutions for the KU problem may not satisfy the cutting pattern required in the 2-staged knapsack problem (See Fig. 4.1). Thus the approximation results for 2-staged problems can not be directly used.

**Main Results.** In this paper we present the first approximation algorithms for the KU problem. Our algorithm combines algorithms for two-dimensional knapsack problems and level based algorithms for the bin packing problem. We design approximation algorithms for 3 variations of the KU problem: a  $(3 + \epsilon)$ -approximation algorithm for the case where the goal is to maximize the profit of squares packed into a square; a  $(4 + \epsilon)$ -approximation algorithm for the general case where the goal is to maximize the profit of rectangles packed

into a square; and a  $(3 + \epsilon)$ -approximation algorithm for the geometrical case where the profits corresponds to the area of the rectangles. We also show some results for the case in which orthogonal rotations are allowed.

The paper is organized as follows. In Section 4.2 we present a general result that leads to three approximation algorithms. In Section 4.3 we present an  $(4 + \epsilon)$ -approximation algorithm for the general case. Finally, in Section 4.4 we draw some conclusions.

## 4.2 A Hybrid Algorithm for the KU problem

In this Section we present a general method to generate approximation algorithms for several versions of the KU problem. Using it we present approximation algorithms for three variations of the problem.

We define a *regular* two-dimensional knapsack problem, as a two-dimensional Knapsack problem in which:

- The items can not overlap.
- The items and the bin have rectangular form;
- Any level based packing is a valid (not necessarily optimal) solution to the problem;

For instance the classical Geometrical and Guillotined cases of the two-dimensional Knapsack problem are *regular*.

First we will prove a general result: given an algorithm for a *regular* two-dimensional knapsack problem and another level based algorithm for the two-dimensional bin packing problem, we can construct another algorithm for the same knapsack problem with the unload constraint.

By level based algorithm we mean an algorithm that packs items into levels (shelves), where items in some level are “separated” from items in other levels: all items in a level  $l_0 = 0$  are packed with their bottom at the bottom of the bin  $B$ , and the items of a subsequent level  $l_{i+1}$  are packed with their bottom in a line above all the items at level  $l_i$  such that  $l_{i+1} = l_i + \max_{a_k \in l_i} (h(a_k))$ . Notice that items in the same level can be arranged horizontally in an arbitrary manner without breaking the levels structure. The algorithms *First-Fit Decreasing Height* (FFDH) and *Next-Fit Decreasing Height* (NFDH) are examples of level based algorithms for the Strip packing problem [2] and the algorithm *Hybrid First-Fit* (HFF) is a level based algorithm for the two-dimensional bin packing problem [8].

Denote by  $p(B_i)$  the sum of the profits of items packed into the bin  $B_i$ , and denote by  $p(L)$  the sum of the profits of items in the list.  $L$

Input list of rectangles

**Theorem 4.2.1** *Let  $\mathcal{A}_K$  be an  $\alpha$ -approximation algorithm for a regular 2D knapsack problem. Let  $\mathcal{A}_{BP}$  be an absolute  $\beta$ -approximation level based algorithm for the 2D bin packing problem. Then there is an  $(\alpha\beta)$ -approximation algorithm, denoted by  $\mathcal{A}_{KU}$  for the same version of the knapsack problem and that respects the unloading constraints.*

Error note: Any  $\beta$  approximation is sufficient. The term *absolute* was misused here.

**Proof.** The Algorithm  $\mathcal{A}_{KU}$  is presented in Algorithm 4. It uses algorithm  $\mathcal{A}_K$  (an  $\alpha$ -approximation algorithm for a *regular* 2D knapsack problem) and algorithm  $\mathcal{A}_{BP}$  (an absolute  $\beta$ -approximation algorithm for the 2D bin packing problem that is based on levels).

In line 3 of Algorithm 4, algorithm  $\mathcal{A}_K$  selects a list of items  $L' \subseteq L$  as a solution for the knapsack problem. Then in line 4 of Algorithm 4, the level based algorithm  $\mathcal{A}_{BP}$  generates the packing  $P$  of the items in  $L'$  into bins. Finally, the Algorithm  $\mathcal{A}_{KU}$  selects the bin with the largest total profit among the created bins, sort each level in this bin by class and return it as a solution (lines 5-8 of Algorithm 4).

---

**Algorithm 4**  $\mathcal{A}_{KU}$  algorithm

---

- 1: **Input:** Algorithms  $\mathcal{A}_K$  and  $\mathcal{A}_{BP}$  and a list  $L$  of items of  $C$  different classes.
  - 2: **Begin**
  - 3:  $L' \leftarrow \mathcal{A}_K(L)$ ;
  - 4:  $P \leftarrow \mathcal{A}_{BP}(L')$ ;
  - 5: Let  $B_1, \dots, B_m$  be the bins in  $P$ .
  - 6: Let  $B$  be the bin with largest total profit  $p(B)$  among the bins in  $P$ .
  - 7: Sort items in each level of  $B$  by non-increasing order of class.
  - 8: return  $B$ .
  - 9: **end**
- 

First notice that the items in  $B$  are partitioned into levels since it is generated by a level based algorithm  $\mathcal{A}_{BP}$ . The items in each level of  $B$  are sorted by non-increasing order of class. This guarantees that the final packing satisfies the unloading constraint. Also, since the problem is *regular* then the items can be packed in levels.

Now consider the total profit in bin  $B$ . Denote by  $OPT_K$  the value of an optimal packing for the *regular* 2D knapsack problem, and  $OPT_{KU}$  the value of an optimal packing for the *regular* 2D knapsack version of the problem with unloading constraints. Given a list of items  $L$ , we have  $OPT_K(L) \geq OPT_{KU}(L)$ .

Assume that  $\mathcal{A}_{BP}(L')$  returned  $m$  bins. Since  $\mathcal{A}_{BP}$  is an absolute  $\beta$ -approximation algorithm, we have  $m \leq \beta$ . Let  $(B_1, \dots, B_m)$  be the set of bins returned by the algorithm  $\mathcal{A}_{BP}$ . We have

$$mp(B) \geq \sum_{j=1}^m p(B_j)$$

$$\begin{aligned}
&= p(L') \\
&\geq \frac{1}{\alpha} OPT_K(L) \\
&\geq \frac{1}{\alpha} OPT_{KU}(L).
\end{aligned}$$

Therefore,

$$p(B) \geq \frac{1}{m\alpha} OPT_{KU}(L) \geq \frac{1}{\alpha\beta} OPT_{KU}(L).$$

From the previous inequality, we conclude that  $\mathcal{A}_{KU}$  is an  $(\alpha\beta)$ -approximation algorithm for the 2D knapsack problem with unloading constraints. □

In [8], van Stee and Harren proved that the HFF algorithm, for the two-dimensional bin packing problem, is an absolute 3-approximation when bins are squares and items cannot be rotated, and is an absolute 2-approximation when 90 degree rotations are allowed. So, using the HFF algorithm, which is a level based algorithm, we can prove the following results applying Theorem 4.2.1:

- Let  $\mathcal{A}_K$  be the PTAS proposed in [12] for the 2D knapsack problem where items are squares, with profits, and the bin is also a square. Then there is a  $(3 + \epsilon)$ -approximation algorithm for the same version of the problem and that respects the unloading constraints. If rotations are allowed then the algorithm is a  $(2 + \epsilon)$ -approximation.
- Let  $\mathcal{A}_K$  be the  $(2 + \epsilon)$ -approximation algorithm proposed in [13] for the 2D knapsack problem, where items are rectangles with profits, and the bin is a square. Then there is a  $(6 + \epsilon)$ -approximation algorithm for the same 2D knapsack problem with unloading constraints. If rotations are allowed then the algorithm is a  $(4 + \epsilon)$ -approximation.
- Let  $\mathcal{A}_K$  be the PTAS proposed in [11] for the 2D knapsack problem where items are rectangles with profits that are proportional to their area (this problem is known as Geometrical Knapsack), and the bin is a square. Then there is a  $(3 + \epsilon)$ -approximation algorithm for the 2D Geometrical Knapsack problem with unloading constraints. If rotations are allowed then the algorithm is a  $(2 + \epsilon)$ -approximation.

Let  $n = |L|$  and  $T_{\mathcal{A}}(n)$  be the time complexity of algorithm  $\mathcal{A}$ . It is important to notice that the time complexity of the algorithm  $\mathcal{A}_{KU}$  is bounded by  $O(T_{\mathcal{A}_K}(n) + T_{\mathcal{A}_{BP}}(n) + O(n \log n))$ , with  $T_{\mathcal{A}_K}(n)$  from line 3 on Algorithm 4,  $T_{\mathcal{A}_{BP}}(n)$  from line 4 on

Algorithm 4 and  $O(n \log n)$  from line 7 on Algorithm 4. Thus, assuming  $\mathcal{A}_K$  and  $\mathcal{A}_{KU}$  being polynomial-time algorithms, then so is  $\mathcal{A}_{KU}$ .

### 4.3 A 4-approximation Algorithm for the KU Problem

In this section we present a  $(4 + \epsilon)$ -approximation algorithm for the general 2D KU problem, without rotations, where items have arbitrary profits and the bin is a square. This result improves the  $(6 + \epsilon)$ -approximation algorithm of the previous section.

Let  $\mathcal{A}_\epsilon$  be the  $(1+\epsilon)$ -approximation algorithm for the one-dimensional knapsack problem with arbitrary profits [9], and consider the NFDH algorithm [2], which is a level based algorithm for the strip packing problem.

With approximation factor of 2.

The NFDH starts by sorting the items by non-increasing order of height. Then, starting from the bottom of the strip as its first level, the algorithm packs items in order on the current level (at the bottom of the level and left justified) as long as they fit; then the current level is closed and the same procedure is applied to a new level above the current level while there are items to be packed.

In Algorithm 5 we present our  $\mathcal{A}'_{KU}$  algorithm for the KU Problem. It starts selecting a set  $L'$  of items with the algorithm  $\mathcal{A}_\epsilon$  where each 2D item  $a_i$  of the original instance is transformed into an item for the 1D knapsack problem: the size of the item,  $s(a_i)$ , has value equal to the area of  $a_i$  and the profit values are the same (line 3-4 of Algorithm 5). Then it uses the NFDH algorithm to generate a strip packing with the 2D items in  $L'$  (lines 5 of Algorithm 5). Finally the algorithm packs the levels generated by NFDH into 4 bins, and selects the one with the largest total profit as a final solution (lines 6-8 of Algorithm 5).

**Teorema 4.3.1** *The  $\mathcal{A}'_{KU}$  algorithm is a  $(4 + \epsilon)$ -approximation algorithm for the KU problem.*

**Proof.** Denote by  $Area(L)$  the total area of items in a given list  $L$ . First, notice that the algorithm  $A_\epsilon$  selects  $L'$  such that  $Area(L') \leq 1$  and  $p(L') \geq \frac{1}{1+\epsilon} OPT_K(L)$  where  $OPT_K(L)$  is the value of an optimal solution for the 2D knapsack problem.

Since  $NFDH(L') \leq 2Area(L') + 1$  (see [2]), we have that  $NFDH(L') \leq 3$ .

Let  $P$  be the strip packing solution computed by NFDH algorithm. We claim that there are at most three levels with height larger than  $1/2$  in this packing (the height of some level is the height of the highest item packed on it). The levels are sorted by height in  $P$ . Suppose for the purpose of contradiction that the first four levels have height  $> 1/2$ .

---

**Algorithm 5**  $\mathcal{A}'_{KU}$  algorithm

---

- 1: **Input:** A list  $L$  of 2D items of  $C$  different classes.
  - 2: **Begin**
  - 3: Let  $L_u$  be the list  $L$  of items where each item now has size  $s(a_i) = h(a_i) \cdot w(a_i)$ , and profits values remain the same.
  - 4: Let  $L'$  be the original 2D items selected by algorithm  $A_\epsilon(L_u)$ .
  - 5: Generate the packing  $P$  with NFDH( $L'$ ).
  - 6: Pack the levels of  $P$  into at most 4 bins.
  - 7: Let  $B$  be the bin with largest total profit  $p(B)$  among the 4 created bins.
  - 8: Sort each level in  $B$  by non-increasing order of class.
  - 9: return  $B$ .
  - 10: **end**
- 

Then all items packed in the first, second and third levels have height  $> 1/2$ . The first item packed in the fourth level also has height  $> 1/2$ . It is easy to see that the items of the first and second level have total area  $> 1/2$  and the items of the third level together with the first item of the fourth level also have total area  $> 1/2$ . But this contradicts the fact that  $Area(L') \leq 1$ , and so at most three levels have height larger than  $1/2$ .

Now we show how to pack all levels of  $P$  into at most 4 bins. Pack each level with height larger than  $1/2$  in three different bins  $B_1$ ,  $B_2$  and  $B_3$  (if there are less than 3 levels with height  $> 1/2$  consider the 3 largest levels). Now for the remaining levels pack them in the first bin until for the first time the total height is larger than 1, then proceed to the second bin, and finally to the third. Notice that only the first and second bins may have total height larger than 1, since the total height of all levels is at most 3. Pack the last packed levels of bin  $B_1$  and  $B_2$  in another bin  $B_4$  (this can be done since each one of these levels have height  $< 1/2$ ).

Finally the algorithm selects the bin  $B$  with largest total profit among  $B_1$ ,  $B_2$ ,  $B_3$  and  $B_4$ . The items in the levels of this bin are sorted in non-increasing order of class values. This guarantees that we have a feasible packing for the KU problem. Finally we have

$$\begin{aligned}
4 \cdot p(B) &\geq \sum_{i=1}^4 p(B_i) \\
&= p(L') \\
&\geq \frac{1}{1+\epsilon} OPT_K(L) \\
&\geq \frac{1}{1+\epsilon} OPT_{KU}(L).
\end{aligned}$$

That is,

$$p(B) \geq \frac{1}{4 + \epsilon'} OPT_{KU}(L)$$

Therefore, the algorithm  $\mathcal{A}_{KU}$  is a  $(4 + \epsilon)$ -approximation algorithm for the KU problem.  $\square$

Moreover, the analysis of Theorem 4.3.1 is tight. Consider the following instance with  $|L| = 7$  in the form  $(h(a_i), w(a_i), p(a_i), c(a_i))$ :

$$\left\{ (1, 2\epsilon, \alpha, 1), \left(\frac{1}{2} + 2\epsilon, 1 - \epsilon, \beta, 1\right), \left(\frac{1}{2} + \epsilon, 2\epsilon, \alpha, 1\right), \left(\frac{1}{4} + \epsilon, 1 - \epsilon, \beta, 1\right), \right. \\ \left. \left(\frac{1}{4}, 2\epsilon, \alpha, 1\right), \left(\frac{1}{8} + \epsilon, 1 - \epsilon, \beta, 1\right), \left(\frac{1}{8}, 2\epsilon, \alpha, 1\right) \right\}.$$

We can choose  $\epsilon$  small enough such that  $Area(L) < 1$ . We also choose a large value for  $\alpha$  and a small value for  $\beta$ . In line 4 of the algorithm  $\mathcal{A}'_{KU}$ , we will have  $L' = L$ . Furthermore, the NFDH algorithm just pile the items one above the other since it sorts the items in non-increasing order of height and uses next fit to pack items into levels.

Finally, the algorithm  $\mathcal{A}'_{KU}$  is going to use 4 bins to pack the list  $L'$  (See Fig. 4.2).

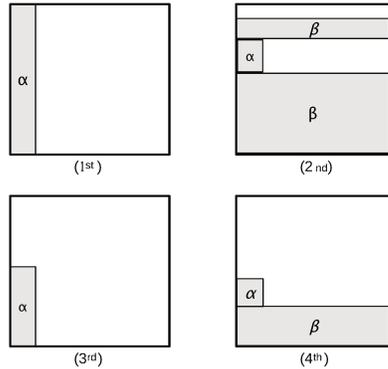


Figure 4.2: The algorithm uses the 4<sup>th</sup> bin since two items did not fit into bins 1 and 2.

After packing all the items into the bins, the  $\mathcal{A}'_{KU}$  algorithm chooses the 2<sup>nd</sup> bin with profit  $\alpha + 2\beta$ . An optimum solution packs all thin items (the ones with width  $2\epsilon$ ) into one bin, side by side with value  $4\alpha$ . Choosing  $\alpha \rightarrow \infty$  and  $\beta \rightarrow 0$  we have

$$\frac{OPT_{KU}(L)}{\mathcal{A}'_{KU}(L)} = \lim_{\alpha \rightarrow \infty, \beta \rightarrow 0} \frac{4\alpha}{\alpha + 2\beta} = 4.$$

Finally, let  $n = |L|$  and  $T_{A_\epsilon}(n)$  be the time complexity of algorithm  $A_\epsilon$ . The time complexity of the algorithm  $\mathcal{A}'_{KU}$  is bounded by  $O(T_{A_\epsilon}(n) + O(n \log n))$ , with  $T_{A_\epsilon}(n)$  from line 4 on Algorithm 5 and  $O(n \log n)$  from lines 5 and 8 on Algorithm 5. Thus, assuming  $A_\epsilon$  being a polynomial-time algorithm, so is  $\mathcal{A}'_{KU}$ .

## 4.4 Concluding Remarks

In this paper we consider some variants of the 2D knapsack problem with unloading constraints. To our knowledge, this is the first paper to present approximation results to this problem. We presented a  $(6 + \epsilon)$ -approximation algorithm for the general case of the problem, and two  $(3 + \epsilon)$ -approximation algorithms, one for the special case where the profits are proportional to the areas of the items and another one for the version where items are squares. Finally, we improve our first result for the general case and present a  $(4 + \epsilon)$ -approximation algorithm and proved that this result is tight.

## 4.5 References

- [1] A. Caprara and M. Monaci. On the two-dimensional knapsack problem. *Operations Research Letters*, 32:5 – 14, 2004.
- [2] E. G. Coffman Jr., M. R. Garey, D. S. Johnson, and R. E. Tarjan. Performance bounds for level-oriented two dimensional packing algorithms. *SIAM Journal on Computing*, 9(4):808–826, 1980.
- [3] J. L. M. da Silveira, E. C. Xavier, and F. K. Miyazawa. Two dimensional knapsack with unloading constraints. *Electronic Notes in Discrete Mathematics*, 37:267–272, 2011.
- [4] J. L. M. da Silveira, Eduardo C. Xavier, and Flávio Keidi Miyazawa. Heuristics for the strip packing problem with unloading constraints. *Computers & OR*, 40(4):991–1003, 2013.
- [5] B. L. P. de Azevedo, P. H. Hokama, F. K. Miyazawa, and E. C. Xavier. A branch-and-cut approach for the vehicle routing problem with two-dimensional loading constraints. In *Simpósio Brasileiro de Pesquisa Operacional - SBPO*, Porto Seguro, Brazil, 2009.
- [6] M. Gendreau, M. Iori, G. Laporte, and S. Martello. A tabu search heuristic for the vehicle routing problem with two-dimensional loading constraints. *Networks*, 51(1):4–18, October 2007.
- [7] R. Harren. Approximating the orthogonal knapsack problem for hypercubes. In *ICALP (1)*, volume 4051 of *Lecture Notes in Computer Science*, pages 238–249, 2006.
- [8] R. Harren and R. van Stee. Absolute approximation ratios for packing rectangles into bins. *Journal of Scheduling*, 2010.

- [9] O. H. Ibarra and C. E. Kim. Fast approximation algorithms for the knapsack and sum of subset problems. *Journal of the ACM*, 22:463–468, October 1975.
- [10] M. Iori, J.-J. S-González, and D. Vigo. An exact approach for the vehicle routing problem with two-dimensional loading constraints. *Transportation Science*, 41(2):253–264, May 2007.
- [11] K. Jansen and L. Prädél. How to maximize the total area of rectangles packed into a rectangle. Technical Report 0908, Christian-Albrechts-Universität zu Kiel, Italy, 2009.
- [12] K. Jansen and R. Solis-Oba. A polynomial time approximation scheme for the square packing problem. In *Proc. of the Integer Programming and Combinatorial Optimization*, volume 5035 of *LNCS*, pages 184–198. Springer Berlin / Heidelberg, 2008.
- [13] K. Jansen and G. Zhang. On rectangle packing: maximizing benefits. In *Proc. of the 15th ACM-SIAM Symposium on Discrete Algorithms*, pages 204–213, Philadelphia, PA, USA, 2004. Society for Industrial and Applied Mathematics.
- [14] L. Junqueira, R. Morabito, and D. Yamashita. Abordagens para problemas de carregamento de contêineres com considerações de múltiplos destinos. *Gestão & Produção.*, 18(1), 2011.
- [15] L. Junqueira, R. Morabito, and D. S. Yamashita. Mip-based approaches for the container loading problem with multi-drop constraints. *Annals of Operations Research*, 199(1):51–75, 2012.
- [16] E. E. Zachariadis, C. T. Kiranoudis, and C. D. Tarantilis. A guided tabu search for the vehicle routing problem with two-dimensional loading constraints. *European Journal of Operational Research*, 195:729–743, 2009.



# Chapter 5

## *Artigo: Two Dimensional Strip Packing with Unloading Constraints*<sup>1</sup>

J. L. M. da Silveira<sup>2</sup>    E. C. Xavier<sup>2</sup>    F. K. Miyazawa<sup>2</sup>

### **Abstract**

In this paper we present approximation algorithms for the two dimensional strip packing problem with unloading constraints. In this problem, we are given a strip  $S$  of width 1 and unbounded height, and  $n$  items of  $C$  different classes, each item  $a_i$  with height  $h(a_i)$ , width  $w(a_i)$  and class  $c(a_i)$ . As in the strip packing problem, we have to pack all items minimizing the used height, but now we have the additional constraint that items of higher classes cannot block the way out of lower classes items. For the case in which horizontal and vertical movements to remove the items are allowed, we design an algorithm whose asymptotic performance bound is 3. For the case in which only vertical movements are allowed, we design a bin packing based algorithm with asymptotic approximation ratio of 5.745. Moreover, we also design approximation algorithms for restricted cases of both versions of the problem. These problems have practical applications on routing problems with loading/unloading constraints.

## **5.1 Introduction**

In this paper we study two variants of the strip packing problem with unloading constraint, that are generalizations of the well known NP-Hard strip packing problem. These

---

<sup>1</sup>*This research was supported by CNPQ and FAPESP*

<sup>2</sup>Institute of Computing, University of Campinas - UNICAMP, Av. Albert Einstein, 1251, Campinas, Brazil; jmoises@ic.unicamp.br & ecx@ic.unicamp.br & fkm@ic.unicamp.br

problems arise in operations research transportation problems, where items are delivered along a route. In these problems, it is necessary to consider the order in which items are packed in a vehicle in order to minimize the effort while unloading it [6, 7, 3, 4, 8].

In the Strip Packing Problem with Unloading Constraint (SPU), we are given a strip  $S$  of width 1 and unbounded height, and  $n$  items of  $C$  different classes, each item  $a_i$  with height  $h(a_i)$ , width  $w(a_i)$  and class  $c(a_i)$ . A packing is a feasible solution, if items do not intersect, all of them are packed inside the strip, and there is an order to unload the items, such that no item of larger class blocks the way out of other items. The class values  $c$  represent the order in which items must be removed. While removing one item, only this item can be moved and only in the available free space of the strip. Since items are removed in increasing order of values  $c$ , we can assume that, in a feasible packing, when removing item  $a_i$ , only items  $a_j$  with class  $c(a_j) \geq c(a_i)$  are still packed. The other items  $a_k$  with  $c(a_k) < c(a_i)$  should be removed previously. In one version of the problem, denoted by  $\text{SPU}_v$ , only vertical movements are allowed while removing one item. In this case, if an item  $a_i$  is packed in  $(x_i, y_i)$  (i.e. its bottom left corner is placed at this position) and  $a_j$  is packed in  $(x_j, y_j)$  and  $c(a_j) > c(a_i)$  then either  $x_j + w(a_j) \leq x_i$  or  $x_i + w(a_i) \leq x_j$  or  $y_j + h(a_j) \leq y_i$ . These constraints ensure that the item  $a_j$  is not blocking  $a_i$  during its removal while using only vertical movements (see Fig. 5.1, parts c and d). The cost of a feasible solution is the height of the used area of the strip.

We also consider the variant (denoted by  $\text{SPU}_{vh}$ ) in which one horizontal movement and then one vertical movement are allowed while removing an item (see Fig 5.1, parts a and b). In our algorithms we assume that orthogonal rotation of the items is allowed unless it is said otherwise.

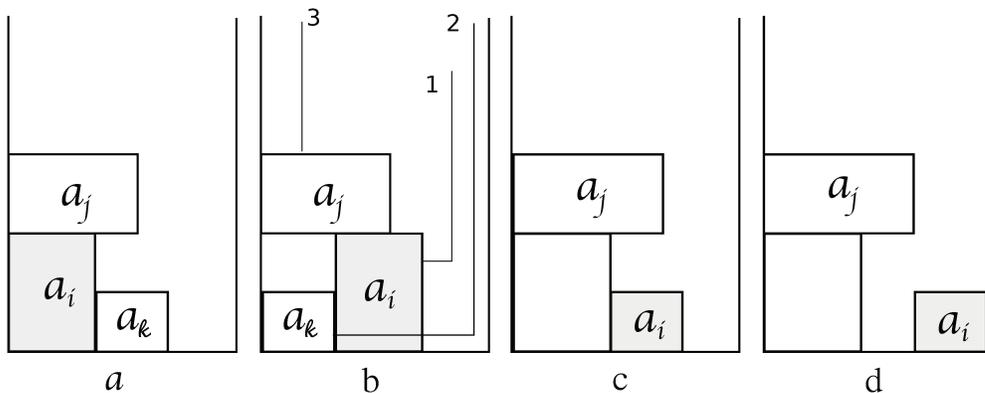


Figure 5.1: Suppose that  $c(a_j) > c(a_k) > c(a_i)$ . (a): An infeasible solution to  $\text{SPU}_{vh}$ , because  $a_k$  and  $a_j$  are blocking  $a_i$ . (b): A feasible placement to  $\text{SPU}_{vh}$ , where the items can be removed in order 1, 2 and 3. (c): This placement is not a feasible solution to  $\text{SPU}_v$ , because  $a_j$  is blocking  $a_i$ . (d): A feasible placement to  $\text{SPU}_v$ , where the items can be removed in order.

In [1], Azar and Epstein proposed an online 4-competitive algorithm to a version of the strip packing problem, where while packing one item there must be a free way from the top of the bin until the position where the item is packed. In this model, a rectangle arrives from the top of  $S$  as in the well known *TETRIS game*, and it should be moved continuously using only the free space until it reaches its place. In this case both horizontal and vertical movements are allowed. Notice that in the final solution, each item (in reversal order) can be removed from the strip using only the free space, since they were packed in order. If we consider that items can perform both horizontal and vertical movements, their online algorithm can be easily modified to an *offline* algorithm to the problem  $\text{SPU}_{vh}$ . If we sort the list  $L$  of items by non-increasing order of class values we get  $L'$ , and then if we use their algorithm in  $L'$  we find a feasible solution to  $\text{SPU}_{vh}$ , since each item  $a_i$  had reached its place when there were packed only items of class greater than or equal to  $c(a_i)$ . Since it could be packed in order, it can also be removed in order. This way, we easily devise an *offline* 4-approximation algorithm for the  $\text{SPU}_{vh}$  problem. The algorithm we propose for the  $\text{SPU}_{vh}$  problem is somewhat similar to the one presented in [1]. In general we use the idea of generating shelves and packing them in an order such that their width do not block other items to be packed.

In [5], Fekete, Kamphans and Schweer, proposed a 2.6154-competitive online algorithm for the strip packing problem, where items must be squares. In this algorithm, the items are packed from the top of  $S$  and are moved only with vertical movements to reach its final position. In addition, an item is not allowed to move upwards and has to be supported from below when reaching its final position. These conditions are called *gravity constraints*. Their slot based algorithm can be easily used to the  $\text{SPU}_v$  problem, achieving a 2.6154-approximation, in the special case where items are squares. We just need to sort the items in non-increasing order of class values. To the authors knowledge the best online algorithm for the general case of rectangles for the strip packing problem, is still the result of Azar and Epstein [1].

author's

In [2], Silveira et. al. presented several heuristics for the SPU problem, and among them, a 6.75-approximation algorithm. In this work we improve this result presenting a 3-approximation algorithm for the  $\text{SPU}_{vh}$  problem. In [7], Iori et. al. tackled the 2L-CVRP problem where the SPU appears as a sub-problem. To solve the packing problem they used the bottom-left heuristic and a branch-and-bound procedure to check the feasibility of the loadings. Finally in [8] Junqueira et. al. presented integer programming models for a 3D version of the problem, where one has to decide if 3D boxes fit in a 3D container, and the packing satisfies unloading constraints.

Given an algorithm  $\mathcal{A}$  to the SPU problem, and an instance  $I$ , we denote by  $\mathcal{A}(I)$  the cost of the solution computed by  $\mathcal{A}$  over the instance  $I$ . We denote by  $\text{OPT}(I)$  the height

used by an optimum packing of  $I$ . The proposed algorithms to the SPU problem are asymptotically bounded, thus they satisfy  $\mathcal{A}(I) \leq \alpha OPT(I) + \beta$ , where  $\beta$  is a constant and  $\alpha$  is the asymptotic approximation ratio.

In this paper we present an algorithm with asymptotic approximation ratio 5.745 to the SPU <sub>$v$</sub>  problem and another one with approximation ratio 3 to the SPU <sub>$vh$</sub>  problem when rotations are allowed. We show an algorithm for the parametric cases of both problems. For the SPU <sub>$vh$</sub>  problem we design an algorithm for the oriented case in which the rectangles have width bounded by  $1/m$ , where  $m \geq 2$ . This algorithm has asymptotic ratio  $(\frac{m}{m-1} + \varepsilon)$  plus an additive constant of  $\frac{2+\varepsilon}{\varepsilon}$ . For the parametric case of the SPU <sub>$v$</sub>  problem, in which the rectangles have width bounded by  $1/m$ ,  $m \geq 3$ , we design an asymptotic  $(\frac{m}{m-2})$ -approximation algorithm.

## 5.2 An algorithm for the SPU <sub>$vh$</sub> problem

In this section we describe a 3-approximation algorithm, denoted by  $\mathcal{A}_{vh}$ , for the SPU <sub>$vh$</sub>  problem. We assume the constraint that items must be removed in order from the final packing using vertical and horizontal movements in the available free space of the strip. We assume that both width and height of each rectangle is bounded by 1. First, we present two algorithms that will be used as routines in the  $\mathcal{A}_{vh}$  algorithm.

Consider the following four types of items:

- $\mathcal{L}$ : items  $a_i$  with  $h(a_i)$  and  $w(a_i) > 2/3$ .
- $\mathcal{T}$ : items  $a_i$  with  $h(a_i) > 2/3$  and  $w(a_i) < 1/3$ .
- $\mathcal{M}$ : items  $a_i$  with any  $h(a_i)$  and  $2/3 \geq w(a_i) \geq 1/3$ .
- $\mathcal{S}$ : items  $a_i$  with  $h(a_i)$  and  $w(a_i) < 1/3$ .

It is easy to see that one can properly rotate an item such that it fits into one of these four item types.

We present now a modified version of the classical NFDH (Next Fit Decreasing Height) algorithm, called MNFDH (*Modified* NFDH) (Alg. 6). The NFDH algorithm generates a packing divided into horizontal shelves, each shelf has height equal to the maximum rectangle height among the rectangles in the shelf. Rectangles packed in a same shelf are packed side by side. As in the classical NFDH algorithm, the MNFDH pack the items into horizontal shelves, but the rule to close a shelf is modified. Also, the height of each shelf is pre-determined, given by a parameter  $h$ . The MNFDH algorithm deals with items of width strictly smaller than  $w$  and generate shelves of height  $h$  ( $w$  and  $h$  are parameters

---

**Algorithm 6** *Modified NFDH*

---

- 1: **Input:** Parameters  $w$  and  $h$ , and a list  $L$  of items with width smaller than  $w$ .
  - 2: **Begin**
  - 3:  $\mathcal{F} \leftarrow \emptyset$ .
  - 4: Let  $F$  be a shelf at the bottom of  $S$  (shelf 0) of height  $h$  and width  $w(F) = 0$ .
  - 5: **for** each item  $a \in L$  in the order occurring in the input **do**
  - 6:   pack  $a$  into  $F$  at the left-most position.
  - 7:    $w(F) = w(F) + w(a)$
  - 8:   **if**  $w(F) \geq 1 - w$  **then**
  - 9:      $\mathcal{F} \leftarrow \mathcal{F} \cup F$ .
  - 10:   Close the actual shelf  $F$  and create a new shelf  $F$ , with  $w(F) = 0$  and height  $h$ , above the last one.
  - 11: If the last created shelf has width smaller than  $1 - w$  (non-full), then let  $\mathcal{F}^{NF}$  be a set containing this shelf, otherwise  $\mathcal{F}^{NF}$  is empty.
  - 12: **Return:**  $(\mathcal{F}, \mathcal{F}^{NF})$ .
  - 13: **end.**
- 

of this algorithm). In the MNFDH algorithm, a shelf  $F$  is closed if its width becomes large enough, specifically  $w(F) \geq 1 - w$ .

Denote each closed shelf as a *full* shelf (Alg. 6, line 10) and *non-full* otherwise. Notice that all the generated shelves are *full*, except possible the last one, since each item  $a$  has  $w(a) < w$ . Furthermore, if we remove the last item (the right-most one) of each *full* shelf  $F$ , then  $w(F) < 1 - w$ , since once  $w(F) \geq 1 - w$  the shelf is closed.

Now we present an algorithm called *Base* (Alg. 7) which deals with items of types  $\mathcal{M}$ ,  $\mathcal{T}$  and  $\mathcal{S}$ . The algorithm MNFDH is used by the *Base* algorithm as a routine to pack items of types  $\mathcal{T}$  and  $\mathcal{S}$ . The algorithm *Base* starts rotating the items  $a$  of types  $\mathcal{S}$  and  $\mathcal{T}$ , such that  $h(a) \geq w(a)$  and items of type  $\mathcal{M}$  such that,  $2/3 \geq w(a) \geq 1/3$  (Line 3). After that, it generates five sets of shelves, based on the types of each item (Lines 8 - 10). The items of types  $\mathcal{S}$  and  $\mathcal{T}$  are packed with the MNFDH algorithm, and the items of type  $\mathcal{M}$  are just piled left justified on the strip  $S$  (packed in individual shelves). The shelves containing one item of type  $\mathcal{M}$ , are considered *full*. Finally, it generates the final packing by properly sorting and packing the sets of shelves created, aiming to attend the unloading constraint (Lines 11 - 14 and Fig. 5.2).

We prove two lemmas concerning the *Base* algorithm. The first one (Lemma 5.2.1), deals with the unloading constraint and Lemma 5.2.2 deals with the fraction of occupied area of the strip  $S$ .

**Lemma 5.2.1** *The packing produced by Base satisfies the constraints of  $SPU_{vh}$*

**Algorithm 7** *Base*

- 
- 1: **Input:** List  $L$  of items of types  $\mathcal{T}$ ,  $\mathcal{M}$  and  $\mathcal{S}$ , partitioned into  $C$  classes.
  - 2: **Begin**
  - 3: Rotate the items  $a$  of type  $\mathcal{S}$  and  $\mathcal{T}$  such that  $h(a) \geq w(a)$  and the items of type  $\mathcal{M}$  such that,  $2/3 \geq w(a) \geq 1/3$ .
  - 4: Let  $L_P = \{a \in L: a \text{ has type } \mathcal{S}\}$
  - 5: Let  $L_T = \{a \in L: a \text{ has type } \mathcal{T}\}$
  - 6: Let  $L_M = \{a \in L: a \text{ has type } \mathcal{M}\}$
  - 7: Partition the set  $L_P$  into  $L_{P_0}, L_{P_1}, \dots, L_{P_k}$ , such that,  $a \in L_{P_i}$  iff  $\frac{1}{2^{i.3}} \geq h(a) > \frac{1}{2^{i+1.3}}$ .
  - 8:  $(\mathbf{P}_i^F, \mathbf{P}_i^{NF}) \leftarrow \text{MNFDH}(L_{P_i}, 1/3, \frac{1}{2^{i.3}})$ , for  $0 \leq i \leq k$
  - 9:  $(\mathbf{T}^F, \mathbf{T}^{NF}) \leftarrow \text{MNFDH}(L_T, 1/3, 1)$
  - 10: Sort items of  $L_M$  in non-increasing order of class and pack each item  $a \in L_M$  in an individual shelf left justified. Denote this packing by  $\mathbf{M}$ .
  - 11: For each shelf in  $\mathbf{T}^F, \mathbf{T}^{NF}, \mathbf{P}_i^F$  and  $\mathbf{P}_i^{NF}$  ( $0 \leq i \leq k$ ) sort its items by non-increasing order of class (Starting from left).
  - 12: Let  $\mathbf{F} \leftarrow \mathbf{T}^F \cup \mathbf{P}_1^F, \cup \dots \cup \mathbf{P}_k^F$ .
  - 13: Sort the shelves in  $\mathbf{F}$  in non-increasing order of class value of the right-most item. Pack in  $S$  the shelves of  $\mathbf{F}$  in this order.
  - 14: Pack in  $S$ ,  $\mathbf{T}^{NF}, \mathbf{P}_i^{NF}, 0 \leq i \leq k$  and  $\mathbf{M}$ , in this order.
  - 15: **Return:** The packing in  $S$ .
  - 16: **end.**
- 

**Proof.** We are going to show that any item can be removed from the strip  $S$  using one horizontal movement and then one vertical movement.

If an item  $a \in \mathbf{M}$  then it can be removed from  $S$  since the items that are packed above  $a$  have lower class values (Line 10). Now consider an item  $a \in \mathbf{T}^{NF} \cup (\mathbf{P}_i^{NF}, 1 \leq i \leq k)$ . In this case, the items in the same shelf of  $a$  that are to its right, will be removed before  $a$  (Line 11). Moreover, since  $a$  can reach the right-boundary of  $S$ , it can be removed from  $S$ , since all strips  $F$  above it have  $w(F) \leq 2/3$  (Line 14). Finally, consider an item  $a \in \mathbf{T}^F \cup (\mathbf{P}_i^F, 1 \leq i \leq k)$ , packed in a shelf  $F$ . Since the items in  $F$  are sorted, from left to right, by non-increasing value of class,  $a$  can reach the right-most side of  $S$  (Line 10). Furthermore, since the shelves are sorted, starting from the bottom of  $S$ , in non-increasing order of class value of the right-most item in the shelf, we can conclude that at least one item (the right-most one) of each shelf above  $F$ , in  $\mathbf{T}^F \cup (\mathbf{P}_i^F, 1 \leq i \leq k)$ , will be removed before  $a$ . Thus,  $a$  can be removed using the right-most side of  $S$ . In the moment of removing  $a$ , each shelf above  $F$  has width strictly smaller than  $1 - w = 2/3$ .  $\square$

**Lemma 5.2.2** *The shelves in the sets  $\mathbf{F}$  and  $\mathbf{M}$  generated by the Base algorithm have at least  $1/3$  of occupied area.*

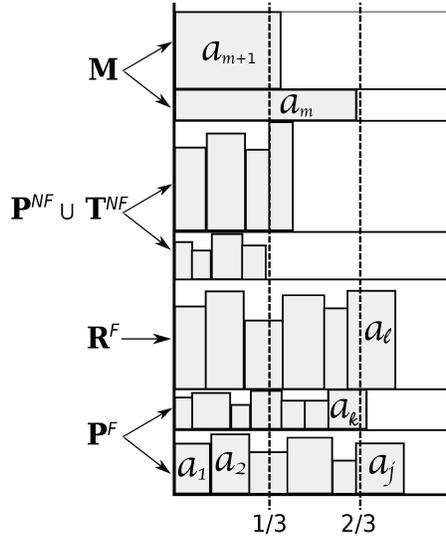


Figure 5.2: The packing generated by the *Base* algorithm. Items in each shelf are sorted by class in non-increasing order (i.e.  $c(a_1) \geq c(a_2) \geq \dots \geq c(a_j)$ ). Furthermore, the shelves in  $\mathbf{P}^F \cup \mathbf{T}^F$  are sorted by the class of the right most item (i.e.  $c(a_j) \geq c(a_k) \geq c(a_l)$ ). Finally, the items in  $\mathbf{M}$  are sorted by class in non-increasing order (i.e.  $c(a_m) \geq c(a_{m+1})$ ).

**Proof.** We are going to show that the shelves in  $\mathbf{T}^F$ ,  $(\mathbf{P}_i^F, 0 \leq i \leq k)$  and  $\mathbf{M}$ , have at least  $1/3$  of occupied area.

The shelves  $F$  with one item  $a$  of type  $\mathcal{M}$  have  $w(a) \geq 1/3$  and  $h(F) = h(a)$ . We conclude each one of these shelves has at least,  $w(a) \cdot h(a)/h(F) \geq 1/3$  of occupied area.

The shelves  $F \in \mathbf{T}^F$  have  $h(F) = 1$  and  $w(F) \geq 1 - w = 2/3$ . They have, at least  $(2/3)^2 > \frac{1}{3}$  of occupied area, since items of type  $\mathcal{T}$  have height strictly larger than  $2/3$ . Finally, the shelves in  $\mathbf{P}_i^F$  have height  $\frac{1}{2^{i+1} \cdot 3}$  and are used to pack items of height larger than  $\frac{1}{2^{i+1} \cdot 3}$ , so, they have, at least

$$\frac{(1 - w) \cdot \frac{1}{2^{i+1} \cdot 3}}{\frac{1}{2^i \cdot 3}} = \frac{1}{3}$$

of occupied area. □

Now we present the algorithm  $\mathcal{A}_{vh}$  (Alg. 8) that computes the packing of all items. The algorithm starts sorting the list  $L$  in non-increasing order of class value with ties broken by widest first (Line 4). Then the algorithm proceeds selecting a maximal prefix of items of types  $\mathcal{S}$ ,  $\mathcal{T}$  and  $\mathcal{M}$ , which are going to be packed with the algorithm *Base* (Lines 10 to 12). After that (Lines 13 to 15), the algorithm selects and removes a maximal prefix of items of type  $\mathcal{L}$ , which are going to be packed in individual shelves (like the items of type  $\mathcal{M}$  in *Base* algorithm). The next step of the algorithm is to push in front of  $L$

the *non-full* shelves which do not brake the unloading constraint (Lines 16 to 19).

break

Then the algorithm proceeds by packing the selected items and shelves (Lines 20 and 21). The algorithm ends when there are no more items to be packed.

---

**Algorithm 8**  $\mathcal{A}_{vh}$ 


---

- 1: **Input:** List  $L$  of  $n$  items partitioned into  $C$  classes.
  - 2: **Begin**
  - 3: Assign each item of  $L$  into one of the types  $\mathcal{L}$ ,  $\mathcal{T}$ ,  $\mathcal{M}$  and  $\mathcal{S}$  doing rotations if necessary.
  - 4: For each item  $a_i \in \mathcal{L} \cup \mathcal{S}$  rotate it such that  $h(a_i) \geq w(a_i)$ ,  $1 \leq i \leq n$ .
  - 5: Sort items of  $L$  in non-increasing order of class value. Ties are broken by widest first.
  - 6: Let  $B = \{b_1, \dots, b_j\}$  be a maximal prefix of  $L$  containing only items of type  $\mathcal{L}$ .
  - 7: Pack each item of  $B$  left justified, in individual shelves at the bottom of  $S$  in order.
  - 8:  $L \leftarrow L \setminus B$
  - 9: **while**  $L \neq \emptyset$  **do**
  - 10:   Let  $\{a_1, \dots, a_i\}$  be a maximal prefix of  $L$  containing only items of types  $\mathcal{S}$ ,  $\mathcal{T}$  or  $\mathcal{M}$ .
  - 11:   Let  $\mathcal{F} \leftarrow Base(\{a_1, \dots, a_i\})$
  - 12:    $L \leftarrow L \setminus \{a_1, \dots, a_i\}$
  - 13:   Let  $B = \{b_1, \dots, b_j\}$  be a maximal prefix of items of type  $\mathcal{L}$  in  $L$ .
  - 14:    $L \leftarrow L \setminus B$
  - 15:   Let  $b$  be the widest item in  $B$
  - 16:   **for** each *non-full* shelf  $F \in \mathcal{F}$  **do**
  - 17:     **if**  $w(F) + w(b) \leq 1$  **then**
  - 18:        $\mathcal{F} \leftarrow \mathcal{F} \setminus \{F\}$ .
  - 19:       Reinsert each item  $a' \in F$  in the beginning of  $L$ , keeping them sorted by non-increase value of class.
  - 20:   Pack  $\mathcal{F}$  in  $S$  in order.
  - 21:   Pack items of  $B$  in  $S$  left justified, each item in an individual shelf above all previous shelves.
  - 22: **Return:** The generated packing.
  - 23: **end.**
- 

**Lemma 5.2.3** *The packing produced by  $\mathcal{A}_{vh}$  satisfies the constraints of the  $SPU_{vh}$  problem.*

**Proof.** We are going to prove that the unloading constraint is satisfied as a loop invariant for line 9 of the  $\mathcal{A}_{vh}$  algorithm.

Notice that before the first iteration we have an empty solution or a sequence of type  $\mathcal{L}$  items piled left justified on  $S$ , which is trivially feasible. Now suppose that the algorithm

is starting its  $i$ th iteration of the while loop, and assume that the current solution is feasible. Let  $\mathcal{F}$  be the set of shelves generated by *Base* algorithm in this iteration, and  $\{b_1, \dots, b_j\}$  be the set of items of type  $\mathcal{L}$  that will be packed in this iteration. By Lemma 5.2.1 the packing is feasible considering only the items in  $\mathcal{F}$ . A problem might occur when considering items of types  $\mathcal{T}$  and  $\mathcal{S}$  in  $\mathcal{F}$  with classes higher than the classes of the items packed in previous iterations. These items came from *non-full* shelves  $F$  that were not packed in previous iterations because  $w(F) + w(b) \leq 1$  (for some item  $b$  of type  $\mathcal{L}$ ). Notice that  $w(F) \leq 1 - w(b) \leq 1/3$  (Line 17). Then, each new shelf  $F'$  in  $\mathcal{F}$  of type  $\mathcal{T}$  or  $\mathcal{S}$  contains at most  $w' = 1/3$  of width occupied with these higher class items. Then these higher class items could only block previously packed items of type  $\mathcal{L}$ . But this does not occur since  $w' + w(b) \leq 1$  for all items  $b$  of type  $\mathcal{L}$  of lower classes. □

### 5.2.1 $\mathcal{A}_{vh}$ Analysis

**Lemma 5.2.4** *The set of shelves generated by the  $\mathcal{A}_{vh}$  algorithm has at least  $1/3$  of occupied area, except for  $5/3$  of height.*

**Proof.** Notice that the *full* shelves have at least  $1/3$  of occupied area by Lemma 5.2.2.

So we just need to prove that the *non-full* shelves and the shelves with items of type  $\mathcal{L}$  also have, at least  $1/3$  of occupied area on average. We are going to associate each *non-full* shelf, that was not removed in the loop of line 16 of the  $\mathcal{A}_{vh}$  algorithm, with the largest item of type  $\mathcal{L}$  above it, that was packed in the same iteration of the while loop (line 9). Notice that at any given iteration, the *Base* algorithm generates at most one *non-full* shelf  $\mathbf{T}^{NF}$  of type  $\mathcal{T}$  and at most one *non-full* shelf  $\mathbf{P}_i^{NF}$  for each  $i \geq 0$ , that packs items of type  $\mathcal{S}$ .

The total height of  $\mathbf{P}_i^{NF}$  shelves is at most

$$\sum_{i=0}^{\infty} \frac{1}{2^i \cdot 3} = 2/3$$

The total height of an item of type  $\mathcal{T}$  is 1.

Error: Total height of a *shelf* containing an item of type  $\mathcal{T}$ .

If no item of type  $\mathcal{L}$  is packed above these non-full shelves, then this is the last iteration. So at most  $5/3$  of height of *non-full* shelves will not be associated with any item of type  $\mathcal{L}$ .

Consider some iteration where *non-full* shelves are packed, and items of type  $\mathcal{L}$  are packed above them. Let  $b$  be the widest item of type  $\mathcal{L}$  packed in this iteration above these *non-full* shelves. We associate non-full shelves uniquely with  $b$ .

The shelf  $B$  that contains  $b$  has  $h(B) = h(b)$  and  $w(B) = w(b) \geq 2/3$ . The possible shelf  $\mathbf{T}^{NF}$  has  $h(\mathbf{T}^{NF}) = 1$  and  $w(\mathbf{T}^{NF}) \geq 1 - w(b)$ , otherwise it would not be packed in this iteration (lines 17-19 of  $\mathcal{A}_{vh}$ ). Furthermore each item  $a$  in  $\mathbf{T}^{NF}$  has  $h(a) \geq 2/3$ . Finally, each shelf  $\mathbf{P}_i^{NF}$  has  $h(\mathbf{P}_i^{NF}) = \frac{1}{2^i \cdot 3}$  and also  $w(\mathbf{P}_i^{NF}) \geq 1 - w(b)$ . Moreover each item  $a$  in  $\mathbf{P}_i^{NF}$  has  $h(a) > \frac{1}{2^i + 1.3}$ .

We can bound the occupied area in the shelves  $B$ ,  $\mathbf{T}^{NF}$ , and all  $\mathbf{P}_i^{NF}$  by

$$\frac{h(b) \cdot w(b) + [1 - w(b)] \cdot 2/3 + [1 - w(b)] \cdot \frac{sh}{2}}{h(b) + 1 + sh} \quad (5.1)$$

where  $2/3 < w(b) \leq h(b) \leq 1$  and  $0 \leq sh \leq 2/3$  ( $sh$  is the total height of  $\mathbf{P}_i^{NF}$  shelves. The minimum of this function occurs when  $w(b) = h(b) = 2/3 + \epsilon$  and  $sh = 2/3$  and has value  $> 1/3$  (see Appendix 5.8.1).  $\square$

**Theorem 5.2.5** *Let  $L$  be a list of rectangles, then  $\mathcal{A}_{vh}(L) \leq 3 \cdot OPT(L) + 5/3$ .*

**Proof.** Let  $S$  be the solution returned by the algorithm  $\mathcal{A}_{vh}$ . It is filled by *full* shelves, and type  $\mathcal{L}$  items with its associated *non-full* shelves. Due to Lemma 5.2.4, the strip  $S$  have at least  $1/3$  of occupied area on average, except perhaps by  $5/3$  of height. We can conclude that  $(\mathcal{A}_{vh}(L) - 5/3) \cdot 1/3 \leq \sum_{a_i \in L} h(a_i) \cdot w(a_i)$  and then

$$\mathcal{A}_{vh}(L) \leq 3 \cdot OPT(L) + 5/3$$

$\square$

### 5.3 An Algorithm to a parametric oriented case of the $\text{SPU}_{vh}$ Problem

In this version of the problem, we are going to assume that the items can not be rotated and the width of the items is bounded by  $1/m$ ,  $m \geq 2$ . Since the approximation analysis is based on area arguments the result remain valid for the case in which rotations are allowed.

The algorithm is called  $\mathcal{A}_{vh}^p$  (Alg. 9) and takes two parameters: the factor  $m$  and a constant  $\epsilon$ . The algorithm starts by partitioning the input list  $L$  by the height of the items. Then it generates a set of strips for each partition using the algorithm MNFDH (Alg. 6). Finally it proceeds by packing and sorting the *full* shelves and then packing the *non-full* ones.

**Algorithm 9**  $\mathcal{A}_{vh}^p$ 

- 
- 1: **Input:** List  $L$  of items partitioned into  $C$  classes,  $m$  and  $\epsilon$ .
  - 2: **Begin**
  - 3:  $\mathcal{P} \leftarrow \emptyset$  and  $\mathcal{N} \leftarrow \emptyset$
  - 4: Let  $r = \frac{1}{1+\epsilon/2}$
  - 5: Let  $L_i = \{a : a \in L \text{ and } r^{i+1} < h(a) \leq r^i\}$ , for  $i \geq 0$ .
  - 6: **for** each  $i$  **do**
  - 7:  $(\mathcal{F}, \mathcal{F}') \leftarrow \text{MNFDH}(L_i, 1/m, r^i)$
  - 8:  $\mathcal{P} \leftarrow \mathcal{P} \cup \mathcal{F}$
  - 9:  $\mathcal{N} \leftarrow \mathcal{N} \cup \mathcal{F}'$
  - 10: Sort items in each shelf in  $\mathcal{P} \cup \mathcal{N}$  by non-increasing order of class (Starting from left).
  - 11: Sort the shelves in  $\mathcal{P}$  by non-increasing order of class of the right-most item.
  - 12: Pack  $\mathcal{P}$  in  $S$  in order and then pack  $\mathcal{N}$  in  $S$  above  $\mathcal{P}$ .
  - 13: **Return:** The generated packing.
  - 14: **end.**
- 

**Lemma 5.3.1** *The packing produced by  $\mathcal{A}_{vh}^p$  satisfies the constraints of  $SPU_{vh}$*

**Proof.** This proof is similar to the proof of the Lemma 5.2.1. □

### 5.3.1 $\mathcal{A}_{vh}^p$ Analysis

**Theorem 5.3.2** *Let  $L$  be a list of rectangles, then  $\mathcal{A}_{vh}^p(L) \leq \left(\frac{m}{m-1} + \epsilon\right) \cdot OPT(L) + \frac{2+\epsilon}{\epsilon}$ .*

**Proof.** First, consider some shelf  $F \in \mathcal{P}$ . Since this shelf is *full*, we have that  $w(F) \geq 1 - 1/m$  and, moreover, suppose that  $F$  was created to pack items from the list  $L_i$ , which means that this shelf has, at least  $\frac{r^{i+1}}{r^i} = r$  of occupied height. Then we can bound the occupied area in  $F$  by  $r(1 - 1/m) = \frac{r(m-1)}{m}$ . Also notice that there is, at most one shelf  $F \in \mathcal{N}$  with items from  $L_i$ . So we can bound the total height of strips in  $\mathcal{N}$  by  $\sum_{i=0}^{\infty} r^i = \frac{1}{1-r} = \frac{2+\epsilon}{\epsilon}$

Finally we have  $(\mathcal{A}_{vh}^p(L) - \frac{2+\epsilon}{\epsilon}) \cdot \frac{r(m-1)}{m} \leq \sum_{a_i \in L} h(a_i) \cdot w(a_i)$  and then

$$\mathcal{A}_{vh}^p(L) \leq \left(\frac{(1 + \epsilon/2) \cdot m}{m-1}\right) \cdot OPT(L) + \frac{2+\epsilon}{\epsilon} \leq \left(\frac{m}{m-1} + \epsilon\right) \cdot OPT(L) + \frac{2+\epsilon}{\epsilon}$$

□

## 5.4 An Algorithm to the $\text{SPU}_v$ Problem

In this Section we present an algorithm, denoted by  $\mathcal{A}_v$  to solve the  $\text{SPU}_v$  problem when only vertical movements of items are allowed and orthogonal rotation of items is allowed. We are going to use the NFDH algorithm. In [10], Meir and Moser proved the following result for the NFDH:

**Theorem 5.4.1** *Any list of rectangles  $L = \{a_1, \dots, a_k\}$  with total area  $A$  can be packed by the NFDH algorithm into a unit square if  $1 \geq w(a_i) \geq h(a_i)$ , for  $i = 1, \dots, k$ ;  $h(a_i) \geq h(a_{i+1})$ , for  $i = 1, \dots, k - 1$  and  $A \leq \frac{7}{16}$ .*

The algorithm  $\mathcal{A}_v$  computes the solution in two stages. First it packs the items into bins of height 1 and width 1 (unit square bins), using the NFDH algorithm. The algorithm also packs some large items alone in one bin. The bins used to pack these large items have height 1 and width equal to the width of the item. Then it packs the bins rotated into  $S$  (the strip) such that each shelf created by the NFDH becomes vertical, but with limited height 1, the maximum width of the bin. We first show the bin packing algorithm named *Bin Packing Decreasing Order* (BPDO) in Algorithm 10, and then we present the  $\mathcal{A}_v$  in Algorithm 11.

Denote by  $B_k$  the  $k$ th bin created by the algorithm BPDO,  $h(B_k)$  its occupied height and  $w(B_k)$  the occupied width. Also denote by  $A(L) = \sum_{a_i \in L} h(a_i)w(a_i)$ , where  $L = (a_1, \dots, a_m)$  is a list of items.

The BPDO algorithm starts by sorting the items in non-increasing order of class values (line 4). Then it selects the largest prefix of items that can be packed in a bin  $B$  of height 1 and width 1 using the NFDH algorithm (line 6). By Theorem 5.4.1 these items can be packed in a single bin (line 9). After packing these items, the algorithm sorts each shelf created by the NFDH algorithm by class values (line 10). The algorithm then removes the packed items from the list of items (line 11). If the first item  $a_{k+1}$  of the remaining list has  $w(a_{k+1})h(a_{k+1}) \geq 0.263422$ , the algorithm packs it alone in a new bin  $B'$ , of width  $w(a_{k+1})$  (line 14). The algorithm repeats this process until  $L = \emptyset$ .

The algorithm  $\mathcal{A}_v$  is presented in Algorithm 11. It just calls the algorithm BPDO, merge all bins returned side by side forming a strip of height 1 and width equal to the total width of the bins. The strip is rotated to provide a solution to the original problem.

**Theorem 5.4.2** *The packing produced by  $\mathcal{A}_v$  satisfies the constraints of  $\text{SPU}_v$*

**Proof.** Let  $B_1, \dots, B_m$  be the bins created by BPDO in the order they were created. These bins are packed in the strip  $S$  in the order they were created. For each successive pair of bins  $B_k$  and  $B_{k+1}$  we guarantee that all items in  $B_{k+1}$  have class smaller than

---

**Algorithm 10** Bin Packing Decreasing Order (BPDO)

---

```

1: Input: List  $L$  of items partitioned into  $C$  different classes.
2: Begin
3: Let  $LB = \emptyset$  be a list of bins.
4: Sort items in  $L$  in non-increasing order of class value.
5: while ( $L \neq \emptyset$ ) do
6:   Let  $L' = (a_1, \dots, a_k)$  be the largest prefix of  $L$  such that  $A(L') \leq 7/16 < A(L' \cup \{a_{k+1}\})$ .
7:   Rotate the items  $a_i$  in  $L'$  such that  $w(a_i) \geq h(a_i)$ .
8:   Sort  $L'$  in non-increasing order of height.
9:   Pack  $L'$  using the NFDH algorithm in a new bin  $B$ .
10:  Sort items in each generated shelf in non-increasing order of class value
11:   $L \leftarrow L \setminus L'$  and  $LB \leftarrow LB + B$ 
12:  if ( $A(\{a_{k+1}\}) \geq 0.263422$ ) then
13:    Rotate  $a_{k+1}$  such that  $h(a_{k+1}) \geq w(a_{k+1})$ 
14:    Pack  $a_{k+1}$  in a new bin  $B'$  of height 1 and width  $w(a_{k+1})$ .
15:     $L \leftarrow L \setminus \{a_{k+1}\}$  and  $LB \leftarrow LB + B'$ 
16: Return  $LB$ .
17: end.

```

---



---

**Algorithm 11**  $\mathcal{A}_v$ 

---

```

1: Input:  $L = \{a_1, a_2, \dots, a_n\}$ 
2: Begin
3: Let  $B_1, B_2, \dots, B_m$  be the bins computed by BPDO in the order they were created.
4: Concatenate  $B_1, B_2, \dots, B_m$  forming one strip  $S$  of height 1 and width  $\sum_{k=1}^m w(B_k)$ .
5: Return  $S$  rotated such that its width is 1 and its height is  $\sum_{k=1}^m w(B_k)$ .
6: end.

```

---

or equal to the items in  $B_k$ , since the algorithm packs items in non-increasing order of classes. So items in one bin will not block items in previous bins. Inside each bin, the feasibility of the solution is guaranteed by the packing in shelves. Items in each shelf are sorted by non-increasing order of class value. Each shelf generated by the NFDH algorithm is rotated in the final solution (line 5 of Alg. 11). Also notice that different shelves does not interfere with each other, since all items are packed completely inside a shelf.  $\square$

### 5.4.1 $\mathcal{A}_v$ Analysis

In this Section we use arguments based on the occupied area of each bin to prove the approximation of the  $\mathcal{A}_v$  algorithm.

**Lemma 5.4.3** *Let  $B_1, \dots, B_m$  be the bins computed by BPDO in the order they were created. Then  $B_1, \dots, B_{m-1}$  have occupied area of at least 0.174 on average.*

**Proof.** We will prove that the bins created in each iteration of the main loop (line 5) have at least 0.174 of occupied area on average (except perhaps the last created bin). Consider some iteration of the main loop at line 5. Let  $B_j$  be the  $j$ th-created bin using NFDH( $L'$ ) at line 9, where  $L' = \{a_1, \dots, a_k\}$ .

**Case 1:** Consider that  $A(\{a_{k+1}\}) < 0.263422$ . Since  $A(L') \leq 7/16 < A(L' \cup \{a_{k+1}\})$  we have

$$A(L') > 7/16 - A(\{a_{k+1}\}) > 7/16 - 0.263422 \approx 0.174078$$

and all items in  $L'$  are packed in  $B_j$  by Theorem 5.4.1. Since  $A(\{a_{k+1}\}) < 0.263422$  only this bin is created on this iteration of the loop.

**Case 2:** Consider that  $A(\{a_{k+1}\}) \geq 0.263422$ . Then  $a_{k+1}$  is rotated, such that  $h(a_{k+1}) \geq w(a_{k+1})$ , and it is packed in a new bin  $B_{j+1}$ , where  $w(B_{j+1}) = w(a_{k+1})$ .

The total width occupied in the two bins  $B_j$  and  $B_{j+1}$  is at most  $(1 + w(a_{k+1}))$  and the total area of the items packed in these two bins is  $h(a_{k+1})w(a_{k+1}) + A(L')$ . Then we can bound the occupied area in the occupied width of bins  $B_k$  and  $B_{k+1}$  by

$$\frac{w(a_{k+1})h(a_{k+1}) + A(L')}{1 + w(a_{k+1})} \tag{5.2}$$

where  $1 \geq h(a_{k+1}) \geq w(a_{k+1}) > 0$ ,  $A(\{a_{k+1}\}) > 0.263422$  and  $0 < A(L') \leq 7/16$ . The minimum of this function occurs when  $w(a_{k+1}) = h(a_{k+1}) \approx 0.51325$ ,  $A(L') = 0$  and has value  $\approx 0.174077 > 0.174$  (see 5.8.2).  $\square$

**Theorem 5.4.4** *Let  $L$  be a list of rectangles, then,  $\mathcal{A}_v(L) \leq 5.745OPT(L) + 1$ .*

**Proof.** Due to Lemma 5.4.3, each bin created by the algorithm has on average 0.174 of occupied area in the corresponding width, except perhaps the last generated bin. Since the total width of the bins corresponds to the total height of the used strip, we have  $(\mathcal{A}_v(L) - 1)0.174 \leq \sum_{a_i \in L} w(a_i) \cdot b(a_i)$  and then

$$\mathcal{A}_v(L) \leq 5.745 \sum_{a_i \in L} w(a_i) \cdot b(a_i) + 1 \leq 5.745OPT(L) + 1.$$

$\square$

## 5.5 An algorithm to a parametric version of the $SPU_v$ problem

In this Section we present an algorithm for a special version of the  $SPU_v$  problem where the width and the height of the items are bounded by  $1/m$ ,  $m \geq 3$  and the items can be rotated. The algorithm presented in this section is similar to the  $\mathcal{A}_v$  algorithm, but even simpler. In this case we are going to use a result presented by Li and Cheng [9].

**Theorem 5.5.1** *A list  $L$  of rectangles  $L = \{a_1, \dots, a_k\}$ , can be packed in a unit square bin by the NFDH algorithm if exists an integer  $m \geq 3$  such that:*

1.  $h(a_i) \leq \frac{1}{m}$ ,  $1 \leq i \leq k$
2.  $w(a_i) \leq \frac{1}{m}$ ,  $1 \leq i \leq k$
3.  $\sum_{i=1}^k h(a_i)w(a_i) \leq \left(1 - \frac{1}{m}\right)^2$

The algorithm for this restricted case is called  $\mathcal{A}_v^p$  (Alg. 12) and takes only one parameter, the value of  $m$ . The algorithm starts by sorting the input list  $L$  in non-increasing order of class value (Line 4). Then, while exist items to be packed, it selects the largest prefix of items that can be packed in a unit square bin  $B$  using the NFDH algorithm (line 6). This prefix is selected based on Theorem 5.5.1 (line 9). After packing this prefix, the algorithm sorts items in each shelf created by the NFDH algorithm by class values (line 10). The algorithm repeats this process until  $L = \emptyset$ . Finally, it joins the created bins forming the final strip  $S$  (Line 11).

**Lemma 5.5.2** *The packing produced by  $\mathcal{A}_v^p$  satisfies the constraints of  $SPU_v$*

**Proof.** This proof is similar to the proof of the Theorem 5.4.2. □

### 5.5.1 $\mathcal{A}_v^p$ Analysis

**Theorem 5.5.3** *Let  $L$  be a list of rectangles, then  $\mathcal{A}_v^p(L) \leq \left(\frac{m}{m-2}\right) \cdot OPT(L) + 1$ .*

**Proof.** Notice that each bin  $B_i \in LB$  has at least  $\left(1 - \frac{1}{m}\right)^2 - \frac{1}{m^2} = \frac{m-2}{m}$  of occupied area, since  $A(B_i) + h(a)w(a) > \left(1 - \frac{1}{m}\right)^2$  for some item  $a$  (Line 6).

Then we have  $(\mathcal{A}_v^p(L) - 1) \cdot \frac{m-2}{m} \leq \sum_{a_i \in L} h(a_i) \cdot w(a_i)$  and then

$$\mathcal{A}_v^p(L) \leq \left(\frac{m}{m-2}\right) \cdot OPT(L) + 1$$

□

---

**Algorithm 12**  $\mathcal{A}_v^p$ 

---

- 1: **Input:** List  $L$  of items partitioned into  $C$  different classes.
  - 2: **Begin**
  - 3: Let  $LB = \emptyset$  be a list of bins.
  - 4: Sort items in  $L$  in non-increasing order of class value.
  - 5: **while** ( $L \neq \emptyset$ ) **do**
  - 6: Let  $L' = (a_1, \dots, a_k)$  be the largest prefix of  $L$  such that  $A(L') \leq (1 - \frac{1}{m})^2 < A(L' \cup \{a_{k+1}\})$ .
  - 7: Rotate the items  $a_i$  in  $L'$  such that  $w(a_i) \geq h(a_i)$ .
  - 8: Sort items in  $L'$  in non-increasing order of height.
  - 9: Pack  $L'$  using the NFDH algorithm in a new bin  $B$ .
  - 10: Sort items in each generated shelf in non-increasing order of class value.
  - 11: Concatenate the list  $LB = \{B_1, \dots, B_m\}$  in order, forming one strip  $S$  of height 1 and width  $\sum_{k=1}^m w(B_k)$ .
  - 12: **Return**  $S$  rotated such that its width is 1 and its height is  $\sum_{k=1}^m w(B_k)$ .
  - 13: **end.**
- 

## 5.6 Concluding Remarks

In this paper we consider a variant of the two dimensional strip packing problem where we have constraints on how items can be removed from the strip. These are called unloading constraints and appear in vehicle routing problems where items are delivered along a route. We presented an algorithm with asymptotic performance bound 5.745 for the  $\text{SPU}_v$  problem when rotations are allowed. We also presented a 3-approximation algorithm for the  $\text{SPU}_{vh}$  problem when rotations are allowed. Finally, we also presented two algorithms for parametric cases of both problems. For the parametric  $\text{SPU}_{vh}$  problem, where the rectangles have width bounded by  $1/m$ ,  $m \geq 2$ , we proposed an algorithm with asymptotic ratio  $(\frac{m}{m-1} + \varepsilon)$  plus an additive constant of  $\frac{2+\varepsilon}{\varepsilon}$ . For the parametric  $\text{SPU}_v$  problem, in which the rectangles have both width and height bounded by  $1/m$ ,  $m \geq 3$ , we presented an asymptotic  $(\frac{m}{m-2})$ -approximation algorithm.

## 5.7 References

- [1] Y. Azar and L. Epstein. On two dimensional packing. *Journal of Algorithms*, 25(2):290 – 310, 1997.
- [2] J. L. M. da Silveira, F. K. Miyazawa, and E. C. Xavier. Heuristics for the strip packing problem with unloading constraints. *Computers & Operations Research*, 40(4):991–1003, 2013.

- [3] B. L. P. de Azevedo, P. H. Hokama, F. K. Miyazawa, and E. C. Xavier. A branch-and-cut approach for the vehicle routing problem with two-dimensional loading constraints. In *Simpósio Brasileiro de Pesquisa Operacional - SBPO*, Porto Seguro, Brazil, 2009.
- [4] K. F. Doerner, G. Fuellerer, R. F. Hartl, and M. Iori. Ant colony optimization for the two-dimensional loading vehicle routing problem. *Computers & Operations Research*, 36:655–673, 2009.
- [5] Sándor P. Fekete, Tom Kamphans, and Nils Schweer. Online square packing with gravity. *Algorithmica*, pages 1–26, 2012.
- [6] M. Gendreau, M. Iori, G. Laporte, and S. Martello. A tabu search heuristic for the vehicle routing problem with two-dimensional loading constraints. *Networks*, 51(1):4–18, 2008.
- [7] M. Iori, J.-J. S-Gonzalez, and D. Vigo. An exact approach for the vehicle routing problem with two-dimensional loading constraints. *Transportation Science*, 41(2):253–264, 2007.
- [8] L. Junqueira, R. Morabito, and D. S. Yamashita. Mip-based approaches for the container loading problem with multi-drop constraints. *Annals of Operations Research*, 199(1):51–75, 2012.
- [9] K. Li and K.-H. Cheng. On three-dimensional packing. *SIAM Journal on Computing*, 19:847–867, 1990.
- [10] A. Meir and L. Moser. On packing of squares and cubes. *Journal of Combinatorial Theory Series A*, 5:116–127, 1968.

## 5.8 Appendix

### 5.8.1 Minimizing function (5.1)

**Proof.** We have to find the minimum of the function

$$f(x, y, z) = \frac{xy + (z/2 + 2/3) \cdot (1 - y)}{x + z + 1}$$

where  $1 \geq y \geq x \geq 2/3$  and  $2/3 \geq z \geq 0$ .

This function has no critical points in the considered region. Moreover the function  $\frac{\partial f}{\partial z} = \frac{x(3-9y)+y-1}{6(x+z+1)^2} < 0$ , in the considered region. Thus the minimum of this function can be found on the limits of the region, in this case when  $z = 2/3$ .

So we just need to consider the function

$$g(x, y) = \frac{3y(x - 1) + 3}{3x + 5}$$

over the regions  $x = 1$ ,  $x = y$  and  $y = 2/3$ .

When  $x = 1$  the function  $g$  has value  $\frac{3}{8}$ . If  $x = y$ , then we must minimize the function  $\frac{3(1-x+x^2)}{5+3x}$ ,  $2/3 < x \leq 1$ , which has a point of minimum when  $x = 2/3$ , which leads to a minimum of  $1/3$ . Finally, when  $y = 2/3$ , we must minimize the function  $\frac{2x+1}{3x+5}$ ,  $2/3 < x \leq 1$ , which has minimum when  $x = 2/3$ , which leads, again, to  $1/3$ .

Thus, the minimum of  $f$  relies on  $x = y = z = 2/3$  and has value  $1/3$ . □

### 5.8.2 Minimizing function (5.2)

**Proof.** Since  $A(L')$  only adds area to the function we can easily see that the minimum of the function occurs when  $A(L') = 0$  and so we have to find the minimum of the function

$$f(x, y) = \frac{xy}{x + 1}$$

where  $1 \geq y \geq x > 0$  and  $xy > 0.263422$  (see the analyzed region in Fig. 5.3).

The critical points of this function can be found with  $\frac{\partial f}{\partial x} = 0$  and  $\frac{\partial f}{\partial y} = 0$  and we have the point  $(x, y) = (0, 0)$ . Since this point is outside of the considered region (see Fig. 5.3) we analyze the boundaries of the region.

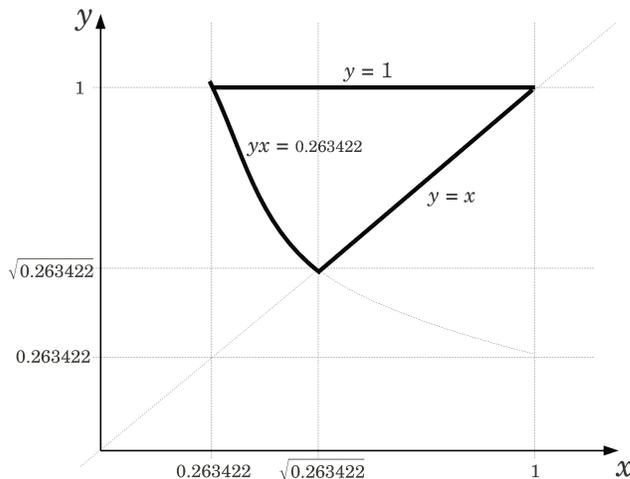


Figure 5.3: The boundaries of the analyzed region:  $y = x$ ,  $y = 1$ ,  $y = \frac{0.263422}{x}$ .

**Case 1:** Consider that  $y = 1$ . Then  $f(x, 1) = \frac{x}{x+1}$  which is strictly increasing at  $(0.263422; 1]$  and so we find the minimum with  $x = 0.263422$ , which leads to a minimum of 0.208499.

**Case 2:** Consider that  $x = y$ . Then  $f(x, x) = \frac{x^2}{x+1}$  which is strictly increasing at  $(\sqrt{0.263422}; 1]$  and so we find the minimum with  $x = \sqrt{0.263422}$ , which leads to a minimum of  $\frac{0.263422}{1+\sqrt{0.263422}}$ .

**Case 3:** Consider that  $xy = 0.263422$ . Then  $f(x, y) = \frac{0.263422}{x+1}$  which is strictly decreasing at  $(0.263422, \sqrt{0.263422}]$  and so we find the minimum with  $x = \sqrt{0.263422}$ , which leads to a minimum  $\frac{0.263422}{1+\sqrt{0.263422}}$ .

So the minimum of the function

$$\frac{w(a_{k+1})h(a_{k+1}) + A(L')}{1 + w(a_{k+1})}$$

occurs when  $w(a_{k+1}) = h(a_{k+1}) = \sqrt{0.263422}$ ,  $A(L') = 0$  and has value  $\frac{0.263422}{1+\sqrt{0.263422}} \approx 0.174077$ .  $\square$



# Chapter 6

## Artigo: *On The Pickup and Delivery with Two Dimensional Loading/Unloading Constraints Problem*<sup>1</sup>

J. L. M. da Silveira<sup>2</sup>    E. C. Xavier<sup>2</sup>

### Abstract

This article addresses the *Pickup and Delivery Problem with Two Dimensional Loading/Unloading Constraints* (PDPLU). In this problem, we are given a weighted complete graph with  $2n + 1$  vertices, a collection  $\mathcal{S} = \{S_1, \dots, S_n\}$  of sets of rectangular items and a bin  $B$  of width  $W$  and height  $H$ . Vertex 0 is a depot and the remaining vertices are grouped into  $n$  pairs  $(p_i, d_i)$ ,  $1 \leq i \leq n$  representing a pickup and a delivery point for each customer. Each set  $S_i$  corresponds to items that have to be transported from  $p_i$  to  $d_i$ . As in the well-known *Pickup and Delivery Problem* (PDP), we have to find the shortest Hamiltonian cycle (route) such that each vertex  $p_i$  is visited before  $d_i$ ,  $1 \leq i \leq n$ , but now we have the additional constraint that there must exist a feasible packing of all items from  $\mathcal{S}$  into  $B$  satisfying Loading/Unloading constraints. This constraint ensures that there exists a free way to insert/remove items into/from  $B$  along the route. We propose two exact algorithms and a GRASP heuristic for the PDPLU. We provide an extensive computational experiment with these algorithms using new instances adapted from existing instances for the PDP.

**Key Words:** Pickup and Delivery, Two Dimensional Knapsack, GRASP.

---

<sup>1</sup>*This research was supported by CNPQ and FAPESP*

<sup>2</sup>Institute of Computing, University of Campinas - UNICAMP, Av. Albert Einstein, 1251, Campinas, Brazil; jmoises@ic.unicamp.br & ecx@ic.unicamp.br

## 6.1 Introduction

The combination of packing and routing problems has received some attention in recent years. This combination models situations where one aims to move goods along a route using vehicles of limited capacity. The *Pickup and Delivery Problem* (PDP) is a classical routing problem [15]. In this problem a graph with a set of pairs of vertices (pickup and delivery) is given and the objective is to generate a route (Hamiltonian cycle) of minimum total cost that visits each pickup client before its delivery pair. We are interested in a generalization of the PDP problem where a set of rectangular items is associated to each pair of pickup and delivery. The objective is to find the PDP solution of minimal cost such that all items can be packed in one vehicle satisfying the traditional packing constraints and loading/unloading constraints. The loading/unloading constraint is the following: While loading/delivering items of a client, there must not exist items of other clients blocking the way in/out of the items of the current client (we always consider loading and unloading movements using the top of the two dimensional bin and using only one vertical movement) (see Fig. 6.1). This problem is called *Pickup and Delivery Problem with Two Dimensional Loading/Unloading Constraints* (PDPLU) [13].

A similar generalization for the *Capacitated Vehicle Routing Problem* was addressed in ([11, 9, 21, 8]), but in this case the packing constraints are easier to tackle, since only the unloading constraint needs to be satisfied.

Besides the routing sub-problem of the PDPLU we also have a particular packing sub-problem: given a list of items with an order in which items are packed and removed from the bin, one has to decide if there is a feasible packing of the items, satisfying the loading/unloading constraints (defined by the order in which items are packed/removed from the bin). The decision version of this problem is a generalization of the classical *Two Dimensional Knapsack Problem* (2KP) [10]. It also generalizes the 2KP version with unloading constraints only [5]. In this problem we are given a set of rectangular items and a bin and we have to find a feasible solution that satisfies the loading/unloading constraint. We call this problem by *Two Dimensional Knapsack Problem with Loading and Unloading Constraints* (2KPLU) (see Fig. 6.1).

To the best of our knowledge, the only paper that addressed the PDPLU problem was the one of Malapert *et. al.* [13], where they proposed a *Constraint Programming* (CP) model for the 2KPLU. No experiments were made with this model in order to assess its quality. Also, the routing problem itself was not addressed.

Besides the scarcity of studies on the PDPLU, the PDP and 2KP problems are well known in the literature.

The PDP version addressed here was studied by Ruland and Rodin in [20]. They proposed an ILP (*Integer Linear Programming*) model and performed several experiments

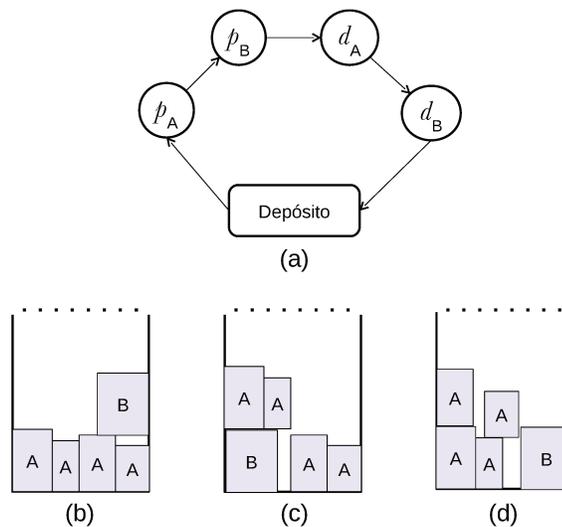


Figure 6.1: Consider the route defined above: Pickup items from A ( $p_a$ ), Pickup items from B ( $p_b$ ), delivery items from A ( $d_a$ ) and delivery items from B ( $d_b$ ) (Part (a)). In (b) we have an infeasible packing, since the removal of two items from A is blocked by an item from B. In (c) we have another infeasible packing, since there are two items from A blocking the insertion of an item from B. In (d) we have a feasible packing.

to assess the practical applicability of the model. Recent advances were made in [7] by Dumitrescu *et al.* which presents polyhedral results based on the work [20]. We use the model proposed of Ruland and Rodin [20] in our exact algorithms for the PDPLU. Besides these exact approaches, Renaud [18] proposed perturbation heuristics to construct feasible routes for the PDP. To the best of our knowledge, these perturbation heuristics are the ones that produce the best results for the PDP in the literature [15]. In our heuristics we use some ideas from [18] for the routing sub-problem.

The 2KP is a popular packing problem. The version with Unloading constraints was addressed in [5] by Silveira *et al.* which presented approximation algorithms for several versions of the problem. Some heuristics were also proposed for this problem: Gendreau *et al.* [9] proposed a heuristic based on the well known *Touching Perimeter* (TP) algorithm. In this heuristic the items are first sorted by inverse order of delivery and then the TP algorithm is used. If no feasible solution is found, then two items are swapped on the input list and the TP algorithm is used again. This procedure is repeated until a feasible solution is found or a number of iterations is reached. Zachariadis *et al.* [21] used similar heuristics but used different strategies to select the point where to pack an item. The strategies used were (TP, Bottom Left (BL), *Min Area*). If none of the heuristics find a feasible point to pack an item, the route being tested is considered infeasible. Some of our algorithms for the 2KPLU are based on the TP and BL ideas. To our knowledge there is

no study on the 2KPLU itself besides the constraint programming model aforementioned [13].

### 6.1.1 Our Results

We propose two exact algorithms for the PDPLU. Both algorithms are based on the Ruland and Rodin's ILP model [20] for the PDP. Both exact algorithms generate routes whose viability, according to the loading/unloading constraints, need to be checked. The two exact algorithms differ on how it is checked the feasibility of the route: one algorithm uses a combinatorial backtracking algorithm to solve the 2KPLU while the other uses the constraint programming model of [13]. Each infeasible route is avoided in the ILP model by inserting a cut, removing this route from the feasible space of the ILP model.

To solve the packing problem (2KPLU) we used exact algorithms and heuristics. One exact algorithm is based on the recursive procedure that uses the *Corner Points* concept [3, 14]. The other exact algorithm is an implementation of the CP model proposed in [13].

We also propose a *Reactive Greedy Randomized Adaptive Search Procedure (GRASP)* heuristic with *Path Relinking* for the PDPLU. This heuristic is based on the insertion of pairs of vertex in the current solution (route). We tested different methods of *Local Search* and *Path Relinking*. We compare this heuristic with an adaptation of the best heuristic of Renaud's [18] for the PDP. In these heuristics the packing problem was solved by simple packing heuristics based on the BL.

The effectiveness of the algorithms and heuristics are assessed through an extensive computational experiment using new instances that were created by adapting instances from the PDP [7].

Our best exact algorithm solved all instances with 5 pairs of vertices, more than half of the instances with 10 pairs of vertices with up to 20 items, and just a few instances with 20 pairs of vertices, including one with 40 items. Finally our *GRASP* heuristic is tested against the exact algorithms. It generates solutions with value close to the optimal while using low CPU time.

### 6.1.2 Paper Organization

This paper is organized as follows: In Section 6.2 we introduce definitions and formalize the description of the PDPLU. The exact algorithms and heuristics for both routing and packing problems are presented in sections 6.3 and 6.4, respectively. In Section 6.5 we describe a data structure used to store a set of feasible/infeasible routes.

The method for generating the instances used on the experiments is described in Section 6.6. In Section 6.7 we summarize our computational experiments and results.

Finally, in Section 6.8, we draw some conclusions about the effectiveness of the proposed heuristics and exact algorithms.

## 6.2 Definitions and Basic Notation

We first define the decision version of the packing problem 2KPLU and then the general problem PDPLU.

An instance for the 2KPLU consists of a bin  $B$  of height  $H$  and width  $W$ , and a list  $L = (a_1, \dots, a_m)$  of rectangular items, each  $a_i$  of width  $w(a_i)$  and height  $h(a_i)$ , pickup point  $p(a_i)$  and delivery point  $d(a_i)$  ( $p(a_i)$  represents the time  $a_i$  must be packed in  $B$  and  $d(a_i)$  represents the time  $a_i$  must be removed from the packing).

Two items  $a_i$  and  $a_j$  **shares** the bin if they appear packed together at some time in the bin. If some item  $a_i$  is such that  $d(a_i) < p(a_j)$  for some  $a_j$ , then these two items do not share the bin.

A packing of  $L$  into  $B$  is feasible if it satisfies the following constraints:

1. If  $a_i$  and  $a_j$  shares the bin then they must not overlap. Formally,

$$\begin{aligned} (p(a_j) < d(a_i) \wedge p(a_i) < d(a_j)) &\Rightarrow \\ (x(a_i) \geq x(a_j) + w(a_j) \vee x(a_j) \geq x(a_i) + w(a_i)) &\vee \\ (y(a_i) \geq y(a_j) + h(a_j) \vee y(a_j) \geq y(a_i) + h(a_i)). & \end{aligned}$$

2. If  $a_i$  is packed and removed while  $a_j$  remains packed, then  $a_j$  must not block  $a_i$  in any of these events (packing or removing). Formally,

$$\begin{aligned} (p(a_j) < p(a_i) \wedge d(a_i) < d(a_j)) &\Rightarrow \\ (x(a_i) \geq x(a_j) + w(a_j) \vee x(a_j) \geq x(a_i) + w(a_i) \vee y(a_i) \geq y(a_j) + h(a_j)). & \end{aligned}$$

3. If  $a_i$  is packed before  $a_j$ , and removed while  $a_j$  is still packed, then  $a_i$  and  $a_j$  must not block each other. Formally,

$$\begin{aligned} (p(a_i) < p(a_j) < d(a_i) < d(a_j)) &\Rightarrow \\ (x(a_i) \geq x(a_j) + w(a_j) \vee x(a_j) \geq x(a_i) + w(a_i)). & \end{aligned}$$

4. Each item  $a_i$  must be packed inside  $B$  from time  $p(a_i)$  to  $d(a_i)$ . Formally,

$$0 \leq x(a_i) \wedge x(a_i) + w(a_i) \leq W \wedge 0 \leq y(a_i) \wedge y(a_i) + h(a_i) \leq H.$$

We consider the decision version of the 2KPLU: Given a list of items  $L$  and a bin  $B$  we need to decide if there is a feasible packing of  $L$  into  $B$  or not.

We consider only the oriented case of the 2KPLU (rotations are forbidden). This problem is strongly NP-Hard since it is a generalization of the classical *Knapsack Problem*.

An instance for the PDPLU consists of a non-oriented complete graph  $G(V, E)$  with costs  $c(e)$  for each edge  $e \in E$ . The size of  $V$  is  $|V| = 2n + 1$ , where the vertices are partitioned into three sets  $V = \{0\} \cup P \cup D$  where vertex 0 represents the depot,  $P = \{p_1, \dots, p_n\}$  represents pickup points and  $D = \{d_1, \dots, d_n\}$  delivery points. We are also given a collection  $\mathcal{S} = \{S_1, \dots, S_n\}$  of sets of rectangular items and a bin  $B$  of width  $W$  and height  $H$ . Each set  $S_i$  is associated with the pair  $(p_i, d_i)$ . The problem is to find a minimum cost Hamiltonian cycle starting and ending at the depot, satisfying that each  $p_i$  is visited before  $d_i$  and that exists a packing for  $\mathcal{S}$  into  $B$  satisfying the 2KPLU constraints when using the ordering of the cycle for the pickup and delivery values of the items.

Through the paper we denote a route  $r$  as an ordered sequence of vertices  $(v_1, \dots, v_k)$  and its size by  $|r| = k$ . We also define the operator  $r - r'$  by the removal of vertices in  $r'$  from  $r$  preserving the order (for instance  $(1, 2, 4, 5) - (1, 4) = (2, 5)$ ). We additionally define  $r + r'$  as the append operation (for instance  $(1, 2, 3) + (4) = (1, 2, 3, 4)$ ). We also say that  $r \subseteq r'$  if the route  $r$  is a sub-sequence of  $r'$  ( $r = r' - r''$  for some  $r''$ ). Finally, an item  $a$  belongs to a route  $r$  ( $a \in r$ ) if and only if  $p_i \in r$  and  $a \in S_i$ .

Let  $\mathcal{P}$  be a packing of a list of items  $L = \{a_1, \dots, a_m\}$ . Denote by  $c(\mathcal{P})$  the induced height of  $\mathcal{P}$ . Formally,

$$c(\mathcal{P}) = \max_{1 \leq i \leq m} \{y(a_i) + h(a_i)\}.$$

## 6.3 Exact Algorithms

In this section we present two exact algorithms for the 2KPLU. The first one is a backtracking algorithm that uses corner points for packing items. The second one is based on the constraint programming model of Malapert *et al.* [13].

### 6.3.1 A Backtracking algorithm for the 2KPLU

In this section we present the backtracking algorithm which we call *Recursive Corner*. The main idea of the algorithm *Recursive Corner* (RC) is to use the concept of *Corner Points* [3, 14] and *Backtracking*. At some point in the search tree we will have a set of items packed in the bin, a set of corner points and the remaining unpacked items. At each level of the search, a new packing is tested combining a different pair of unpacked item and a corner point. If there is such valid pair then the algorithm is recursively called

for the remaining items. If there is no valid pairs remaining to test, then a Backtracking is done.

Aiming to prove the correctness of the algorithm we first prove some properties about packing and *Corner Points*. First, we show that there is always a way to sort the items in any packing in such way that items that appear later on the sequence are “above and at the right” from previous items. After that, we show that the RC algorithm finds a valid packing if such packing exists.

### Items ordering

In this section we prove the existence of a particular order for items in any packing  $\mathcal{P}$ . Notice that for the 2KPLU we can assume a feasible packing in which all items are packed, including items that do not share the bin. The packing must satisfy all constraints of the problem and items that do not share the bin may overlap to each other.

First, we define the relation  $\prec$  over the set of items in a packing.

**Definition 6.3.1** *Given two items  $a_i$  and  $a_j$ , then  $a_i \prec a_j$  iff*

$$( x(a_j) \geq x(a_i) + w(a_i) \quad \wedge \quad y(a_j) + h(a_j) > y(a_i) ) \quad (6.1)$$

$\vee$

$$( y(a_j) \geq y(a_i) + h(a_i) \quad \wedge \quad x(a_j) + w(a_j) > x(a_i) ), \quad (6.2)$$

*moreover, if we assume no overlap between two items,  $a_i \prec a_j$  iff*

$$x(a_j) + w(a_j) > x(a_i) \wedge y(a_j) + h(a_j) > y(a_i). \quad (6.3)$$

Equation (6.1) defines that the left border of  $a_j$  is to the right of the right border of  $a_i$ , while that  $a_j$  top is above the bottom of  $a_i$ . Equation (6.2) defines that the bottom of  $a_j$  is above the top of  $a_i$  and that the right border of  $a_j$  is to the right of  $a_i$ 's left border. Finally, equation (6.3) has the same effect of (6.1) and (6.2) assuming that there is no overlap between  $a_i$  and  $a_j$ . It defines that  $a_j$ 's right border is at right of  $a_i$ 's left border and that  $a_j$ 's top is above the bottom of  $a_i$ . Equation (6.3) can be also stated as follows: the top-right point of  $a_j$  dominates the bottom-left point of  $a_i$  (except the cases in which both points have equal coordinates).

We use both definitions through the paper. Figure 6.2 presents the graphic representation of the relation  $\prec$ .

Define  $\not\prec$  as the inverse of  $\prec$  using the negative of the propositions (6.1), (6.2) and (6.3) into (6.4), (6.5) e (6.6), respectively. So we have that  $a_i \not\prec a_j$  iff:

$$( (x(a_j) < x(a_i) + w(a_i)) \quad \vee \quad (y(a_j) + h(a_j) \leq y(a_i)) ) \quad (6.4)$$

$\wedge$

$$( (y(a_j) < y(a_i) + h(a_i)) \quad \vee \quad (x(a_j) + w(a_j) \leq x(a_i)) ), \quad (6.5)$$

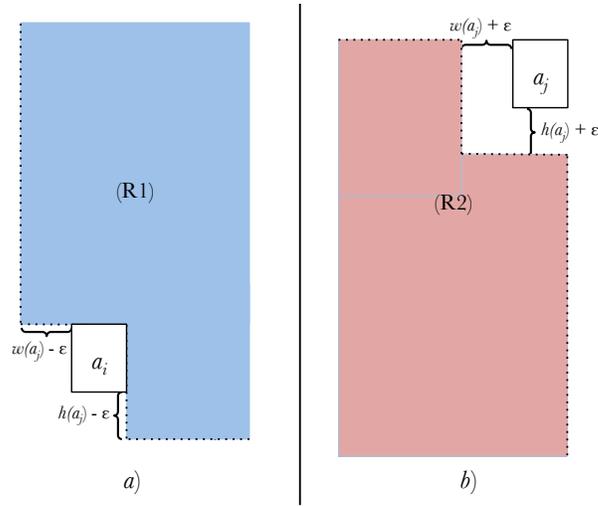


Figure 6.2: Suppose  $\epsilon > 0$  sufficiently small and  $a_i \prec a_j$ , then the bottom left corner of  $a_j$  must be in (R1) (part a). Also  $a_i$ 's bottom-left corner must be in (R2) (part b).

moreover, if we assume no overlap between two items,  $a_i \not\prec a_j$  iff

$$x(a_i) \geq x(a_j) + w(a_j) \wedge y(a_i) \geq y(a_j) + h(a_j). \quad (6.6)$$

**Lemma 6.3.1** *Let  $\mathcal{P}$  be a valid packing of a list of items (without loading/unloading constraints). Then  $\prec$  defines a partial order  $\pi = \{a_1, \dots, a_n\}$  over the items of  $\mathcal{P}$ .*

**Proof.** See the Appendix 6.10. □

**Corollary 6.3.2** *Let  $\prec_{LU}$  be a binary relation over the items of a packing  $\mathcal{P}$  of items  $L = \{a_1, \dots, a_m\}$  for the 2KPLU, defined as:  $a_i \prec_{LU} a_j$  iff*

$$(a_i \prec a_j) \wedge (p(a_j) < d(a_i) \wedge p(a_i) < d(a_j))$$

*Then  $\prec_{LU}$  induces a partial ordering of  $\mathcal{P}$ .*

### Corner Points

In this section we define the concept of *Corner Points*. First, define as  $r(a_i)$ , the region of  $\mathbb{R}^2$  covered by  $a_i$ , formally,  $r(a_i) = \{(x, y) : x(a_i) \leq x < x(a_i) + w(a_i) \text{ and } y(a_i) \leq y < y(a_i) + h(a_i)\}$ . Besides that, we define as  $r(B)$  the region of  $\mathbb{R}^2$  occupied by the bin  $B$ .

In definition 6.3.2 we describe the region of the plane called envelope of  $\mathcal{P}$ . It can be defined as a subset of points  $p$  of  $r(B)$  such that there exists an item  $a_i$  that contains a point that dominates  $p$ .

**Definition 6.3.2** Let  $\mathcal{P}$  be a packing of a list  $L$  of items. Thus, the envelope of  $\mathcal{P}$  ( $e(\mathcal{P})$ ) can be stated as follows:

$$e(\mathcal{P}) = \{(x, y) : \exists(x', y') \in r(a_i) \text{ for some } a_i \in \mathcal{P} \text{ such that } x' \geq x \text{ and } y' \geq y\}$$

We define as  $e^-(\mathcal{P}) = r(B) \setminus e(\mathcal{P})$ , the region of  $r(B)$  minus the envelop of  $\mathcal{P}$ . Define the set of horizontal line segments that divide  $e^-(\mathcal{P})$  and  $e(\mathcal{P})$  as the contour lines of  $\mathcal{P}$  (See Fig. 6.3).

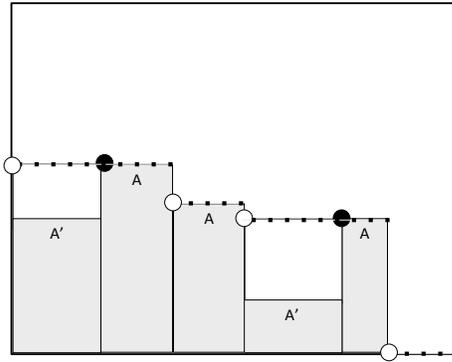


Figure 6.3: In this figure we represent the contour lines of the packed items ( $l(\mathcal{P})$ ) with dashed lines. The region below  $l(\mathcal{P})$  is the  $e(\mathcal{P})$  and above  $e^-(\mathcal{P})$ . The white circles represent the set  $corner(\mathcal{P})$ . Suppose that an item  $a$  is being packed and that it will be removed after the items  $A'$ , then we have the set  $orderContour(a, \mathcal{P})$  represented by black circles.

**Definition 6.3.3** Let  $\mathcal{P}$  be a packing of a list  $L$  and  $e(\mathcal{P})$  its envelope, denote as the contour lines of  $\mathcal{P}$  ( $l(\mathcal{P})$ ) the following set of points:

$$l(\mathcal{P}) = \{(x, y) : (x, y) \in e^-(\mathcal{P}) \text{ such that } \nexists(x', y') \in e^-(\mathcal{P}) \text{ where } y' < y\}.$$

Now we define formally the concept of *Corner Points* as the leftmost point in each segment on  $l(\mathcal{P})$ .

**Definition 6.3.4** Let  $\mathcal{P}$  a packing of a list  $L$ , we denote as the *Corner Points* of  $\mathcal{P}$  ( $corner(\mathcal{P})$ ) the following set of points:

$$corner(\mathcal{P}) = \{(x, y) : (x, y) \in l(\mathcal{P}) \text{ and } \nexists(x', y) \in l(\mathcal{P}) \text{ where } x' < x\}.$$

In order to consider the loading/unloading constraints, we consider some additional points besides  $corner(\mathcal{P})$ . For each item  $a$  still not packed in  $\mathcal{P}$ , we define as  $corner^*(a, \mathcal{P})$  the set points that considers the 2KPLU constraints. This set contains the points in  $corner(\mathcal{P})$  and the points in  $l(\mathcal{P})$  that are projections of the right borders of any item in  $\mathcal{P}$  that will be removed before  $a$  ( $orderContour(a, \mathcal{P})$ ). Formally

$$orderContour(a, \mathcal{P}) = \{(x(b) + w(b), y) : \\ \exists b \in \mathcal{P} \text{ such that } (x(b) + w(b), y) \in l(\mathcal{P}) \text{ where } d(b) < d(a)\}$$

Finally, the set of points where  $a$  can be packed given  $\mathcal{P}$  (See Fig. 6.3) is defined as:

$$corner^*(a, \mathcal{P}) = \{(x, y) : (x, y) \in corner(\mathcal{P}) \cup orderContour(a, \mathcal{P}) \\ \text{and } \nexists b \in \mathcal{P} \text{ such that packing } a \text{ in } (x, y) \text{ blocks } b\}$$

### The RC Algorithm

The RC algorithm is basically a brute-force algorithm (See Alg. 13), which tests the packing of each unpacked item in each corner point.

Now we prove the correctness of the algorithm RC (Theorem 6.3.5) using the Lemma 6.3.1 and Observations 6.3.3 and 6.3.4.

**Observation 6.3.3** *If  $\mathcal{P}$  is a feasible packing for the 2KPLU, then there is a packing  $\mathcal{P}'$ , with  $c(\mathcal{P}') \leq c(\mathcal{P})$ , where the items can not be moved to the left or down without breaking the constraints of the problem.*

**Observation 6.3.4** *Let  $I = (L, B)$  be an instance of the 2KPLU. Before the RC algorithm answers “No”, it must have generated all permutations of  $L$ , and for each permutation its tested the packing of each item on all the contour points defined.*

**Theorem 6.3.5** *Let  $I = (L, B)$  be an instance of the 2KPLU. If there is a feasible packing  $\mathcal{P}$  for  $I$ , then the algorithm RC will find it.*

**Proof.** First, we define  $\mathcal{P}'$  as in Observation 6.3.3, by moving each item down and to the left while the packing remains feasible. Let  $\sigma = (a_1, \dots, a_n)$  be the permutation of  $L$  imposed by  $\prec_{LU}$  in  $\mathcal{P}'$ . By 6.3.4, the RC algorithm, will eventually pack the items of  $L$  in the order defined by  $\sigma$ .

Thus, we must prove that if the RC algorithm pack the items in this order it will find the packing  $\mathcal{P}'$ . We do it by induction on the size  $k$  of a prefix of  $\sigma$  (the prefix is denoted by  $\sigma(k)$ ).

---

**Algorithm 13** Recursive Corner (RC)
 

---

```

1: Input: A list  $L$  of items, a bin  $B$ , and an empty packing  $\mathcal{P}$ .
2: Output: “Yes” or “No” depending on the existence of a feasible packing of  $L$  into  $B$ 
   satisfying the 2KPLU constraints.
3: Begin
4: if  $L = \emptyset$  then
5:   Return Yes.
6: for  $a \in L$  do
7:   // List of items in  $\mathcal{P}$  that share some part of the route with  $a$ .
8:    $\mathcal{L} \leftarrow \{a_i \in \mathcal{P} : d(a_i) > p(a) \text{ and } d(a) > p(a_i)\}$ .
9:   if  $\mathcal{L} \neq \emptyset$  then
10:     $P \leftarrow \text{corner}^*(a, \mathcal{L})$ .
11:   else
12:     $P \leftarrow \{(0, 0)\}$ .
13:   for  $p \in P$  do
14:     Pack  $a$  in  $p$ .
15:     if  $\text{RC}(L \setminus a, \mathcal{P} \cup a) = \text{Yes}$  then
16:       Return Yes.
17:     Remove  $a$  from  $p$ .
18: Return No.
19: end.

```

---

**Base:** Let  $k = 1$ , and notice that the item  $a_1$  is packed on  $(0, 0)$ . Since there is no item  $i$  in  $\sigma$  such that  $a_i \prec_{LU} a_1$ , we can assume that  $a_1$  is in  $(0, 0)$  in  $\mathcal{P}'$  (See observation 6.3.3 and Fig. 6.2 part b);

**Induction Hypothesis:** The RC algorithm packs a prefix of  $\sigma$  of size  $k$  in the same position as in  $\mathcal{P}'$ .

**Step:** Suppose the RC algorithm is packing the item  $a_{k+1}$ ,  $1 < k \leq n$ . By induction, we have that  $a_i$ ,  $1 \leq i \leq k$  were packed in their corresponding positions according to  $\mathcal{P}'$ . By  $\prec_{LU}$  we have that all items  $a_j \in \sigma$ , such that  $a_j \prec_{LU} a_{k+1}$  were packed.

If we show that the position of  $a_{k+1}$  in  $\mathcal{P}'$  is in  $corner^*(a_{k+1}, \bigcup_{i=1}^k a_k)$  then we prove the theorem, since the algorithm will eventually pack the item on this position.

Notice that if  $a_{k+1}$  is positioned according to  $\mathcal{P}'$ , then it cannot have an intersection with  $e(\bigcup_{i=1}^k a_k)$  since we would have a contradiction. This occurs because  $a_{k+1} \prec_{LU} a_j$ , for all  $1 \leq j \leq k$ . Besides that, in  $\mathcal{P}'$ ,  $a_{k+1}$  was positioned in the most bottom-left feasible position. So we know that  $(x(a_{k+1}), y(a_{k+1}))$  is in  $e^-(\bigcup_{i=1}^k a_k)$  and  $\nexists(x(a_{k+1}), y)$ ,  $y < y(a_{k+1})$  in  $e^-(\bigcup_{i=1}^k a_k)$ .

We have three cases according to what happened when trying to move  $a_{k+1}$  to the left. In the first case, suppose that  $a_{k+1}$  cannot be moved to the left in  $\mathcal{P}'$  because it touches the right border of another item  $a_j$  ( $x(a_j) + w(a_j) = x(a_{k+1})$ ). This means that  $\nexists(x, y(a_{k+1}))$ ,  $x < x(a_{k+1})$  in  $e^-(\bigcup_{i=1}^k a_k)$ . Therefore,  $(x(a_{k+1}), y(a_{k+1})) \in corner(P)$ .

In the second case, suppose that  $a_{k+1}$  cannot be moved to the left because it touches the left border of the bin. Since  $a_{k+1}$  is in its lowest position possible, this point also belongs to  $corner(P)$ .

In the third case, suppose that  $a_{k+1}$  cannot be pushed to the left in  $\mathcal{P}'$  because it would block an item  $a_j$  already packed. In this case  $x(a_{k+1}) = x(a_j) + w(a_j)$  for some  $1 \leq j \leq k$ . By induction hypothesis  $a_j$  is packed in the same position as in  $\mathcal{P}'$ . Besides that,  $(x(a_j) + w(a_j), y(a_{k+1})) \in orderContour(a_{k+1}, \bigcup_{i=1}^k a_k)$ . Thus, this position will be tested by the algorithm RC.  $\square$

Notice that the order imposed by  $\prec_{LU}$  has nothing to do with the order in which the items are loaded by RC. For example, consider the instance defined by the route  $(0, 1, 2, 3, 5, 4, 6, 0)$  with 3 items where  $p(A) = 1$ ,  $d(A) = 6$ ,  $p(B) = 2$ ,  $d(B) = 4$ ,  $p(C) = 3$  and  $d(C) = 5$ . The order of loading on the bin would be  $(A, B, C)$ . In this case, the RC algorithm would not find a feasible solution using this order since it would pack the items  $A$  and  $B$  with the configuration presented in part (1) of Figure 6.4. On the other hand, if we consider the pack in part (2) of the same figure, we would find that the order imposed by the  $\prec_{LU}$  could be  $(B, A, C)$ . In this case, the algorithm would find the packing in part (2) of the same figure.

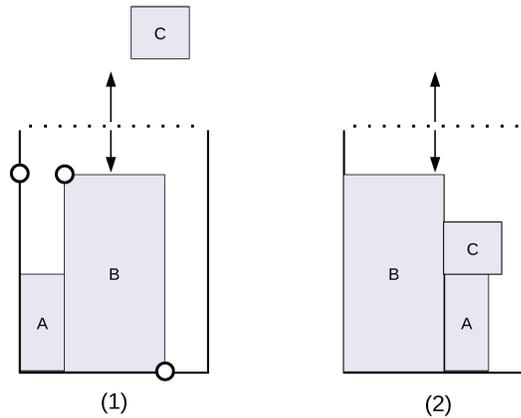


Figure 6.4: Example of packings using different orderings of items.

### Improvements on the RC algorithm

Notice that, by Observation 6.3.4, the algorithm RC may try all  $n!$  possible permutations of the input list. Thus, it becomes impractical to solve the problem even for instances of small size. In this Section we propose some techniques used to reduce the number of packings tested by the algorithm:

1. **Avoid floating items:** By Observation 6.3.3, if the algorithm RC is considering to pack an item in a point without any support (the bottom of the item does not touch the base of the knapsack or the top of another item), then this packing can be discarded. This may be done since there is a packing with no higher induced height that does not use this point (See Fig. 6.5).
2. **Lower bound for the height of the packing:** In [4], Silveira *et al.* presented a lower bound on the height of packings that must satisfy the unloading constraint. It provides a minimum height  $h$  required to pack all items satisfying the unloading constraint. If  $h$  is greater than the height of the bin, then there is no feasible packing for this set of items.
3. **Width projection of the items:** Consider a set of items already packed  $S$  and an unpacked item  $a_j$  such that  $p(a_i) < p(a_j) < d(a_i) < d(a_j)$  for each  $a_i \in S$ . By definition, the projections of the items in  $S$  on the base of the bin and the projection of the item  $a_j$  must not overlap. Thus for each item  $a_j$  to be packed, we compute the set  $S$  and check if there is enough space on the projection of the items of  $S$  to pack  $a_j$ . If there is no space for  $a_j$  then this packing is infeasible. (see Fig. 6.6).

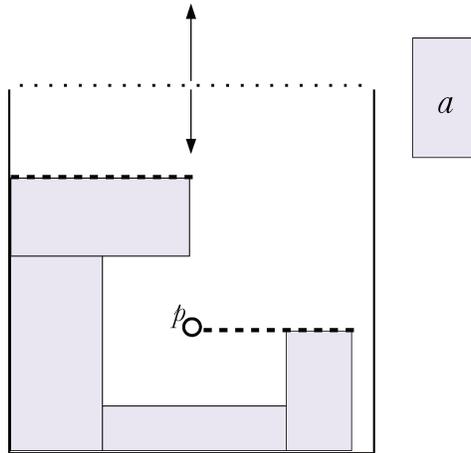


Figure 6.5: In this case, the item  $a$  do not need to be packed in  $p$ . There is another order of the items that can produce a packing where item  $a$  is packed on the point below  $p$ .

### 6.3.2 A CP Model for the 2KPLU

In this Section we present a *Constraint Programming* model similar to the model studied by Malapert *et al.* [13] (we call it CP). The formulation is based on the definition we presented in Section 6.2. Given a list  $L = \{a_1, \dots, a_m\}$  of items, the model is formed by  $2m$  integer variables ( $n$  pairs  $(x_i, y_i)$  representing the bottom left corner coordinates where the  $i$ -th item is packed).

The CP model is defined by the following constraints over the variables  $(x_i, y_i)$ ,  $1 \leq i \leq m$ :

$$Find : \quad x_i \in [0, W - w(a_i)] \quad \text{and} \quad y_i \in [0, H - h(a_i)] \quad \forall a_i \in L \quad (6.7)$$

subject to

$$\begin{aligned}
 p(a_i) = p(a_j) \Rightarrow & x(a_i) + w(a_i) \leq x(a_j) \vee \\
 & x(a_j) + w(a_j) \leq x(a_i) \vee \\
 & y(a_i) + h(a_i) \leq y(a_j) \vee \\
 & y(a_j) + h(a_j) \leq y(a_i) \quad \forall a_i, a_j \in L \quad (6.8)
 \end{aligned}$$

$$\begin{aligned}
 p(a_i) < p(a_j) < d(a_j) < d(a_i) \Rightarrow & x(a_i) + w(a_i) \leq x(a_j) \vee \\
 & x(a_j) + w(a_j) \leq x(a_i) \vee \\
 & y(a_i) + h(a_i) \leq y(a_j) \quad \forall a_i, a_j \in L \quad (6.9)
 \end{aligned}$$

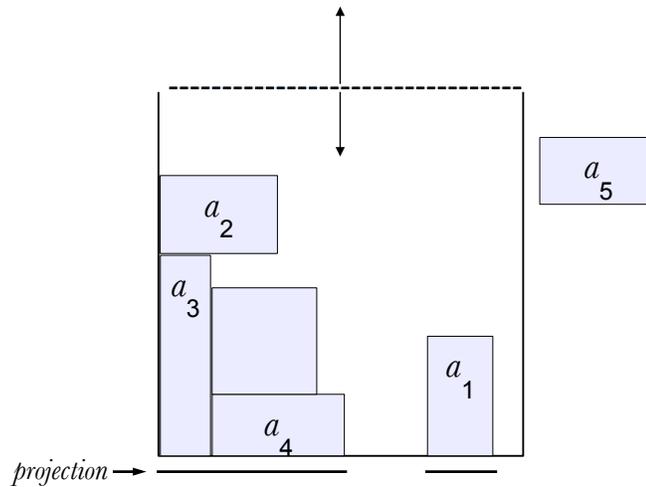


Figure 6.6: Suppose  $a_5$  is not packed yet, and that items  $a_i$   $1 \leq i \leq 4$  belongs to the set  $S$ , i.e,  $a_5$  is going to be packed after them, and all of them are going to be removed before  $a_5$ . There is no space to pack  $a_5$ , thus this configuration is invalid.

$$p(a_i) < p(a_j) < d(a_i) < d(a_j) \Rightarrow x(a_i) + w(a_i) \leq x(a_j) \vee x(a_j) + w(a_j) \leq x(a_i) \quad \forall a_i, a_j \in L \quad (6.10)$$

The domains of  $x_i$  and  $y_i$  are defined such that the items are fully contained into the bin (Constraints 6.7). Constraints (6.8) impose that items that belong to the same client (same pickup and delivery vertex) will not overlap. The unloading constraint is imposed by the constraints (6.9) which guarantees that if  $a_j$  is packed and delivered while  $a_i$  remains packed then either  $a_j$  is packed above  $a_i$  or their projections on the base of the bin do not overlap. Finally, the Constraints in (6.10) ensure that if  $a_i$  is packed before  $a_j$ , and it is delivered before  $a_j$  then the projections of  $a_i$  and  $a_j$  on the base of the bin do not overlap.

### 6.3.3 An ILP based algorithm for the PDPLU

In [7], Dumitrescu *et al.* analyzed and improved the ILP model proposed by Ruland and Rodin [20] for the PDP problem. In this Section we use the same model to solve the PDPLU problem. The model generates routes satisfying the pickup and delivery constraints but there is no guarantee that the items can be packed in the order determined by the route. So we added a new separation procedure using exact algorithms for the 2KPLU to check the feasibility of the route. If the route is infeasible, then we insert a cut in the model to avoid the generation of this route.

Let  $G = (V, E)$  be the graph that is part of the input of the PDPLU problem. We add a dummy delivery vertex ( $d_0$ ) to be the final point of the route (i.e. the delivery point of the initial depot denoted by  $p_0$ ). We denote by  $\delta(S) = \{(i, j) \in E : i \in S, j \notin S \text{ or } i \notin S, j \in S\}$  to be the edges of the cut given by some  $S \subseteq V$ . We also denote by  $\delta(v) = \delta(\{v\}), \forall v \in V$ .

The formulation is presented below. It contains  $|E|$  binary variables: one variable  $x_{ij}$  for each edge  $e_{ij}$ . We also define  $x(E') = \sum_{(i,j) \in E'} x_{ij}$ , for  $E' \subseteq E$ .

$$\min \sum_{e_{ij} \in E} c(e_{ij})x_{ij} \quad (6.11)$$

subject to

$$x_{p_0, d_0} = 1 \quad (6.12)$$

$$x(\delta(v)) = 2 \quad \forall v \in V \quad (6.13)$$

$$x(\delta(S)) \geq 2 \quad \forall S \subseteq V, 2 \leq |S| \leq |V|/2 \quad (6.14)$$

$$x(\delta(S)) \geq 4 \quad \forall S \in \mathcal{U} \quad (6.15)$$

$$x_{ij} \in \{0, 1\} \quad \forall e_{ij} \in E, \quad (6.16)$$

where  $\mathcal{U}$  is the set of subsets of  $S \subseteq V$  that satisfies  $3 \leq |S| \leq |V| - 2$  with  $p_0 \in S$ ,  $d_0 \notin S$  and there exists  $i \in \{1, \dots, n\}$  such that  $d_i \in S$  and  $p_i \notin S$ . Constraints (6.12), (6.13), (6.14) and (6.16), define the classic *Traveling Salesman Problem* (TSP). Finally, the constraint (6.15), guarantees that for each pair  $(p_i, d_i)$ ,  $p_i$  is visited before  $d_i$ .

Due to the exponential number of constraints ((6.14) and (6.15)) Ruland [20] and Dumitrescu *et al.* [7] proposed a series of separation procedures for a *Branch-and-Cut* algorithm. In our implementation, we used two of these procedures in order to guarantee the feasibility of the solution and to avoid the use of all the constraints in (6.14) and (6.15).

The first procedure is a separation method for constraints (6.14), and makes use of an algorithm for the max flow problem. Let  $x^*$  be an optimal fractional solution for the model. It is possible to verify the constraints (6.14) calculating the max flow from  $p_0$  to  $d_0$  (considering  $x^*$  as the edges capacities). If the total flow is smaller than 2, with corresponding minimum cut  $S$ , then we add to the model the constraint  $x(S) \geq 2$ .

The second procedure is a separation method for the constraints (6.15). In this case, the procedure aims to find a subset of  $\mathcal{U}$  that violates the constraint (6.15). This problem can also be modeled as a max flow problem. One just need to find the max flow from  $\{p_0, d_i\}$  to  $\{p_i, d_0\}$ ,  $i \in \{1, \dots, n\}$  on the same graph defined for the first procedure. If the total flow is smaller than 4, with corresponding minimum cut  $S$ , then the constraint  $x(S) \geq 4$  is added to the model.

If none of these procedures add a new cut into the model and all variables have integer values, then we test if the route induced by the solution is feasible from the 2KPLU stand point.

To solve the PDPLU we solve the model presented using the separation procedures described above. We call this algorithm  $ILP_{PDPLU}$ . Notice that  $ILP_{PDPLU}$  is an exact algorithm as long as  $\mathcal{A}_{2KPLU}$  is also an exact algorithm for the 2KPLU problem.

## 6.4 Heuristic Algorithms

In this section we present heuristics for the problems 2KPLU and PDPLU. In Section 6.4.1 we show three heuristics for the 2KPLU that are based on the Bottom Left strategy. Then in Section 6.4.2 we present an algorithm based on the work of Renaud *et al.* [18] for the PDPLU. Finally, in Section 6.4.3 we describe a *GRASP* heuristic for the PDPLU.

### 6.4.1 Bottom Left heuristics for the 2KPLU

In this section we present three similar heuristics for the 2KPLU. These algorithms are used to check if there is a feasible packing for the items given that they need to be packed and removed according to some route.

The first heuristic ( $BL_{LU}$ ) is an adaptation of the classic Bottom Left (BL) algorithm [1, 2]. The heuristic tries to pack the items in order (sorted by width) in the lowest available position, left aligned. However the packing of an item must satisfy the constraints of the 2KPLU. Thus items are packed in the lowest available feasible position left aligned. If the packing becomes infeasible, we swap two randomly chosen items on the input list and use the algorithm again. This process is repeated until a solution is found or the algorithm reaches a certain number of iterations.

The  $BL_{LU}$  is presented in Algorithm 14 and an example is given in Figure 6.7.

The second algorithm is similar to the  $BL_{LU}$  but it uses a Touching Perimeter [12] strategy instead of a BL. The touching perimeter of a packed item, corresponds to the size of its edges that are touching other items or the borders of the bin. The algorithm starts sorting the items by area (largest first). Then the algorithm proceeds by packing the items in order and left aligned in the position that maximizes the touching perimeter. As for the  $BL_{LU}$  algorithm, this one also swaps items from the list and repeats the process when no feasible solution is found.

The idea of the TP algorithm is to avoid the formation of small holes on the packing. In Figure 6.8 we present an example of packing obtained with the algorithm  $TP_{LU}$ .

The third heuristic has the same idea of the first two. However, instead of using the lowest point or the maximum perimeter, we use a maximum overlap strategy. By

---

**Algorithm 14**  $BL_{LU}$ 

---

```

1: Input: A list of items  $L$  and a maximum number of iterations  $maxIter$ .
2: Output: Return yes or no, which depends if a feasible packing is found or not.
3: Begin
4:  $iter \leftarrow 0$ .
5:  $L' = L$ 
6: while  $iter < maxIter$  do
7:   for  $a \in L'$  do
8:     Let  $p$  be the lowest feasible position to pack  $a$ , left aligned.
9:     if There is no such  $p$  then
10:      Stop the for loop.
11:    else
12:      Pack  $a$  in  $p$ .
13:       $L' = L' - a$ .
14:    if  $L' == \emptyset$  then
15:      Return YES.
16:    // Swap Items.
17:    Let  $a_i$  and  $a_j$  be two randomly chosen items in  $L$ .
18:    Swap  $a_i$  and  $a_j$ .
19:     $iter = iter + 1$ .
20: Return No.
21: end.

```

---

definition, items that do not share the bin can overlap in a solution for the 2KPLU. For example suppose that we have two pairs  $(p_1, d_1)$  and  $(p_2, d_2)$ , and one item for each pair. Consider the route  $(p_1, d_1, p_2, d_2)$ , it is easy to see that both items can be packed at position  $(0, 0)$ . Given a list of items they are packed one after the other, in the position that maximizes the overlap with already packed items that do not share the bin (ties broken by maximum *Touching Perimeter*), left aligned, satisfying the 2KPLU constraints. We call this algorithm  $OVERLAP_{LU}$ . An example is presented in Figure 6.9.

### 6.4.2 A simple heuristic for the PDPLU

In this Section we describe a simple heuristic called 4-Opt\*\*\* for the PDPLU based on a heuristic presented by Renaud et. al. [18] for the PDP.

In [18], Renaud et. al. compared seven heuristics that are based on the following strategy: use a constructive algorithm together with an improvement algorithm called 4-Opt\*\* (an adaptation from the 4-Opt\* proposed in [16, 17]). The 4-Opt\*\* works by removing and reinserting sets of 4 edges from the solution. After that, there are phases of perturbation and improvement which are repeated until a stopping criteria is met.

Among the seven heuristics compared, the best one performs three basic steps:

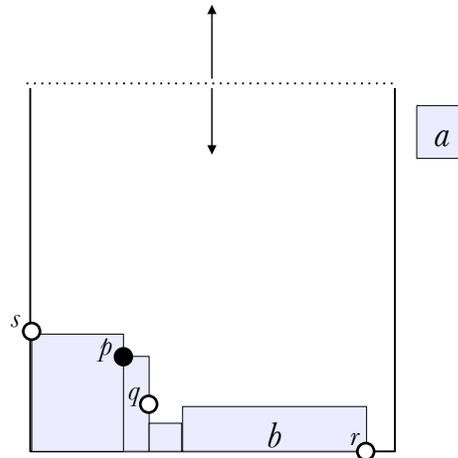


Figure 6.7: In this figure we have an example for the  $BL_{LU}$  where  $b$  is the only item that will be removed before  $a$ . The item  $a$  can not be packed in the point  $r$  since  $a$  would not fit in the bin. Point  $q$  is also infeasible for  $a$  since the packing would not satisfy the unloading constraint. Points  $p$  and  $s$  are feasible and  $p$  is chosen since it is the lowest one.

- *Initialization:* Construct a feasible solution (route) for the PDP inserting pairs of vertices in a greedy way. Then apply the 4-Opt\*\* in order to optimize the solution.
- *Perturbation:* Generate a route  $S1$  by removing and reinserting pairs of vertices from the current solution in feasible random positions. Then generate  $S2$  in a similar way but reinserting the removed vertices in a greedy way. Finally, consider a prefix from  $S1$ . Construct a complete route by using this prefix and completing it with the remaining vertices in the order they appear in  $S2$ . Do the same thing by using a prefix of  $S2$  and completing the route with vertices in the order they appear in  $S1$ .
- *Post-optimization:* Apply the 4-Opt\*\* to the solutions found in the previous step (only the combinations of  $S1$  and  $S2$ ). If a stop criteria is met then stop, otherwise go to the *Perturbation* phase.

Our proposed heuristic is a simple modification of this one in order to construct routes that have a feasible packing: when inserting pairs of vertices in the route, a 2KPLU algorithm is used to tests if the insertion is valid.

### 6.4.3 A GRASP heuristic for the PDPLU

In this section we describe a *Reactive GRASP* with *Path Relinking* for the PDPLU.

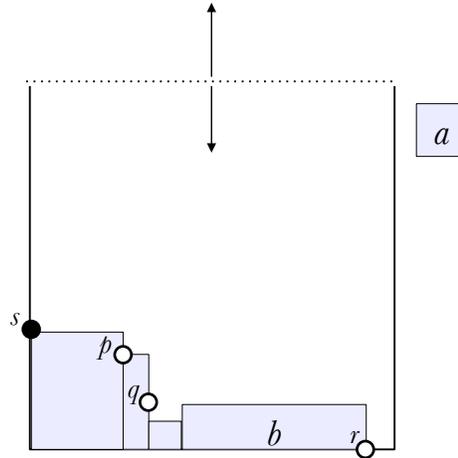


Figure 6.8: In this figure we have the same scenario from the Fig. 6.7. In this case, the point  $s$  is chosen since it maximizes the touching perimeter of  $a$ .

First consider the *Constructive* phase. We always keep a partial feasible route containing pairs of pickup and delivery points. This partial solution starts empty. At each iteration, a new pair of vertices is selected from a *Restricted Candidates List (RCL)* and inserted on the partial solution. The *RCL* is updated on each iteration by choosing the  $x\%$  best pairs of pickup and delivery points. The best pairs are the ones whose insertion in the partial solution causes the minimum increase of its cost. In Alg. 15 we show the *Constructive* phase.

The heuristic is said to be *Reactive* since the size of the *Restricted Candidates List* ( $x$ ) is self-adjustable. To adjust the value of  $x$ , we used the best approach described in [19]. We have a set of possible values for  $x$  called *Reactive Set*,  $\mathcal{X} = \{x_1, x_2, \dots, x_l\}$ . Each time the constructive phase is ran, we select the value for  $x$  randomly from this set. The probability to select  $x_i$  is  $pr(x_i), 1 \leq i \leq l$  which is initially set as  $1/l$ . Every time a certain number of iterations ( $It\_update$ ) are completed, the probabilities are updated with a given *Aggressiveness* (see [19] for details). The idea is to increase the probabilities of selecting values  $x_i$  that result in good initial solutions. These parameters, such as  $l$  and by how much update the probabilities, were set using a preliminary set of experiments.

We tested two methods for the *Local Search*. The first one is the 4-Opt\*\* [18] but considering the packing constraints (Section 6.4.2). In the second local search, for each pair of vertices  $(p_i, d_i)$ , they are removed and re-inserted again in the lowest cost position (see Alg. 16). The algorithm performs this operation until no improvement is obtained.

We also propose three routines for *Path-Relinking (PR)*. Here we also follow the best strategy recommended in [19]. We keep a small pool of size  $Pool\_size$ , containing *elite*

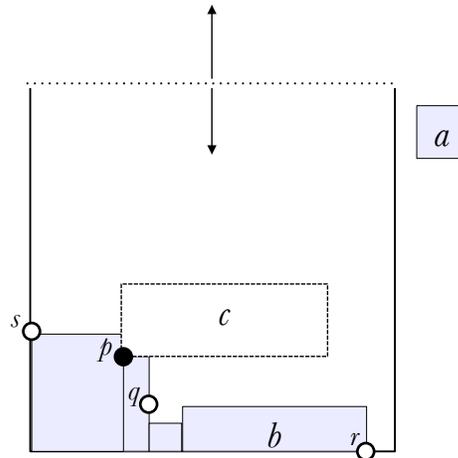


Figure 6.9: In this figure we have a similar scenario from the Figures 6.7 and 6.8. Suppose that items  $a$  and  $c$  do not share the bin. In this case, the point  $p$  is chosen due to the overlap between  $a$  and  $c$ .

solutions: the best *Pool\_size* solutions generated so far by the GRASP heuristic. Every time a new “good solution” is generated by the *Local Search*, we perform a path relinking between this solution and one randomly chosen from the elite pool. By “good solution” we mean a solution whose cost is at most  $d\%$  worst than the best solution already found. We set the value for  $d$  during our experiments ( $d$  can be seen as a *filter* for the PR usage). The path relinking is done by transforming the elite solution into the current solution, performing small modifications at each step, until the solutions becomes the same. We can also consider the other direction, i.e, start from the current solution, and perform modifications on it until it becomes the elite solution. During each modification step from one solution to the other, we check if this modified solution has lower cost than the worst solution on the elite set. If a better solution is created during some of these steps the elite set is updated.

The first path-relinking routine (*Path Relinking Prefix*) perform modifications based on prefixes of routes (an adaptation from [18]). Let  $\text{pref}(s, i)$  be the prefix of route  $s$  with size  $i$ , i.e. the first  $i$  vertices visited by  $s$  after the depot. The path of modification steps from solution  $s$  to  $s'$  consists of  $|s| - 2$  new solutions  $s_1, s_2, \dots, s_{|s|-2}$  where  $s_i = \text{pref}(s', i) + (s - \text{pref}(s', i))$ , i.e., the route  $s_i$  is formed by the prefix of  $s'$  of size  $i$ , followed by the remaining vertices in the order they appear in  $s$ . For instance, Let  $s = (1, 2, 3, 7, 6, 4, 8, 5)$  and  $s' = (1, 3, 7, 2, 4, 5, 8, 6)$ , then  $s_3 = (1, 3, 7, 2, 6, 4, 8, 5)$ . This procedure is showed in Alg. 17.

The second path-relinking (*Path Relinking Remove*) is based on the removal and in-

---

**Algorithm 15** *Constructive phase*

---

```

1: begin
2: Input: A list  $L = (p_i, d_i)$ ,  $1 \leq i \leq n$ , of vertices and the size of the RCL denoted by  $x$ .
3: Output: A solution for the PDPLU.
4:  $Solution \leftarrow \emptyset$ 
5: while  $L \neq \emptyset$  do
6:   Build the RCL  $\subseteq L$ , with the  $x\%$  best pairs of vertices from  $L$  (lowest insertion cost).
7:   Randomly select a pair  $(p, d) \in RCL$ .
8:   Insert  $(p, d)$  on Solution, on the positions that minimize the insertion cost.
9:    $L \leftarrow L \setminus \{(p, d)\}$ .
10: return Solution.
11: end

```

---



---

**Algorithm 16** *Local Search Pairs*

---

```

1: begin
2: input: An initial solution  $s$  for the PDPLU.
3:  $changed \leftarrow true$ 
4: while  $changed$  do
5:    $changed \leftarrow false$ 
6:   for each pair  $(p, d)$  on  $s$  do
7:     if there is a position  $pos_p, pos_d$  for  $p$  and  $d$  that reduces the cost of  $s$  then
8:        $changed \leftarrow true$ 
9:       Remove  $p$  and  $d$  from  $s$ , and insert them back on positions  $pos_p, pos_d$ .
10: return  $s$ .
11: end

```

---

sersion of pairs of vertices in a route  $s$  until it becomes equal to  $s'$ . Let  $s(i)$  be the  $i$ -th vertex on the route  $s$  after the depot. The method starts by finding the first position  $i$  in  $s$  such that  $s(i) \neq s'(i)$  and then remove the pair that contains  $s(i)$  and reinsert the vertices into  $s$  on the same position as they appear in  $s'$ . For example, let  $(v, v + 4)$   $1 \leq v \leq 4$  be the pairs of pickup and delivery in an instance. Let  $s = (1, 2, 3, 7, 6, 4, 8, 5)$  and  $s' = (1, 3, 7, 2, 4, 5, 8, 6)$  be the routes considered by the second routine, then the new routes tested are  $(1, 3, 7, 2, 6, 4, 8, 5)$ ,  $(1, 3, 7, 2, 4, 6, 8, 5)$ ,  $(1, 3, 7, 2, 4, 5, 6, 8)$ , in order. This routine is described in Algorithm 18.

The third path-relinking routine is called (*Path Relinking Remove\_best*). We want to transform route  $s$  into  $s'$ . On the first step we create one neighbor solution for each pair of vertices  $(p, d)$  in  $s$ : For each  $(p, d)$  in  $s$  remove it and reinsert the vertices of this pair in the positions as they appear in  $s'$ , creating a new route. Among all these new generated

---

**Algorithm 17** *Path Relinking* Prefix

---

```

1: begin
2: Input: Two feasible routes  $s$  and  $s'$  for the PDPLU.
3: Output: The best intermediate solution between  $s$  and  $s'$ , inclusive.
4: for  $i$  in  $[0, |s'|]$  do
5:    $s_i = \text{pref}(s', i) + (s - \text{pref}(s', i))$ 
6: return  $\min_{s_i} (\text{cost}(s_i))$ .
7: end

```

---



---

**Algorithm 18** *Path Relinking* Remove

---

```

1: begin
2: Input: Two feasible routes  $s$  and  $s'$  for the PDPLU.
3: Output: The best intermediate solution between  $s$  and  $s'$ , inclusive.
4:  $\text{newSolution} \leftarrow s$ .
5: while  $s \neq s'$  do
6:   Let  $i$  be the smallest position in  $s$  such that  $s(i) \neq s'(i)$ .
7:   Let  $(p, d)$  such that  $s(i) = p$  or  $s(i) = d$ .
8:   Remove  $p$  and  $d$  from  $s$ .
9:   Insert  $p$  and  $d$  in  $s$  on the same positions they appear in  $s'$ .
10:  if  $\text{cost}(s) < \text{cost}(\text{newSolution})$  then
11:     $\text{newSolution} \leftarrow s$ .
12: return  $\text{newSolution}$ .
13: end

```

---

solutions, select the one with the lowest cost as the new current one. Then repeat this process until the current route becomes equal to  $s'$ .

## 6.5 List of Routes

In this Section we discuss about a data structure used to improve the running time of the route feasibility test. All the proposed algorithms for the PDPLU use the 2KPLU algorithms to check whether a route is feasible or not. The exact algorithm based on the ILP model always checks the feasibility of complete routes (routes that contain all clients). The heuristics from sections 6.4.2 and 6.4.3 can also check the feasibility of smaller routes while they are being constructed. To solve reasonable sized instances, millions of feasibility tests are done by the algorithms and heuristics.

One may note that many of these tests are useless. Consider for example a feasible route  $r = (p_1, p_2, p_3, d_2, d_3, p_4, d_1, d_4)$  with its corresponding packing  $\mathcal{P}$ . Now consider that the feasibility of  $r$  was already checked, and a route  $s = (p_1, p_3, d_3, d_1)$  in construction is going to be tested. It is easy to see that  $s$  is feasible with packing defined by a subset

of  $\mathcal{P}$ . Thus, keeping a cache of already checked routes, we may discover the feasibility of routes generated by the algorithm without executing the packing algorithms.

To be more formal: Given that a route  $r = (r_1, \dots, r_n)$  is feasible, with corresponding packing  $\mathcal{P}$ , then a route  $s = (s_1, \dots, s_m)$  such that  $s \subseteq r$ , is also feasible. Furthermore,  $\mathcal{P}' = \{a : a \in \mathcal{P}, a \in s\}$  is a feasible packing for  $s$ . On the other hand suppose  $r$  is infeasible. If  $s \supseteq r$  then  $s$  is infeasible.

We maintain a list of the routes already tested while solving a PDPLU instance. Before trying to test if a route is feasible or not, a pretest is made checking the list of routes already tested. If no result comes from the pretest then the packing algorithm is used and the result is stored on the list. By doing this, we aim to reduce the packing time, thus the overall time of the algorithms.

To be more descriptive, the data structure Routes List (RL) maintains two sets of routes (feasible/infeasible) and a set of packings (for the feasible routes). The operations that can be performed by the RL are:

- Insert a feasible route and its packing.
- Insert a infeasible route.
- Check whether a route is feasible, infeasible or unknown.

The first approach to implement this data structure is simple. Keep an array of arrays for each set. The insertion operations can be done in  $O(1)$  time-complexity. To check if a route  $s$  is feasible or not, one need to compare this route against each route in the data structure. This can be done in  $O(kn)$  where  $n$  is the size of the largest route and  $k$  is the number of routes stored.

Our experiments showed that even this simple approach improved the total execution time of the heuristics in 50% on average.

After some improvements in the data-structure we achieved a version of it that reduced the execution time of the heuristics in 90% on average. First, aiming to reduce the number of routes on the list, the following modification was applied to the insertion operation: while inserting a new feasible route  $r$ , if there is a feasible route  $s$  on the set, such that  $s \subseteq r$ , then  $s$  can be removed from the set without losing any coverage. A similar improvement can be done for the infeasible routes. Another improvement comes from the simple observation that if a route  $s$  contains an element that is not in  $r$  then  $s \not\subseteq r$ . So using only a set of bits and one operation we can test this before testing if  $s \subseteq r$ .

It is important to notice that if we use this approach with heuristics for the 2KPLU then the conclusions about infeasibility may not hold. In this case they would serve as an heuristic.

## 6.6 Generated Instances

The algorithms were tested using adapted instances from the PDP. The instances were generated from the set of 35 instances used in [7]. Each one of these 35 instances consists of a graph representing the depot, and pickup and delivery pairs. The number of pickup and delivery pairs varies from 5 to 35. For each instance we generated 10 new ones by creating 2D items to be carried from each pickup vertex to its corresponding delivery vertex. We used a similar approach as in [11], that created 2D items using the graph instances for the PDP.

The instances can be separated in four classes depending on the characteristics of the generated items: (1) unitary instances, (2) tall and narrow, (3) short and wide, and (4) homogeneous. Remember that the size of the vehicle is  $(W, H)$ . For instances of classes (2), (3) and (4) we use the following cutting method to create a predefined number of items. First create a list  $L$  with some items of dimensions  $(W, H)$ . Repeat the following method until the predefined number of items is not obtained: select an item from  $L$  and cut it into two new ones by doing a guillotined horizontal or vertical cut at some random position. Replace the original item with these two new ones in  $L$ . In what follows are the details of how instances were generated:

- **unitary:** Generate an unitary square for each pair  $(p, d)$ .
- **tall and thin:** In the cutting method, cut an item vertically with probability 80% otherwise cut it horizontally. Three instances are created as follows:
  1. Create  $n$  items using the cutting method, starting with  $L = \{(W, H)\}$ . Assign one item of the list for each pair of pickup and delivery at random.
  2. Create  $3n/2$  items using the cutting method, starting with  $L = \{(W, H), (W, H)\}$ . Assign one item for each pair randomly and the remaining  $n/2$  items are randomly assigned to the  $n$  pairs.
  3. Create  $2n$  items using the cutting method, with  $L = \{(W, H), (W, H), (W, H)\}$  to start. Assign one item for each pair randomly and the remaining  $n$  items are randomly assigned to the  $n$  pairs.
- **short and wide:** In the cutting method, cut an item vertically with probability 20% otherwise cut it horizontally. Three instances are created with  $n$ ,  $3n/2$  and  $2n$  items using the same approach of the last instance class.
- **homogeneous:** In the cutting method, for each selected item, cut it in its largest dimension and with cut between [35% and 65%] of the size of this dimension. Again

three instances are created with  $n$ ,  $3n/2$  and  $2n$  items using the same approach of the last instance class.

Thus we have 350 instances with  $5 \leq n \leq 35$  and  $5 \leq |L| \leq 70$ . Furthermore we can evaluate the algorithms with different packing characteristics. These instances are publicly available at: [www.loco.ic.unicamp.br/instances/pdplu/](http://www.loco.ic.unicamp.br/instances/pdplu/).

## 6.7 Computational experiments

The algorithms were coded in C++ and executed on an Intel i7-2600 3.40GHz processor with 8 GB 1333 MHz DDR3 of main memory. The stopping criteria for all algorithms was 2 CPU hours. For the GRASP heuristics we used a limit of 1000 iterations or 100 without improvement.

The raw results can be found at: [www.loco.ic.unicamp.br/instances/pdplu/](http://www.loco.ic.unicamp.br/instances/pdplu/). The instances are named as follows:  $\text{pdp\_type\_t}$ , where “pdp” is the PDP instance used,  $\text{type} \in \{\text{“unitary”}, \text{“homogeneous”}, \text{“tall”}, \text{“short”}\}$  and  $t \in \{1, 2, 3\}$  that corresponds respectively to the number of items generated ( $n$ ,  $3n/2$ , or  $n$ ). For the unitary type,  $t$  is always equal to 1.

We used the following notations for the algorithms tested:  $\mathcal{A}(\mathcal{B}[\mathcal{B}])$ , where  $\mathcal{A}$  is the algorithm for the PDPLU problem and  $\mathcal{B}$  is a 2KPLU algorithm. Moreover the algorithms for the 2KPLU are used exactly in the given order. For instance  $G(\text{BL}, \text{CP})$  stands for the GRASP heuristic combined with both the BL heuristic and the CP algorithm (when the BL fails).

In order to solve the problems up to optimality with the  $\text{ILP}_{\text{PDPLU}}$  algorithm, we used some combinations of exact algorithms and heuristics for the 2KPLU. In general we tested  $\text{ILP}_{\text{PDPLU}}(\text{RC})$  and  $\text{ILP}_{\text{PDPLU}}(\text{CP})$ , and since the best results were due to  $\text{ILP}_{\text{PDPLU}}(\text{CP})$  we also tried the version  $\text{ILP}_{\text{PDPLU}}(\text{BL}, \text{TP}, \text{OVERLAP}_{LU}, \text{CP})$ , in order to check if the execution of heuristics for the packing problem could accelerate the feasibility check. However, in general the heuristics only increased the running time of the  $\text{ILP}_{\text{PDPLU}}$  algorithm, since the number of feasible packings tested is usually really small when compared with the number of infeasible ones. So we only present the results comparing the exact algorithms  $\text{ILP}_{\text{PDPLU}}(\text{CP})$  and  $\text{ILP}_{\text{PDPLU}}(\text{RC})$ .

We present some graphs showing the *performance profiles* [6] of the algorithms. This method of comparing optimization softwares and algorithms, was proposed by Dolan and Moré in [6]. See for example the graph in Figure 6.10. The idea is to compare algorithms among themselves, computing for each instance of the problem the ratio between the running time (or solution cost) of one algorithm and the best achieved running time (or solution cost) for the instance. We compute all such ratios for all instances. If, for

example, some algorithm has ratio equal to 1 for some instance, this means that this algorithm obtained the best result for this instance among all algorithms. With the ratios computed for all instances, we can generate the performance profiles. In the  $x$  axis we have all possible values of ratios. For any given ratio, in the  $y$  axis we have the percentage of instances that an algorithm could solve, such that the ratio of its running time to the best one is at most the given ratio. We can also generate performance profiles comparing the quality of solutions generated by the algorithms. For the exact algorithms this is not done, since they always produce optimal solutions, but we construct performance profiles comparing the quality of solutions for the heuristics.

The  $\text{ILP}_{\text{PDPLU}}(\text{CP})$  algorithm solved 99 instances up to optimality, while the algorithm  $\text{ILP}_{\text{PDPLU}}(\text{RC})$  solved 73. Besides that, if we consider the instances where no optimal solution was found, in general the solutions of the  $\text{ILP}_{\text{PDPLU}}(\text{CP})$  are of better quality. In Figure 6.10 we present the performance profiles for the cost and running time of the exact algorithms. The CP algorithm clearly outperforms the RC, having the best running time in about 80% of the instances. Besides that, it did not found the best cost solution in only 2 instances among 164 (those that some result was found by at least one of the algorithms).

If we consider the different types of instances, the ones with “unitary” items turned out to be the easiest ones, as expected. The  $\text{ILP}_{\text{PDPLU}}(\text{CP})$  algorithm solved 16 instances among the 35 tested. On the other hand, the instances with “short and wide” items were the hardest ones with only 22 instances solved to optimality (among 105). It seems that the wider the items are, the harder the packing problem is, mainly due to the loading and unloading constraints. The  $\text{ILP}_{\text{PDPLU}}(\text{RC})$  algorithm solved the same number of “unitary” instances as the  $\text{ILP}_{\text{PDPLU}}(\text{CP})$ . It also had a lower running time in some instances.

The exact algorithms found optimal solutions in 99 of the 350 instances tested. All instances with 5 pairs of vertices were solved up to optimality, however no instance with more than 20 pairs of vertices was solved to optimality. The results where optimal solutions were found, are used to calculate the *gap* between solutions computed by the heuristics and the optimal.

We used a smaller set of instances to set up the GRASP parameters. In Table 6.1 we provide the parameters tested and the ones used in our final experiments. For the 4-Opt\*\*\* heuristic we used the parameters as described in [18].

In our tests the GRASP heuristic achieved the best results and that is why we show more versions of it in our analysis.

The heuristics were able to found optimal solutions for several instances (considering the 99 instances with known optimal values). The GRASP heuristics  $\text{G}(\text{OVERLAP}_{LU})$  and  $\text{G}(\text{BL,TP,OVERLAP}_{LU})$  found solutions with optimal value for more instances, 64

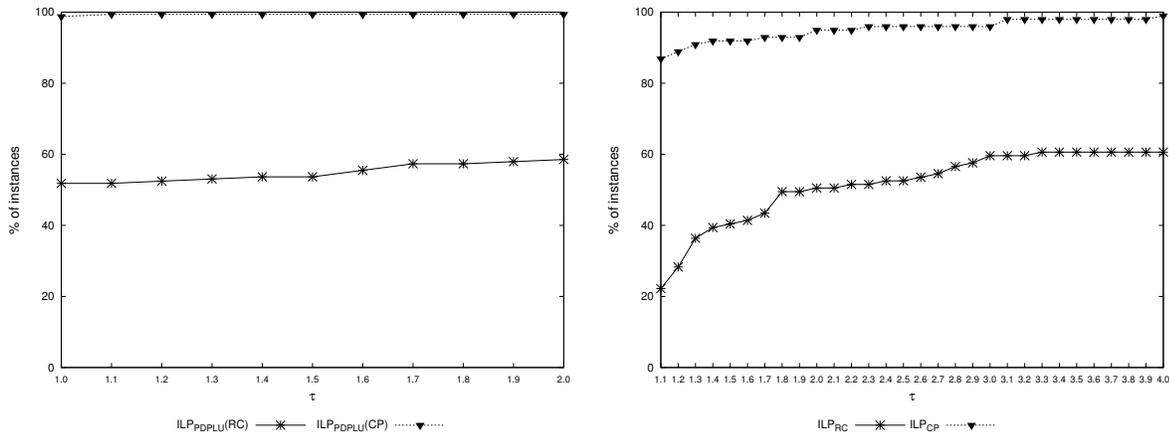


Figure 6.10: Performance profiles for the cost(left) and running time(right) of the exact algorithms for the PDPLU.

group	Parameter	Tested	Best
Reactive GRASP	Reactive Sets	{(10%, 20%, 50%, 100%), (10%, 20%, ..., 100%)}	(10%, 20%, 50%, 100%)
	It_update	{every iteration, 20, 50, 100, 150}	100
	Aggressiveness	{1,2,4,8}	4
Path Relinking	PRAlgorithm	{Prefix, Remove, Remove_Best}	Remove_Best
	Pool Size	{10,20,50}	20
	Direction	{Current to Elite, Elite to Current, Both Ways}	Elite to Current
	Filter	{5%, 10%, 15%}	10%
Local Search	lsAlgorithm	{Pairs, 4-Opt***}	Pairs

Table 6.1: Parameters for the GRASP heuristic.

instances each (among 99 known). Besides that, the worst result was due to algorithm 4-Opt\*\*\*(BL,TP,OVERLAP<sub>LU</sub>), that found optimal solutions to 41 instances.

In Figure 6.11 we present the performance profiles of the heuristics considering all instances that have at least 10 pairs of vertices and took more than 1 second of running time, while in the Figures 6.12 to 6.15 we separate the results in Figure 6.11 by instance type. In each one of these figures, the graph to the left is the performance profiles considering the cost of the solutions, while the graph to the right presents the performance profiles considering the running times of the heuristics.

In the graph to the left of Figure 6.11, we see that G(BL,TP,OVERLAP<sub>LU</sub>) found the largest number of best solutions (about 80%), with the heuristic G(OVERLAP<sub>LU</sub>) very close to it. It obtained the best results in 209 instances, against 178 from G(OVERLAP<sub>LU</sub>), 125 from G(TP), 116 from G(BL) and, 48 from 4-Opt\*\*\*(BL,TP,OVERLAP<sub>LU</sub>). On the other hand the heuristic G(BL,TP,OVERLAP<sub>LU</sub>) is the slowest one as one can see from the graph to the right of the figure 6.11.

One interesting point to notice is that if we consider the instances for which the ILP<sub>PDPLU</sub>(CP) algorithm found sub-optimal solutions (64 instances), the best solution found by one of the GRASP heuristics was better in 53 instances. Considering the in-

stances for which an optimal solution was found by the exact algorithms, the average gap between the solutions of the *GRASP* heuristics and the optimal ones was 13.23%.

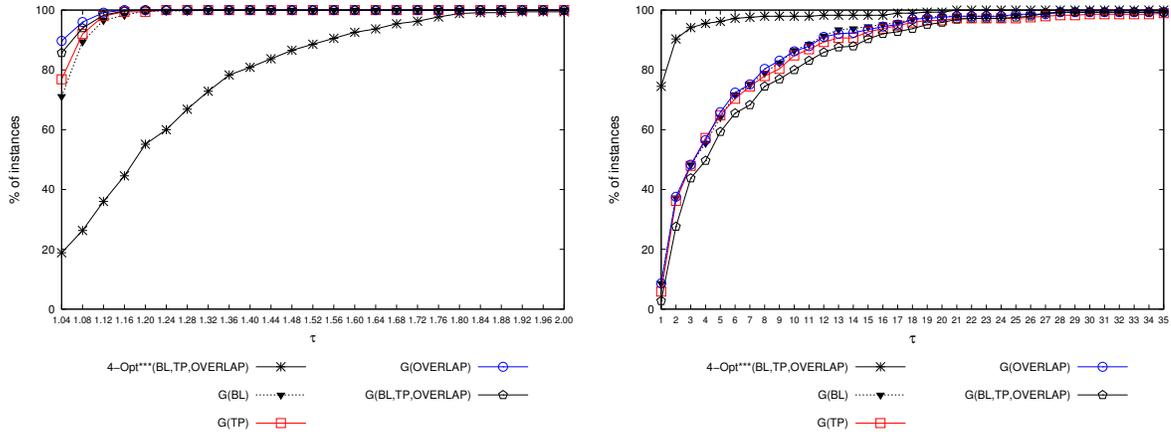


Figure 6.11: Cost(left) and time(right) factor for the heuristics in all instances.

From Figure 6.12 we can see that all GRASP heuristics were effective in finding good solutions for the unitary instances. The GRASP heuristics found almost 100% of the best solutions, while the heuristic 4-Opt\*\*\* generated the best solutions in only about 20% of the instances. It is interesting to notice that for these type of instances the GRASP heuristics were faster than the 4-Opt\*\*\* heuristic. The best results were found by the G(OVERLAP<sub>LU</sub>) heuristic in 28 instances, followed by 26 from G(BL,TP,OVERLAP<sub>LU</sub>) (see Fig. 6.12).

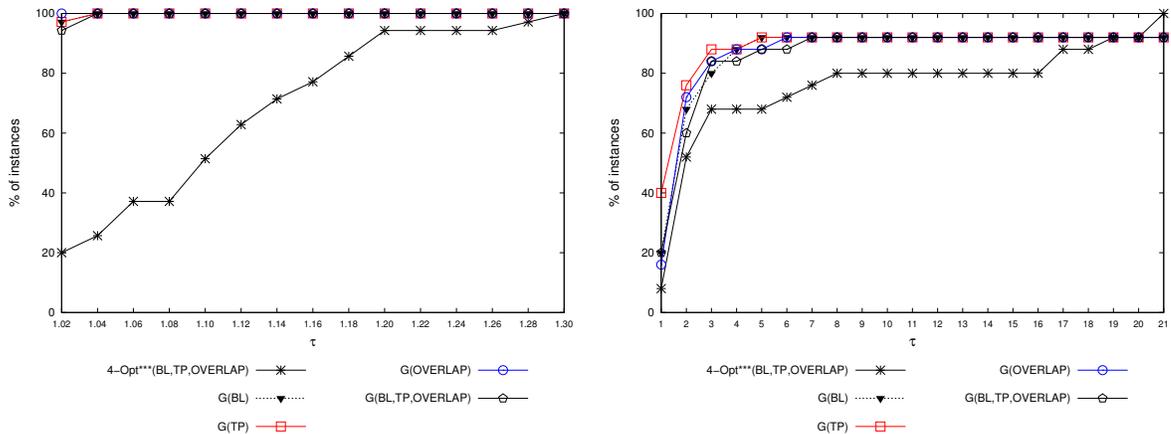


Figure 6.12: Performance profiles for the solutions cost (left) and running time (right) for the heuristics. Instances with unitary items.

For the remaining type of instances (Figures 6.13 to 6.15), the results were similar.

The GRASP heuristics were the ones that found the largest number of best solutions with  $G(BL,TP,OVERLAP_{LU})$  founding more than 80% of the best solutions, while the 4-Opt\*\*\* heuristic founding less than 20% of the best solutions. On the other hand the 4-Opt\*\*\* heuristic was the fastest heuristic being more than 4 times faster than the GRASP heuristics in more than 40% of the instances.

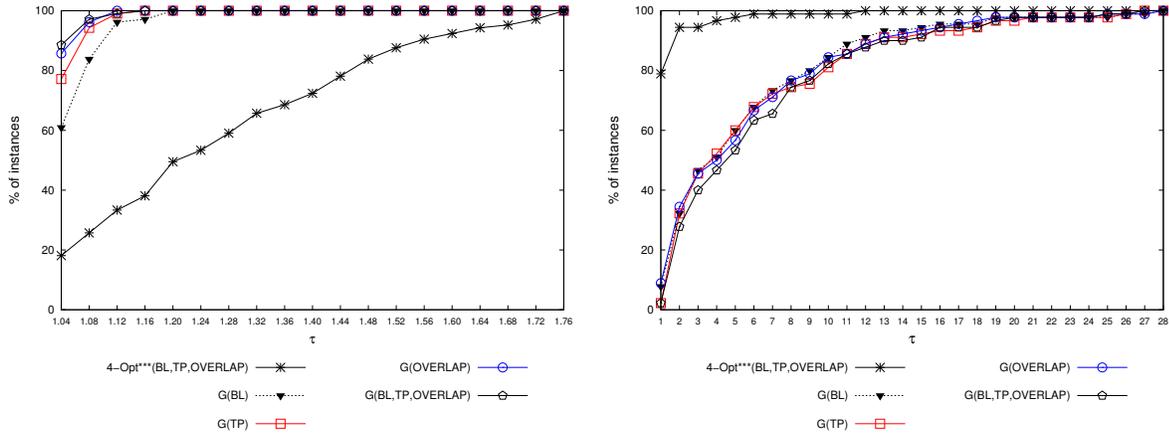


Figure 6.13: Performance profiles for the solutions cost (left) and running time (right) for the heuristics. Instances with “short and wide” items.

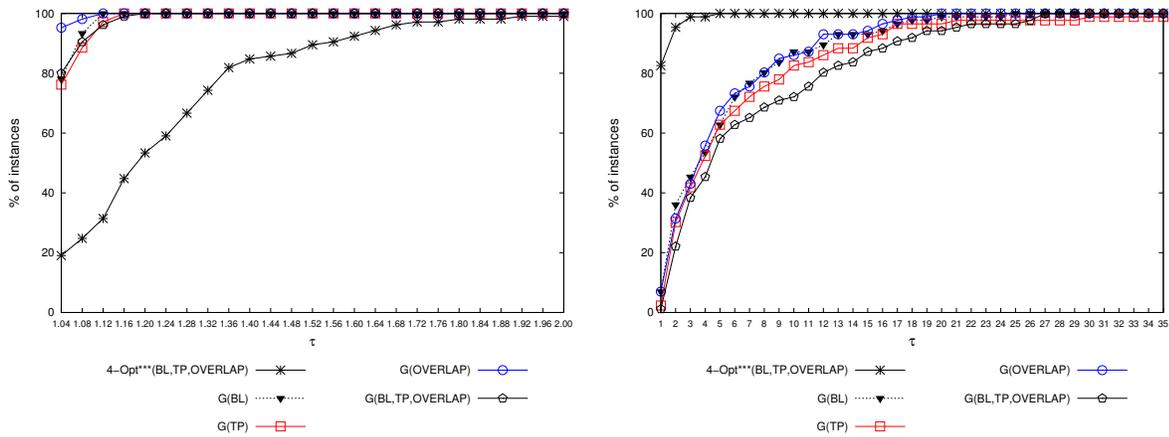


Figure 6.14: Performance profiles for the solutions cost (left) and running time (right) for the heuristics. Instances with “tall and thin” items.

In Figure 6.16 we present the running time of the GRASP heuristics as a function of the instance size being solved. The time did not increase as expected. It seems that the heuristics running time increases linearly from 20 to 35 pairs of items, despite its time complexity. The explanation for this observation comes from the use of the Routes List

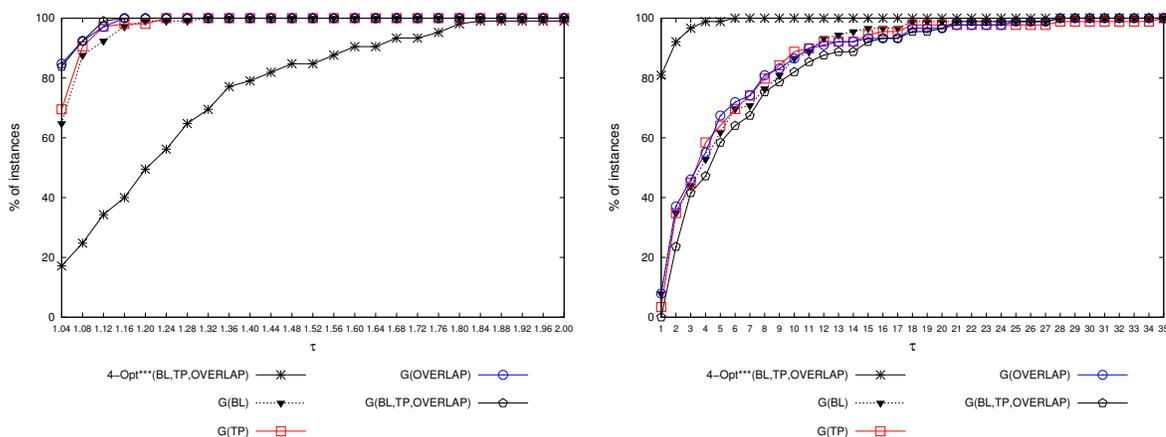


Figure 6.15: Performance profiles for the solutions cost (left) and running time (right) for the heuristics. Instances with “homogeneous” items.

data structure. In Figure 6.17, for each algorithm, we present the average proportion of the number of packing sub-problems solved using the data structure (avoiding the use of the packing algorithms again). Notice that the more packing problems solved with the data structure, the faster the algorithm runs. From the figure we can see that the proportion of sub-problems solved with the data structure stop decreasing in instances with 25 pairs of vertices (about 70% of the sub-problems are solved with the data structure). Then from instances with 25 to 35 pairs of vertices, the proportion of sub-problems solved stabilizes and increase a little bit to about 75%. This explains why the increase of running time diminishes for the instances of these sizes.

## 6.8 Conclusions

In this paper we proposed heuristics and exact algorithms for the *Pickup and Delivery Problem with Two Dimensional Loading/Unloading Constraints* (PDPLU). The packing part of the problem (*Two Dimensional Knapsack Problem with Loading and Unloading Constraints*) was first studied in [13], and to the best of our knowledge this is the first work to study the PDPLU and provide practical results for it.

Our exact algorithms for the PDPLU were based on an ILP model for the PDP found in the literature. We combined this model with exact packing algorithms: one based on Corner Points and another one based on *Constraint Programming* (CP). We also developed heuristics for both problems (2KP and PDPLU). The *GRASP* based heuristic achieved the best results. We tested several versions of the *GRASP* heuristic choosing good methods of the *Local Search*, and the *Path Relinking*. During these tests we also selected the best

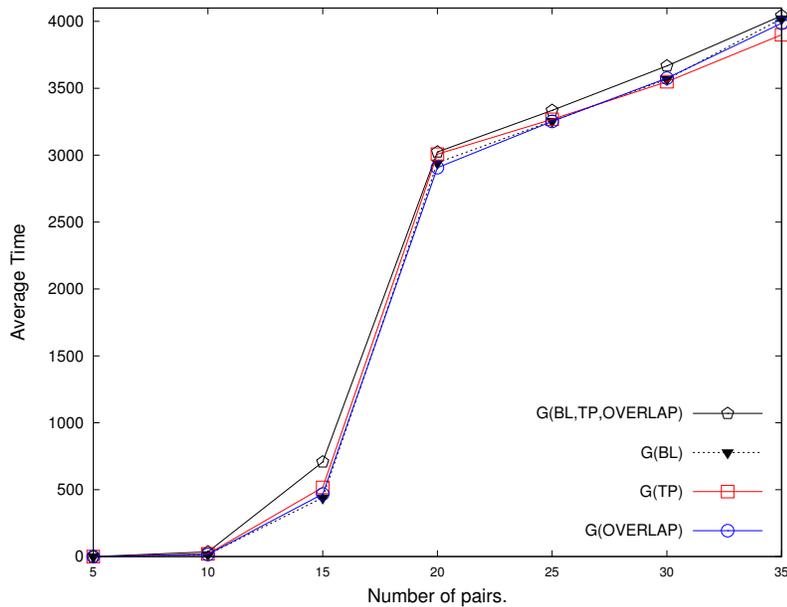


Figure 6.16: For each heuristic, the figure presents the average time in seconds used to solve an instance of a given size.

parameters for the *Reactive* part of the *GRASP*.

The best optimal algorithm was composed by the ILP model for the PDP together with the CP algorithm to solve the 2KPLU. Among the heuristics, the *GRASP* together with the three packing heuristic obtained the best results.

We performed several tests to assess the quality of the proposed algorithms. Our exact algorithms were able to solve 99 instances up to optimality (from 350). They were able to solve some instances with 20 pairs of clients and 40 items.

The heuristics showed to be a good alternative when sub-optimal solutions are acceptable. The best *GRASP* heuristic was able to solve 64 instances to optimality (among 99).

## 6.9 References

- [1] B. S. Baker, Jr. E. G. Coffman, and R. L. Rivest. Orthogonal packings in two dimensions. *SIAM Journal on Computing*, 9(4):846–855, 1980.
- [2] B. Chazelle. The bottom-left bin-packing heuristic: An efficient implementation. *IEEE Trans. Comput.*, 32(8):697–707, 1983.
- [3] N. Christofides and C. Whitlock. An algorithm for two-dimensional cutting problems. *Operations Research*, 25(1):30–44, 1977.

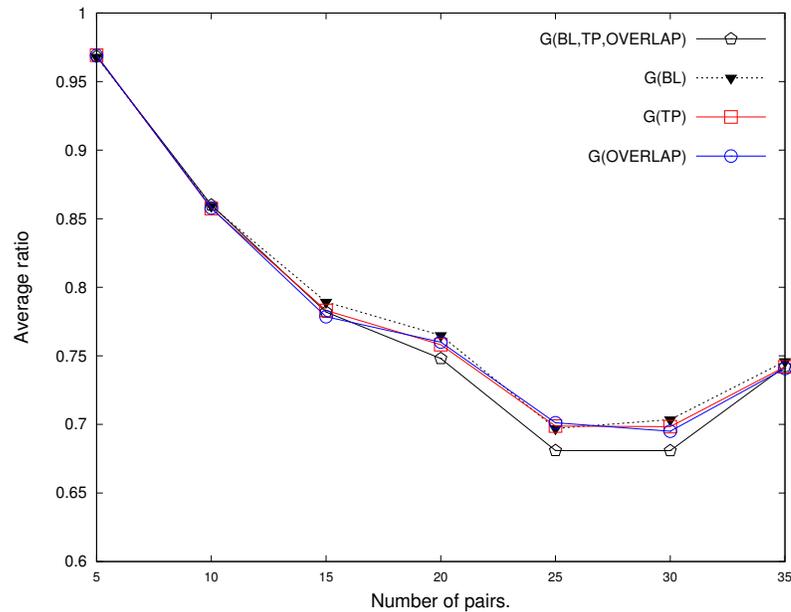


Figure 6.17: For each heuristic, and for each instance size, the figure presents the average proportion of packing sub-problems solved by the Routes List data structure.

- [4] J. L. M. da Silveira, F. K. Miyazawa, and E. C. Xavier. Heuristics for the strip packing problem with unloading constraints. *Computers & Operations Research*, 40(4):991 – 1003, 2013.
- [5] J. L. M. da Silveira, E. C. Xavier, and F. K. Miyazawa. Two dimensional knapsack with unloading constraints. *Electronic Notes in Discrete Mathematics*, 37:267–272, 2011.
- [6] E. D. Dolan and J. J. Morér. Benchmarking optimization software with performance profiles. *Mathematical Programming*, 91(2):201 – 213, 2002.
- [7] I. Dumitrescu, S. Ropke, J.-F. Cordeau, and G. Laporte. The traveling salesman problem with pickup and delivery: polyhedral results and a branch-and-cut algorithm. *Mathematical Programming*, 121(2):269–305, 2010.
- [8] G. Fuellerer, K. F. Doerner, R. F. Hartl, and M. Iori. Ant colony optimization for the two-dimensional loading vehicle routing problem. *Computers & OR*, 36(3):655–673, 2009.
- [9] M. Gendreau, M. Iori, G. Laporte, and S. Martello. A tabu search heuristic for the vehicle routing problem with two-dimensional loading constraints. *Networks*, 51(2):153, 2008.

- [10] E. Hadjiconstantinou and N. Christofides. An exact algorithm for general, orthogonal, two-dimensional knapsack problems. *European Journal of Operational Research*, 83(1):39 – 56, 1995.
- [11] M. Iori, J.-J. S-Gonzalez, and D. Vigo. An exact approach for the vehicle routing problem with two-dimensional loading constraints. *Transportation Science*, 41(2):253–264, 2007.
- [12] A. Lodi, S. Martello, and D. Vigo. Heuristic and metaheuristic approaches for a class of two-dimensional bin packing problems. *INFORMS J. on Computing*, 11(4):345–357, 1999.
- [13] A. Malapert, C. Guéret, N. Jussien, A. Langevin, and L.-M. Rousseau. Two-dimensional pickup and delivery routing problem with loading constraints. In *First CPAIOR Workshop on Bin Packing and Placement Constraints (BPPC'08)*, 2008.
- [14] S. Martello, D. Pisinger, and D. Vigo. The three-dimensional bin packing problem. *Operations Research*, 48(2):256–267, 2000.
- [15] S. N. Parragh, K. F. Doerner, and R. F. Hartl. A survey on pickup and delivery problems. *Journal für Betriebswirtschaft*, 58(1):21–51, 2008.
- [16] J. Renaud, F. F. Boctor, and G. Laporte. A fast composite heuristic for the symmetric traveling salesman problem. *INFORMS Journal on Computing*, 8(2):134–143, 1996.
- [17] J. Renaud, F. F. Boctor, and J. Ouenniche. A heuristic for the pickup and delivery traveling salesman problem. *Computers & Operations Research*, 27(9):905 – 916, 2000.
- [18] J. Renaud, F. F. Boctor, and J. Ouenniche. Perturbation heuristics for the pickup and delivery traveling salesman problem. *Computers & Operations Research*, 29:905–916, 2002.
- [19] M. G. C. Resende and C. C. Ribeiro. Greedy randomized adaptive search procedures. In *Handbook of metaheuristics*, pages 219–249. Springer US, 2003.
- [20] K. S. Ruland and E. Y. Rodin. The pickup and delivery problem: Faces and branch-and-cut algorithm. *Computers & Mathematics with Applications*, 33(12):1 – 13, 1997.
- [21] E. E. Zachariadis, C. D. Tarantilis, and C. T. Kiranoudis. A guided tabu search for the vehicle routing problem with two-dimensional loading constraints. *European Journal of Operational Research*, 195(3):729 – 743, 2009.

## 6.10 Appendix: Proof of Lemma 1

Given a list  $L$  and the packing  $\mathcal{P}$ , we define a digraph  $D := (V, A)$ , where  $V := \{v_i : a_i \in L\}$  and  $A := \{(v_i, v_j) : a_i \prec a_j\}$ .

Suppose  $D$  does not contains oriented cycles, then we have that a partial order exists. Therefore, we just need to show that there is no oriented cycle in  $D$ .

This is done by induction in  $1 \leq k \leq |V|$ , showing that  $D$  does not have a cycle of size  $k$ . So we prove that for any path  $v_1 v_2 \dots v_k$  we have that  $(v_k, v_1) \notin A$ , thus finishing the proof.

**Base:** Consider the cases  $k \in 1, 2, 3$ . If  $k = 1$ , for sake of contradiction suppose the *loop*  $v_i v_i$  in  $D$  and thus  $a_i \prec a_i$  for some  $a_i \in L$ . Therefore, by (6.1) and (6.2),

$$x(a_i) \geq x(a_i) + w(a_i) \vee y(a_i) \geq y(a_i) + h(a_i) \rightarrow \leftarrow,$$

and so  $(v_i, v_i) \notin A$ .

If  $k = 2$ , then, for sake of contradiction we have  $v_i v_j v_i \in D$ , thus  $a_i \prec a_j \prec a_i$ . For each pair in the cycle, we have that (6.1) or (6.2) is true. Suppose (6.1) true in  $a_i \prec a_j$ . If (6.1) is also true in  $a_j \prec a_i$ , then we have the following absurd:

$$x(a_j) \geq x(a_i) + w(a_i) > x(a_i) \geq x(a_j) + w(a_j) > x(a_j) \rightarrow \leftarrow .$$

On the other hand, if (6.1) is false in  $a_j \prec a_i$ , then we have the following contradiction:

$$y(a_j) + h(a_j) > y(a_i) \geq y(a_j) + h(a_j) \rightarrow \leftarrow .$$

A similar analysis may be done if (6.2) is true in  $a_i \prec a_j$ . Therefore  $(v_j, v_i) \notin A$ .

Finally, consider  $k = 3$ . So, for sake of contradiction we have  $v_i v_j v_k v_i$ , which implies  $a_i \prec a_j \prec a_k \prec a_i$ . In this case we have 8 combinations for equations (6.1) and (6.2) (two in each  $\prec$  relation). We use the following notation below:  $(e)(f)(g)$  denotes “e” valid in  $a_i \prec a_j$ , “f” valid in  $a_j \prec a_k$  and, finally, “g” valid in relation  $a_k \prec a_i$ .

1. Case (1)(1)(1):

$$x(a_i) \geq_1 x(a_k) + w(a_k) > x(a_k) \geq_2 x(a_j) + w(a_j) > x(a_j) \geq_3 x(a_i) + w(a_i) > x(a_i) \rightarrow \leftarrow .$$

From  $a_k \prec a_i$  we have 1, from  $a_j \prec a_k$  we have 2, and 3 from  $a_i \prec a_j$ .

2. Case (1)(1)(2):

$$x(a_i) + w(a_i) >_1 x(a_k) \geq_2 x(a_j) + w(a_j) > x(a_j) \geq_3 x(a_i) + w(a_i) \rightarrow \leftarrow .$$

Where have 1 from  $a_k \prec a_i$ , 2 from  $a_j \prec a_k$  and 3 from  $a_i \prec a_j$ .

3. Case (1)(2)(1):

$$x(a_i) \geq_1 x(a_k) + w(a_k) >_2 x(a_j) \geq_3 x(a_i) + w(a_i) > x(a_i) \rightarrow \leftarrow .$$

Where 1 is from  $a_k \prec a_i$ , 2 from  $a_j \prec a_k$ , 3 from  $a_i \prec a_j$ .

4. Case (1)(2)(2):

$$y(a_j) + h(a_j) >_1 y(a_i) \geq_2 y(a_k) + h(a_k) > y(a_k) \geq_3 y(a_j) + h(a_j) \rightarrow \leftarrow .$$

Again 1 from  $a_i \prec a_j$ , 2 from  $a_k \prec a_i$  and 3 from  $a_i \prec a_j$ .

5. Case (2)(1)(1): Similar to (1)(2)(2).

6. Case (2)(1)(2): Similar to (1)(2)(1).

7. Case (2)(2)(1): Similar to (1)(1)(2).

8. Case (2)(2)(2): Similar to (1)(1)(1).

Thus  $(v_k, v_i) \notin A$ .

**Induction Hypothesis:** Let  $C = v_1 v_2 \dots v_{k'}$  be a path in  $D$  with  $k' < k$ , then  $(v_{k'}, v_1) \notin A$ , i. e.,  $a_{k'} \not\prec a_1$ .

**Step:** Let  $k \geq 4$ . For sake of contradiction suppose

$$v_1 \rightsquigarrow v_i \rightsquigarrow v_j \rightsquigarrow v_k v_1.$$

So we have the following configuration:

$$a_1 \prec \dots \prec a_i \prec \dots \prec a_j \prec \dots \prec a_k \prec a_1.$$

Let  $1 \leq i < i + 1 < j \leq k$  such that  $j \neq k$  or  $i > 1$  (two non-adjacent items on the configuration):

- If  $a_i \prec a_j \Rightarrow a_1 \prec \dots \prec a_i \prec a_j \prec \dots \prec a_k$ .

Since this sequence has size smaller than  $k$ , we have by Induction Hypotheses that

$$(v_k, v_1) \notin A \Rightarrow a_k \not\prec a_1 \rightarrow \leftarrow . \quad (6.17)$$

- If  $a_j \prec a_i \Rightarrow a_i \prec \dots \prec a_j \prec a_i$ .

As before, by Induction Hypothesis.

$$(v_j, v_i) \notin A \Rightarrow a_j \not\prec a_i \rightarrow \leftarrow . \quad (6.18)$$

So we just showed that there is no relation between any pair of non-adjacent items on the cycle. Now we divide the proof in two cases based on the relation between  $a_1$  and  $a_2$ .

**Case 1: Suppose (6.1) valid in  $a_1 \prec a_2$**

Thus, we have  $(x(a_2) \geq x(a_1) + w(a_1))$ .

So we prove by induction on  $1 \leq j < k$  that the item  $a_{j+1}$  is at right of  $a_j$ , formally,  $x(a_{j+1}) \geq x(a_j) + w(a_j)$ .

**Base 2:**  $j = 1$  was supposed to be true.

**Induction Hypothesis 2:** If  $1 \leq j < k$ , then  $x(a_{j+1}) \geq x(a_j) + w(a_j)$ .

**Step 2:** If  $j \geq 2$  and by Induction Hypothesis 2 we have that  $x(a_j) \geq x(a_{j-1}) + w(a_{j-1})$ .

If  $j = 2 \Rightarrow j + 1 = 3 < k$

If  $j \geq 3 \Rightarrow j - 1 > 1$

Thus, by (6.17) and (6.18) we have  $a_{j-1} \not\prec a_{j+1}$  and  $a_{j+1} \not\prec a_{j-1}$ . Thus:

- From  $a_{j+1} \not\prec a_{j-1}$  (6.6), we have that

$$y(a_{j+1}) \geq y(a_{j-1}) + h(a_{j-1}) \vee x(a_{j+1}) \geq x(a_{j-1}) + w(a_{j-1}) \quad (6.19)$$

- Analogously, from  $a_{j-1} \not\prec a_{j+1}$  we know that

$$y(a_{j-1}) \geq y(a_{j+1}) + h(a_{j+1}) \vee x(a_{j-1}) \geq x(a_{j+1}) + w(a_{j+1}) \quad (6.20)$$

Using Induction Hypothesis 2 ( $x(a_j) \geq x(a_{j-1}) + w(a_{j-1})$ ) and  $a_{j-1} \prec a_j$  (6.3) we find:

$$x(a_j) \geq x(a_{j-1}) + w(a_{j-1}) \wedge y(a_j) + h(a_j) > y(a_{j-1}) \quad (6.21)$$

Therefore, from  $a_j \prec a_{j+1}$  (6.3) we have  $x(a_{j+1}) + w(a_{j+1}) \geq x(a_j)$ , Thus, adding (6.21) we find:

$$x(a_{j+1}) + w(a_{j+1}) \geq x(a_j) \geq x(a_{j-1}) + w(a_{j-1}) > x(a_{j-1}) \quad (6.22)$$

using (6.20) and (6.22) we wave

$$y(a_{j-1}) \geq y(a_{j+1}) + h(a_{j+1}). \quad (6.23)$$

Finally, from (6.23) and  $a_{j-1} \prec a_j$  (6.3) we have  $y(a_j) + h(a_j) > y(a_{j-1}) \geq y(a_{j+1}) + h(a_{j+1}) > y(a_{j+1})$ . Thus, from  $a_j \prec a_{j+1}$ , we have that  $x(a_{j+1}) \geq x(a_j) + w(a_j)$ . Thus we proved the Induction Hypothesis 2.

From that we know that

$$x(a_1) + w(a_1) \leq x(a_2) < x(a_3) < \dots < x(a_k).$$

**Case 2: Suppose (2) valid in  $a_2 \prec a_1$**

Thus  $y(a_2) \geq y(a_1) + h(a_1)$ . By symmetry with **Case 1**, we have that:

$$y(a_1) + h(a_1) \leq y(a_2) < y(a_3) < \dots < y(a_k).$$

Since **Case 1** or **Case 2** are valid then:  $x(a_k) > x(a_1) + w(a_1) \vee y(a_k) > y(a_1) + h(a_1)$ , which means  $v_k \not\prec v_1$  (6.6)  $\rightarrow \leftarrow$ .



# Capítulo 7

## Considerações Finais

Neste trabalho apresentamos vários algoritmos para problemas de roteamento e empacotamento. Desenvolvemos algoritmos combinatórios exatos, aproximados e heurísticas. Também implementamos algoritmos baseados em PLI e CP. Estas abordagens nos permitiram atacar diversos problemas diferentes alcançando sempre bons resultados.

No Capítulo 4 apresentamos algoritmos para o *Two Dimensional Knapsack Problem with Unloading Constraints*. Este é uma versão do clássico 2KP, com a adição de que os itens possuem uma ordem de remoção da mochila. Esta restrição define que itens devem possuir um caminho livre para serem removidos do *bin* em ordem. Neste trabalho apresentamos os primeiros resultados da literatura para o problema: Uma  $(4 + \epsilon)$ -aproximação para o caso em que o *bin* é um quadrado e duas  $(3 + \epsilon)$ -aproximações para casos especiais do problema.

No Capítulo 5 mostramos algoritmos aproximados para o *Strip Packing Problem with Unloading Constraints*. Este é uma versão do clássico SPP, com uma generalização similar a do *Two Dimensional Knapsack Problem with Unloading Constraints*. Consideramos duas versões deste problema, uma onde apenas um movimento vertical pode ser realizado para remover os itens do *bin* e outra onde um movimento horizontal adicional também pode ser realizado. Neste trabalho também atacamos versões paramétricas destes problemas, ou seja, onde os itens possuem tamanhos limitados por um fator. Os principais resultados são uma 5.745-aproximação para a versão com apenas um movimento vertical e uma 3-aproximação para o segundo caso. Os resultados apresentados no Capítulo 5 melhoram os resultados apresentados por Silveira *et al.* em [11].

Por fim, no Capítulo 6 atacamos o problema *Pickup and Delivery Problem with Two Dimensional Loading/Unloading Constraints*. Este problema é uma generalização do clássico PDP, com a adição de uma restrição de empacotamento (é necessário transportar itens retangulares entre pares de clientes). Este empacotamento deve satisfazer restrições de carregamento e descarregamento dos itens. Neste trabalho propusemos al-

goritmos exatos e uma heurística GRASP para o problema. Mostramos que a técnica de CP obteve os melhores resultados entre os algoritmos exatos para o problema de empacotamento, enquanto que a heurística GRASP obteve bons resultados em tempo razoável para o problema geral.

## 7.1 Trabalhos Futuros

Do ponto de vista do problema 2L-CVRP uma pergunta surge após os resultados apresentados no Capítulo 6: Será que modelos de *Constraint Programming* melhorariam os melhores resultados exatos para o 2L-CVRP, apresentados por Iori *et al.* em [16]? A utilização desta técnica obteve melhorias consideráveis quanto comparadas ao algoritmo que utiliza *Pontos de Contorno*, que também é utilizado por Iori *et al.* [16].

Quanto ao modelo de *Constraint Programming* apresentado no Capítulo 6 para o 2KPLU: Como podemos refiná-lo de forma a reduzir seu tempo de execução? Uma possível forma seria utilizando o conceito de *pontos de discretização de largura* ou *Reduced Raster Points* [46, 4, 15].

Quanto às heurísticas para o 2KPLU: Será que heurísticas mais elaboradas como a heurística GRASP apresentada por Silveira *et al.* [11] gerariam melhores resultados para o problema PDPLU como um todo? Estas mostraram-se efetivas para o problema 2L-CVRP, portanto, o mesmo pode ser esperado para o PDPLU.

Quanto às restrições de carregamento e descarregamento: No Capítulo 5, mostramos uma forma de “relaxar” a restrição de descarregamento de forma a permitir mais de um movimento para a remoção dos itens. Isto permite uma abordagem ainda mais prática dos problemas atacados na tese. Desta forma, seria interessante estudar tanto o 2KPU quanto o 2KPLU sob esta perspectiva. Além disto, analisar outras formas de relaxamento das restrições de forma a modelar outras situações práticas.

Outra questão que permanece em aberto é quanto à existência de algoritmos aproximados para o 2KPLU. No Capítulo 4 apresentamos diversos algoritmos para uma versão restrita que considera apenas a restrição de descarregamento. Talvez o uso direto de empacotamento em níveis não funcione de maneira tão intuitiva como no 2KPU, já que pode não existir uma ordenação viável dentro de cada nível.

Quanto ao PDPLU, seria interessante averiguar o desempenho das heurísticas *GRASP* apresentadas no capítulo 6 no contexto do problema PDP (desconsiderando-se o problema de empacotamento associado). A melhor heurística para o PDP foi consistentemente superada pelas heurísticas *GRASP* no contexto do PDPLU e talvez, com pequenas adaptações, o mesmo aconteça no caso do PDP.

# Referências Bibliográficas

- [1] S. Arora and C. Lund. *Hardness of approximations*. PWS Publishing, 1997.
- [2] B. S. Baker, Jr. E. G. Coffman, and R. L. Rivest. Orthogonal packings in two dimensions. *SIAM Journal on Computing*, 9(4):846–855, 1980.
- [3] M. Bellmore and G. L. Nemhauser. The Traveling Salesman Problem: A Survey. *Operations Research*, 16(3):538–558, may 1968.
- [4] E. G. Birgin, R. D. Lobato, and R. Morabito. An effective recursive partitioning approach for the packing of identical rectangles in a rectangle. *Journal of the Operational Research Society*, 61(2):306–320, 2009.
- [5] A. Borodin and R. El-Yaniv. *Online Computation and Competitive Analysis*. Cambridge University Press, 1998.
- [6] B. Chazelle. The bottom-left bin-packing heuristic: An efficient implementation. *IEEE Trans. Comput.*, 32(8):697–707, 1983.
- [7] A. Cobham. The Intrinsic Computational Difficulty of Functions. In Y. Bar-Hillel, editor, *Logic, Methodology and Philosophy of Science, proceedings of the second International Congress, held in Jerusalem, 1964*, Amsterdam, 1965. North-Holland.
- [8] E. G. Coffman Jr., M. R. Garey, and D. S. Johnson. An application of bin-packing to multiprocessor scheduling. *SIAM Journal on Computing*, 7(1):1–17, feb 1978.
- [9] A. Coloni, M. Dorigo, and V. Maniezzo. Distributed optimization by ant colonies. In *Proceedings of the first European conference on artificial life*, volume 142, pages 134–142. Paris, France, 1991.
- [10] J. L. M. da Silveira. Algoritmos de aproximação para problemas de empacotamento em faixa com restrições de descarregamento. Master’s thesis, Universidade Estadual de Campinas, Campinas, Brazil, 2011.

- [11] J. L. M. da Silveira, F. K. Miyazawa, and E. C. Xavier. Heuristics for the strip packing problem with unloading constraints. *Computers & Operations Research*, 40(4):991 – 1003, 2013.
- [12] J. L. M. da Silveira, E. C. Xavier, and F. K. Miyazawa. Two dimensional knapsack with unloading constraints. *Electronic Notes in Discrete Mathematics*, 37:267–272, 2011.
- [13] J. L. M. da Silveira, E. C. Xavier, and F. Keidi Miyazawa. Two dimensional strip packing with unloading constraints. *Electronic Notes in Discrete Mathematics*, 37:99–104, 2011.
- [14] G. B. Dantzig and J. H. Ramser. The truck dispatching problem. *Management Science*, 6(1):80–91, 1959.
- [15] T. A. de Queiroz, F. K. Miyazawa, Y. Wakabayashi, and E. C. Xavier. Algorithms for 3d guillotine cutting problems: Unbounded knapsack, cutting stock and strip packing. *Computers & Operations Research*, 39(2):200 – 212, 2012.
- [16] K. F. Doerner, G. Fuellerer, R. F. Hartl, M. Gronalt, and M. Iori. Metaheuristics for the vehicle routing problem with loading constraints. *Networks*, 49(4):294–307, 2007.
- [17] I. Dumitrescu, S. Ropke, J.-F. Cordeau, and G. Laporte. The traveling salesman problem with pickup and delivery: polyhedral results and a branch-and-cut algorithm. *Mathematical Programming*, 121(2):269–305, 2010.
- [18] H. Dyckhoff. A typology of cutting and packing problems. *European Journal of Operational Research*, 44(2):145 – 159, 1990.
- [19] T. A. Feo and M. G. C. Resende. A probabilistic heuristic for a computationally difficult set covering problem. *Operations Research Letters*, 8(2):67 – 71, 1989.
- [20] T. A. Feo and M. G.C. Resende. Greedy randomized adaptive search procedures. *Journal of Global Optimization*, 6:109–133, 1995.
- [21] C. E. Ferreira, C. G. Fernandes, F. K. Miyazawa, J. A. R. Soares, J. C. Pina Jr., K. S. Guimarães, M. H. Carvalho, M. R. Cerioli, P. Feofiloff, R. Dahab, and Y. Wakabayashi. *Uma introdução sucinta a algoritmos de aproximação*. Colóquio Brasileiro de Matemática -IMPA, Rio de Janeiro - RJ, 2001.
- [22] J. S. Ferreira, M. A. Neves, and P. Fonseca e Castro. A two-phase roll cutting problem. *European Journal of Operational Research*, 44(2):185 – 196, 1990.

- [23] G. Fuellerer, K. F. Doerner, R. F. Hartl, and M. Iori. Ant colony optimization for the two-dimensional loading vehicle routing problem. *Computers & OR*, 36(3):655–673, 2009.
- [24] N. Fujimoto and K. Hagihara. Near-optimal dynamic task scheduling of independent coarse-grained tasks onto a computational grid. In *Parallel Processing, 2003. Proceedings. 2003 International Conference on*, pages 391–398, 2003.
- [25] M. R. Garey and D. S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1990.
- [26] M. Gendreau, M. Iori, G. Laporte, and S. Martello. A tabu search heuristic for the vehicle routing problem with two-dimensional loading constraints. *Networks*, 51(2):153, 2008.
- [27] F. Glover. Future paths for integer programming and links to artificial intelligence. *Computers & Operations Research*, 13(5):533–549, 1986.
- [28] F. Glover. Tabu search and adaptive memory programming – advances, applications and challenges. In *Interfaces in Computer Science and Operations Research*, pages 1–75. Kluwer, 1996.
- [29] D. S. Hochbaum. *Approximation Algorithms for NP-Hard Problems*. PWS Publishing Company, 1997.
- [30] J. H. Holland. *Adaptation in natural and artificial systems: An introductory analysis with applications to biology, control, and artificial intelligence*. U. Michigan Press, 1975.
- [31] E. G. Coffman Jr., M. R. Garey, D. S. Johnson, and R. E. Tarjan. Performance bounds for level-oriented two-dimensional packing algorithms. *SIAM J. Comput.*, 9(4):808–826, 1980.
- [32] S. Kirkpatrick, D. Gelatt Jr., and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.
- [33] S. Kumar and R. Panneerselvam. A survey on the vehicle routing problem and its variants. *Intelligent Information Management*, 4(3):66–74, 2012.
- [34] W. Leinberger, G. Karypis, and V. Kumar. Multi-capacity bin packing algorithms with applications to job scheduling under multiple constraints. In *Proceedings of the 1999 International Conference on Parallel Processing*, pages 404–, Washington, DC, USA, 1999. IEEE Computer Society.

- [35] K. Li and K.-H. Cheng. On three-dimensional packing. *SIAM J. Comput.*, 19:847–867, 1990.
- [36] A. Lodi, S. Martello, and M. Monaci. Two-dimensional packing problems: A survey. *European Journal of Operational Research*, 141(2):241 – 252, 2002.
- [37] A. Lodi, S. Martello, and D. Vigo. Heuristic and metaheuristic approaches for a class of two-dimensional bin packing problems. *INFORMS J. on Computing*, 11(4):345–357, 1999.
- [38] A. Malapert, C. Guéret, N. Jussien, A. Langevin, and L.-M. Rousseau. Two-dimensional pickup and delivery routing problem with loading constraints. In *First CPAIOR Workshop on Bin Packing and Placement Constraints (BPPC'08)*, 2008.
- [39] S. Martello, D. Pisinger, and D. Vigo. The three-dimensional bin packing problem. *Operations Research*, 48(2):256–267, 2000.
- [40] A. Meir and L. Moser. On packing of squares and cubes. *Journal of Combinatorial Theory Series A*, 5:116–127, 1968.
- [41] S. N. Parragh, K. F. Doerner, and R. F. Hartl. A survey on pickup and delivery problems. *Journal für Betriebswirtschaft*, 58:81–117, 2008.
- [42] R. Roberto S. Peixoto. Algoritmos para problemas de escalonamento em grades. Master's thesis, Universidade Estadual de Campinas, Campinas, Brazil, 2011.
- [43] M. Prais and C. C. Ribeiro. Reactive grasp: An application to a matrix decomposition problem in tdma traffic assignment. *INFORMS J. on Computing*, 12(3):164–176, 2000.
- [44] J. Renaud, F. F. Boctor, and J. Ouenniche. Perturbation heuristics for the pickup and delivery traveling salesman problem. *Computers & Operations Research*, 29:905–916, 2002.
- [45] M. G. C. Resende and C. C. Ribeiro. Greedy randomized adaptive search procedures. In *Handbook of metaheuristics*, pages 219–249. Springer US, 2003.
- [46] G. Scheithauer. Equivalence and dominance for problems of optimal packing of rectangles, 1995.
- [47] J. L. M. da Silveira, E. C. Xavier, and F. K. Miyazawa. A note on a two dimensional knapsack problem with unloading constraints. *RAIRO - Theoretical Informatics and Applications*, eFirst, 8 2013.

- [48] P. Toth and D. Vigo. *The Vehicle Routing Problem*. Society for Industrial & Applied Mathematics, Philadelphia, 1 edition, 2001.
- [49] V. Vazirani. *Approximation Algorithms*. Springer-Verlag, 2001.
- [50] G. Wäscher, H. Haussner, and H. Schumann. An improved typology of cutting and packing problems. *European Journal of Operational Research*, 183(3):1109 – 1130, 2007.
- [51] D. Williamson. Lecture notes on approximation algorithms. Technical Report 21409, T. J. Watson Research Center (IBM Research Division), Michigan State University, Yorktown Heights, New York, 1998.
- [52] E. C. Xavier and F. K. Miyazawa. The class constrained bin packing problem with applications to video-on-demand. *Theoretical Computer Science*, 393(1-3):240–259, mar 2008.
- [53] E. C. Xavier, R. R. S. Peixoto, and J. L. M. da Silveira. Scheduling with task replication on desktop grids: theoretical and experimental analysis. *Journal of Combinatorial Optimization*, To appear.
- [54] E. E. Zachariadis, C. D. Tarantilis, and C. T. Kiranoudis. A guided tabu search for the vehicle routing problem with two-dimensional loading constraints. *European Journal of Operational Research*, 195(3):729 – 743, 2009.