Instituto de Computação
Universidade Estadual de Campinas

Algoritmos de Aproximação para Problemas de Empacotamento em Faixa com Restrições de Descarregamento

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por Jefferson Luiz Moisés da Silveira e aprovada pela Banca Examinadora.

Campinas, 25 de Março de 2011.

Eduardo Candido Xavier (Orientador)

Flávio Keidi Miyazawa (Coorientador)

Dissertação apresentada ao Instituto de Computação, UNICAMP, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP
Bibliotecária: Maria Fabiana Bezerra Müller – CRB8 / 6162

Silveira, Jefferson Luiz Moisés da
Si39a Algoritmos de aproximação para problemas de empacotamento em faixa com restrições de descarregamento/Jefferson Luiz Moisés da Silveira-- Campinas, [S.P. : s.n.], 2011.

Orientador : Eduardo Candido Xavier; Flávio Keidi Miyazawa.
Dissertação (mestrado) - Universidade Estadual de Campinas, Instituto de Computação.

1.Algoritmos. 2. Problema de empacotamento. I. Xavier, Eduardo Candido. II.Miyazawa, Flávio Keidi. III. Universidade Estadual de Campinas. Instituto de Computação. IV. Título.

Título em inglês: Approximation algorithms for the strip packing problem with unloading constraints

Palavras-chave em inglês (Keywords): 1.Algorithms. 2.Packing problem.

Área de concentração: Teoria da Computação

Titulação: Mestre em Ciência da Computação

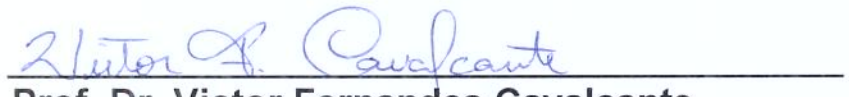
Banca examinadora: Prof. Dr. Eduardo Candido Xavier (IC – UNICAMP)
Prof. Dr. Victor Fernandes Cavalcante (IBM Research – Brasil)
Prof. Dr. Orlando Lee (IC - UNICAMP)

Data da defesa: 25/03/2011

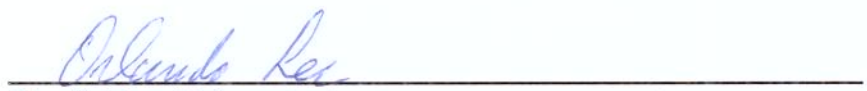
Programa de Pós-Graduação: Mestrado em Ciência da Computação

TERMO DE APROVAÇÃO

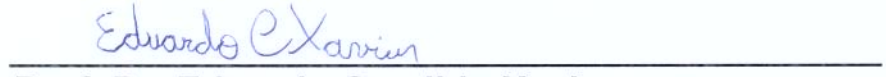
Dissertação Defendida e Aprovada em 25 de março de 2011, pela Banca examinadora composta pelos Professores Doutores:



Prof. Dr. Victor Fernandes Cavalcante
IBM Research Brasil



Prof. Dr. Orlando Lee
IC / UNICAMP



Prof. Dr. Eduardo Candido Xavier
IC / UNICAMP

Instituto de Computação
Universidade Estadual de Campinas

Algoritmos de Aproximação para Problemas de Empacotamento em Faixa com Restrições de Descarregamento

Jefferson Luiz Moisés da Silveira[1]

Março de 2011

Banca Examinadora:

- Eduardo Candido Xavier (Orientador)
- Victor Fernandes Cavalcante
  IBM Research, Brasil
- Orlando Lee
  Instituto de Computação, UNICAMP
- Yoshiko Wakabayashi
  Instituto de Matemática e Estatística, USP (suplente)
- Guilherme Telles
  Instituto de Computação, UNICAMP (suplente)

[1]Suporte financeiro de: Bolsa do CNPq (processo 130123/2009-9) 2009–2011

Resumo

Neste trabalho estudamos problemas de empacotamento com restrições de descarregamento considerados NP-difíceis. Estes problemas possuem aplicações nas áreas de logística e roteamento. Assumindo a hipótese de que $P \neq NP$, sabemos que não existem algoritmos eficientes para resolver tais problemas. Uma das abordagens consideradas para tratar tais problemas é a de algoritmos de aproximação, que são algoritmos eficientes (complexidade de tempo polinomial) e que geram soluções com garantia de qualidade. Estudamos técnicas para o desenvolvimento de algoritmos aproximados e também alguns algoritmos para problemas de empacotamento *online* que podem ser utilizados na resolução do problema estudado. Propomos também algumas heurísticas para o problema e, além disto, provamos que duas destas heurísticas possuem garantias de aproximação com fatores constantes. Realizamos testes computacionais com estes algoritmos propostos. Dentre estes, a heurística *GRASP* foi a que obteve melhores resultados para as instâncias de teste consideradas.

Abstract

In this work we study some NP-hard packing problems with unloading constraints. These problems have applications in logistics and routing problems. Assuming $P \neq NP$, there are no efficient algorithms to solve these problems. On way to deal with these problems is using approximation algorithms, that are efficient algorithms (polynomial time complexity) that produce solutions with quality guarantee. We study techniques used in the development of approximation algorithms and some algorithms for online packing problems which can be used to solve the considered problem. We propose some heuristics for the problem and prove that two of them have constant approximation guarantees. We also perform computational tests with the proposed algorithms. Among them, the *GRASP* heuristic achieved the best results on the considered instances.

Ao meu pai, Joaquim Silveira (*in memoriam*).

Agradecimentos

Esta é, sem dúvida alguma, a parte mais "pessoal" de uma tese/dissertação e, exatamente por este motivo, achei-a a parte mais interessante em boa parte dos trabalhos que já tive o prazer de folhear. Aqui vemos o autor, não os seus resultados. Espero que, ao ler estas poucas palavras, o leitor consiga sentir o quanto devo a cada uma das pessoas citadas, seja profissional ou pessoalmente. São muitas pessoas, então, inevitavelmente, esquecerei alguém.

Em primeiro lugar, gostaria de agradecer ao meu orientador, o professor Eduardo Xavier, pela orientação e apoio no decorrer destes 2 anos no IC. Agradeço principalmente pela sua atenção, dedicação e paciência. Agradeço também ao meu coorientador, Flávio Miyazawa, pelo apoio especial durante os dois semestres em que o Eduardo esteve viajando. Agradeço também por me ensinar Algoritmos Aproximados.

Agradeço a toda minha família pelo apoio, direto e indireto, para a realização deste trabalho. Para ser mais conciso: "mainha", "Tó", "Nay", "Mi", "Tali", Osmar, "Hudson", "Coquinho", "Quinho", Paula e "Victorssauro" ... Esse trabalho também é de vocês. Muito obrigado pelo carinho, amor, recepções, decepções, brincadeiras, gargalhadas e choros.

Serei eternamente grato a Mari, por me incentivar a vir e não me deixar voltar. Esse trabalho é tão meu, quanto dela. Obrigado por tudo, sempre. Eu obviamente, não conseguiria agradecer tanto quanto queria, então agradeço em forma de confissão. Te amo. Agradeço também a toda família da Mari, por sempre me querer e receber tão bem em qualquer ocasião, Severino, Maria Luiza, Marcela, Yuri, "Dudinha" e Davi, muito obrigado.

Agradeço também aos amigos do São Francisco que, mesmo distantes, sempre estiveram próximos em lembranças ou pelo Gtalk, Rodolpho, Alisson, Fagner e Kleber. Aproveito para agradecer as tantas pessoas que conheci aqui em Campinas desde que cheguei, em especial, Roberto, Juliana, Victor, Thaís, Clarissa, dona Lúcia, Hugo, João, Robinho, Rafael, Tiago(s) e Aline.

Agradeço aos professores João Meidanis e Arnaldo Moura por me ensinarem Teoria da Computação e Algoritmos tão bem. Agradeço aos meus professores da graduação, em particular a Mirele e Luis pelas cartas que me ajudaram tanto. Agradeço ao professor

Luis pelos anos de orientação.

Não poderia deixar de agradecer à algumas pessoas que me fizeram esquecer dos meus problemas tantas vezes, John Lennon, Paul McCartney, George Harrison, Ringo Star, Arnaldo Baptista, Chico Buarque e Caetano Veloso. Muito obrigado.

Por fim, agradeço ao povo brasileiro por ter contribuído, mesmo que contra a sua vontade, com recursos financeiros para o desenvolvimento deste trabalho (por consequência agradeço ao CNPq).

Eu falo de ultrapassar a velocidade da luz pra pessoas que nunca ultrapassaram "duzentos por hora", então isso pode ser distante demais...

(Arnaldo Baptista)

Sumário

Lista de Tabelas

Lista de Figuras

Lista de Algoritmos

Lista de Siglas e Abreviações

SPU - *Strip Packing with Unloading constraints*

SPUH - SPU com moviementos horizontais

GRASP - *Greedy Randomized Adaptative Search Procedure*

PTAS - *Polynomial Time Approximation Scheme*

FPTAS - *Fully Polynomial Time Approximation Scheme*

APTAS - *Asymptotic Polynomial Time Approximation Scheme*

AFPTAS - *Asymptotic Fully Polynomial Time Approximation Scheme*

LRC - Lista Restrita de Candidatos

2L-CVRP - *The Vehicle Routing Problem with Two Dimensional Loading Constraints*

NFDH - *Next-Fit Decreasing Height*

FFDH - *First-Fit Decreasing Height*

BL - *Bottom-Left*

ORP - *Online Rectangle Packing*

U_{ORP} - *Unloading Online Rectangle Packing*

OSP - *Online Square Packing*

U_{OSP} - *Unloading Online Square Packing*

LBP - *Level Bin Packing*

BPDO - *Bin Packing Decreasing Order*

SC - *Slot Class*

SO - *Slot Order*

SO_u - SO empacotando como o OSP

SO_{ur} - SO_{ur} com rotações

G - Heurística *GRASP* sem rotações

G_r - Heurística *GRASP* com rotações

lb_3 - Limitante inferior baseado na estrutura dos itens

Capítulo 1

Introdução

Neste trabalho estudamos algoritmos de aproximação e heurísticas para alguns problemas de empacotamento. Em geral, problemas de empacotamento são problemas de otimização que pertencem à classe NP-difícil. Problemas de otimização, na sua forma geral, têm como objetivo maximizar ou minimizar uma função definida sobre um certo domínio. A teoria clássica de otimização é voltada para o caso em que o domínio é infinito. Já os chamados problemas de otimização combinatória tratam do caso em que o domínio é finito e, além disto, em geral é "fácil" listar os seus elementos e também testar se um dado elemento pertence a esse domínio. Ainda assim, a ideia ingênua de listar e testar todos os elementos deste domínio na busca pelo melhor (aquele que maximiza ou minimiza a função objetivo do problema) mostra-se inviável na prática, mesmo para instâncias de tamanho moderado.

Neste trabalho, assumimos a hipótese de que $P \neq NP$. Portanto, não existem algoritmos eficientes para tais problemas de empacotamento que são NP-difíceis. Muitos destes problemas aparecem em aplicações práticas e a busca por bons algoritmos para estes é fortemente motivada por razões econômicas. Problemas de empacotamento aparecem no arranjo de caixas em caminhões, problemas de corte de insumos (metais, tecidos e vidros por exemplo) em indústrias, alguns problemas específicos de escalonamento, entre outros. Neste trabalho consideramos problemas de empacotamento de itens em uma única faixa de tal forma a maximizar a área da faixa preenchida pelos itens, ou seja, minimizar a altura total do empacotamento. Este problema é conhecido como empacotamento em faixa ou *Strip Packing Problem*. Consideramos também casos em que os itens podem estar sujeitos a vários tipos de restrições, como por exemplo a orientação fixa (impossibilidade de rotacionar os itens), caso *online*, restrição de *gravidade*, dentre outras. Uma outra restrição que pode ser considerada é a *restrição de descarregamento*. Neste caso, além das informações de largura e altura, cada item possui uma ordem (classe) e o empacotamento final deve ser tal que pode-se remover itens em ordem sem movimentar outros itens e usando apenas a área livre da faixa.

Dada a impossibilidade de resolver tais problemas de maneira exata e eficiente, busca-se alternativas que sejam úteis. Na prática vários métodos são utilizados como programação inteira, heurísticas e meta-heurísticas, algoritmos de aproximação (algoritmos aproximados), dentre outros. Boa parte destes métodos sacrificam a otimalidade em troca da garantia de uma solução aproximada computável eficientemente. Dentre os métodos citados, destacamos os algoritmos de aproximação, cujo interesse é, apesar de sacrificar a otimalidade, fazê-lo de forma que ainda possamos dar boas garantias sobre o valor da solução obtida, procurando ganhar o máximo em termos de eficiência computacional.

Em linhas gerais, algoritmos aproximados são aqueles que possuem complexidade polinomial e que não necessariamente produzem uma solução ótima, mas soluções que estão dentro de um certo fator da solução ótima. Como esta garantia deve ser satisfeita para todas as instâncias do problema, devemos dar uma demonstração formal deste fato.

1.1 Objetivos do Trabalho

O objetivo principal deste trabalho é estudar técnicas usadas no desenvolvimento de algoritmos aproximados e aplicá-las ao problema específico de empacotamento chamado *Strip Packing Problem with Unloading Constraints* (SPU). Buscamos desenvolver heurísticas para o SPU e provar fatores de aproximação para estas. A fim de demonstrar a viabilidade dos algoritmos propostos, realizamos diversos experimentos computacionais com instâncias tanto novas como obtidas na literatura.

1.2 Organização do Texto

Primeiramente, damos uma breve introdução a algoritmos aproximados e falamos sobre a meta-heurística *GRASP* (Cap. 2). Neste mesmo Capítulo, apresentaremos as notações utilizadas e definiremos formalmente os problemas SPU e SPUH (uma versão mais relaxada que veremos adiante), além de descrever os principais resultados relacionados aos problemas atacados neste trabalho.

No Capítulo 3, apresentamos algoritmos de aproximação para dois problemas de empacotamento *online*. Ao final da análise de cada algoritmo apresentamos, caso existam, ideias de como aplicá-lo diretamente aos problemas SPU ou SPUH , ou quais ideias são interessantes para serem utilizadas. Estudamos problemas de empacotamento em faixa *online* com restrições *Tetris*. Nestes caso os itens são apresentados de maneira *online* e, assim como no conhecido jogo *Tetris*, devem percorrer um caminho livre na faixa até encontrar a sua posição final de empacotamento. Os algoritmos estudados para este problema possuem características muito interessantes para serem aproveitadas no problema

em questão. Primeiramente estudamos um algoritmo para a versão geral do problema que considera que os itens podem ser rotacionados e depois um algoritmo para o caso específico onde os itens são quadrados.

É importante lembrar que existem inúmeros algoritmos aproximados para problemas de empacotamento *online* e não é nossa intenção mostrar todos estes algoritmos. Neste trabalho buscamos estudar os algoritmos com técnicas mais interessantes que possam ser usadas no problema SPU.

No Capítulo 4 apresentamos as heurísticas originais propostas para o SPU. Primeiramente apresentamos um novo algoritmo 6.75-aproximado para o problema. Após isto, apresentamos um algoritmo 2-aproximado para o caso específico onde instâncias são tais que o número de classes dos itens é constante. Por fim, apresentamos uma heurística *GRASP* para o SPU que é fortemente baseada no algoritmo que possui o melhor resultado prático para o *Strip Packing Problem* sem rotações.

Em seguida, são apresentadas as instâncias utilizadas e os experimentos computacionais realizados com os algoritmos (Cap. 5).

No Capítulo 6 são apresentados tanto um resumo dos estudos dos algoritmos aproximados quanto os resultados dos experimentos realizados com os algoritmos propostos.

Por fim, no Capítulo 7 são apresentadas as conclusões deste trabalho.

Capítulo 2

Fundamentação Teórica

Neste Capítulo apresentamos, de maneira resumida, definições e noções básicas que serão utilizadas no decorrer do trabalho. Definimos os conceitos básicos sobre algoritmos aproximados, dando definições relacionadas ao tema e discutindo brevemente algumas técnicas que serão utilizadas. Introduzimos também a meta-heurística *GRASP* (*greedy randomized adaptative search procedure*), ressaltando suas ideias fundamentais. Também apresentamos os problemas de empacotamento considerados neste trabalho, definindo-os formalmente, além de definir também a notação que será utilizada no trabalho. Por fim, descrevemos os principais resultados da literatura relacionados aos problemas abordados neste trabalho.

2.1 Algoritmos de Aproximação

Nesta Seção introduzimos brevemente o tema algoritmos de aproximação apresentando a notação utilizada e também conceitos básicos sobre o tema. Nas definições a seguir, consideramos sempre problemas de minimização, lembrando que as mesmas definições podem ser adaptadas para o caso de problemas de maximização.

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

Dois passos básicos devem ser seguidos no desenvolvimento de um algoritmo aproximado $\mathcal{A}$. Primeiramente, deve-se demonstrar que $\mathcal{A}$ possui complexidade de tempo polinomial. Em seguida, para o caso não assintótico, buscar uma prova do seu fator de aproximação (α), ou seja, demonstrar que para qualquer instância do problema em questão vale que $\mathcal{A}(I) \leq \alpha OPT(I)$. Outro aspecto interessante é verificar se o fator α demonstrado é o melhor possível. Para tanto, devemos apresentar pelo menos uma instância onde vale que $\mathcal{A}(I)/OPT(I) = \alpha$ ou $\mathcal{A}(I)/OPT(I) = \alpha + \epsilon$ (para ϵ tão próximo de zero quanto se queira). Neste caso, o fator de aproximação do algoritmo não pode ser melhorado e é considerado justo.

Algoritmos aproximados são geralmente 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 de caráter geral 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 [25, 44, 20, 41]).

Programação linear tem sido usado 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, chamaremo-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 α (fator de inaproximabilidade). Nestes casos, dado um problema P, devemos demonstrar que não pode existir um algoritmo α-aproximado para P. Uma das formas de fazê-lo é através de reduções, demonstrando que caso exista um algoritmo polinomial α-aproximado para P, então podemos resolver algum problema NP-difícil em tempo polinomial. Para mais detalhes em inaproximabilidade veja [2, 41].

Para determinados problemas ditos *online*, não é possível conhecer toda a instância I de entrada a priori. Nestes casos 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 em algoritmos online veja [10].

Um algoritmo $\mathcal{A}$ é dito ser α-competitivo se é online e é α-aproximado. Nestes casos o valor $\mathcal{A}(I)$ é comparado com o valor da solução ótima offline $OPT(I)$, afim de obtermos uma aproximação.

2.2 Meta-heurística *GRASP*

Nesta Seção descreveremos as principais ideias da meta-heurística *GRASP* que serão utilizadas em alguns dos algoritmos propostos neste trabalho.

Assim como os algoritmos de aproximação, as heurísticas são algoritmos utilizados para obter soluções aproximadas para problemas computacionalmente difíceis, porém, neste caso, pode não haver garantias formais da qualidade da solução. As meta-heurísticas são procedimentos genéricos utilizados para guiar o desenvolvimento 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 meta-heurísticas mais conhecidas podemos citar os algoritmos genéticos, a *Busca Tabu*, *simulated annealing*, *GRASP* e colônias de formigas.

A meta-heurística *GRASP* foi proposta por Feo *et al.* [18], mais formalmente em [19]. Esta meta-heurística 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 a característica de múltiplos pontos iniciais (*multistart*) do *GRASP*.

Nos Algoritmos 2.1, 2.2 e 2.3 são apresentados pseudocódigos que sintetizam a meta-heurística *GRASP* e suas fases.

Algoritmo 2.1 Fase-Construtiva

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

Algoritmo 2.2 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**

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). Na literatura existem alguns exemplos de aplicação da heurística GRASP em problemas de empacotamento com bons resultados, como [1] e [8]. Neste trabalho esta meta-heurística foi escolhida por possuir bons resultados

Algoritmo 2.3 GRASP

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

para a versão clássica do *Strip Packing* utilizando faixas verticais em sua estrutura de empacotamento.

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 LCR. Caso LCR 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 ajudar 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* [38], pois, ajusta até os seus parâmetros durante a sua execução, com base nos valores das soluções encontradas. Para o problema de empacotamento em faixa bidimensional, foi proposta uma heurística *GRASP reativa* [1] com resultados superiores as heurísticas *GRASP* comuns.

Outra técnica bastante utilizada em conjunto com o *GRASP* é a conexão de caminhos (*Path-relinking*). Esta técnica foi proposta por Glover [23] para buscar melhores soluções no “caminho” entre boas soluções encontradas com *Busca Tabu* e *Scatter Search*. Esta técnica busca mesclar diferentes características das melhores soluções encontradas pelo *GRASP* gerando possivelmente, soluções melhores.

2.3 Problemas de Empacotamento

Nesta seção apresentamos algumas definições sobre problemas de empacotamento e definiremos as notações que serão utilizadas no restante do trabalho.

Nos problemas de empacotamento temos um ou mais objetos grandes n-dimensionais, os quais chamamos de *recipientes*, e vários objetos menores também n-dimensionais os quais chamamos de *itens*. O nosso objetivo é empacotar itens dentro de recipientes, de forma a maximizar ou minimizar uma dada função objetivo. Tanto os itens quanto os recipientes podem assumir formas regulares ou irregulares. Formas regulares são formas retangulares e irregulares são qualquer forma, ou modelo (retângulos, esferas, formas quaisquer etc.). Além disto, o empacotamento dos itens pode ser submetido a 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 recipiente e do problema sejam respeitadas.

Uma tipologia para vários tipos de problemas de empacotamento foi feita por Dyckhoff [16] e mais recentemente uma nova tipologia foi proposta por Wäscher *et al.* [43].

Definiremos agora três problemas de empacotamento básicos.

O primeiro problema, chamado *bin packing*, tem como entrada uma lista de itens $I = (a_1, \ldots, a_m)$, cada item com tamanho $s(a_i)$, e um número B que indica o tamanho dos recipientes que podem ser utilizados para empacotar os itens. Assumimos que para todo item $a_i \in I$, vale que $s(a_i) \leq B$. Este problema consiste em empacotar todos os itens de I no menor número possível de recipientes, ou seja, devemos achar uma partição $P_1, \ldots, P_q$ de I tal que q seja mínimo e $\sum_{a_i \in P_j} s(a_i) \leq B$, para cada parte P_j.

Existem versões onde os itens e recipientes podem ter mais dimensões, como *bin packing* bidimensional, tridimensional, etc. Nestes casos, cada recipiente ou item possui tamanho dado por uma tupla, onde cada componente indica seu tamanho na correspondente dimensão.

Há versões em que cada item $a_i \in I$ possui uma multiplicidade d_i. Neste caso devemos gerar um empacotamento que contém d_i itens do tipo a_i, $i = 1, \ldots, m$. Os problemas com multiplicidade são conhecidos na literatura como *cutting stock*.

O segundo problema é conhecido como *strip packing*. Na versão bidimensional deste problema temos uma lista de itens bidimensionais $I = (a_1, \ldots, a_m)$, cada item a_i com tamanho $(x(a_i), y(a_i))$, 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 terceiro problema é conhecido como *knapsack*, ou problema da mochila. Neste caso temos apenas um recipiente de tamanho B, e uma lista de itens $I = (a_1, \ldots, a_m)$,

cada item a_i com tamanho $s(a_i)$ e valor $p(a_i)$, $i = 1, \ldots, m$. O objetivo do problema é empacotar um subconjunto dos itens de I em um recipiente de tamanho B de tal forma que a soma dos valores destes itens empacotados seja maximizada. Também podemos considerar as versões multidimensionais deste problema.

O problema, como foi definido, é conhecido como restrito, ou mochila 0/1. Neste caso, cada item pode ser empacotado apenas uma vez. Na versão não-restrita, um item $a_i \in I$ pode ser empacotado várias vezes, ou seja, várias cópias do mesmo item.

Diversos 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.

2.3.1 O Problema Strip Packing com Restrições de descarregamento (SPU)

Neste trabalho estamos interessados no problema *strip packing* bidimensional com itens retangulares e restrições de descarregamento.

O problema básico de strip packing consiste em determinar um arranjo para itens em um recipiente (ambos n-dimensionais), sem haver sobreposição entre os itens minimizando a altura (a única dimensão variável) induzida pelo empacotamento. Porém, em determinadas aplicações, a modelagem do problema exige que os itens estejam dispostos de maneira ordenada no recipiente.

Considere, por exemplo, o seguinte problema chamado 2L-CVRP (*The Vehicle Routing Problem with Two Dimensional Loading Constraints*) [29, 22, 45, 21]. 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 (recipientes) que estão inicialmente situados no fornecedor dos produtos. Para acomodar os itens nos recipientes é utilizado um algoritmo de empacotamento com restrição de descarregamento, para garantir que, durante a entrega de produtos de cada cliente, não haja nenhum produto de outros clientes bloqueando a saída do recipiente.

Neste caso, os itens devem ser retirados do recipiente em uma ordem preestabelecida e, além disto, esta remoção ocorre por apenas uma das extremidades do recipiente. Portanto, o empacotamento deve garantir que não seja necessário alterar a organização dos itens no recipiente durante a remoção de um item, ou seja, garantir que exista um caminho livre para o item percorrer entre a sua posição e a saída do recipiente no momento de sua remoção.

Na Figura 2.1 é possível ver dois exemplos de empacotamento para este problema. Na parte (a) é apresentado uma sequência de clientes que serão visitados para entrega

de produtos (A-B-C). Em (b) é apresentado um empacotamento inválido, pois não há como remover todos os itens de A (o primeiro cliente) sem movimentar os outros itens do recipiente, considerando que a saída é realizada pelo topo da faixa. Por fim, em (c) temos um empacotamento válido para o SPU.

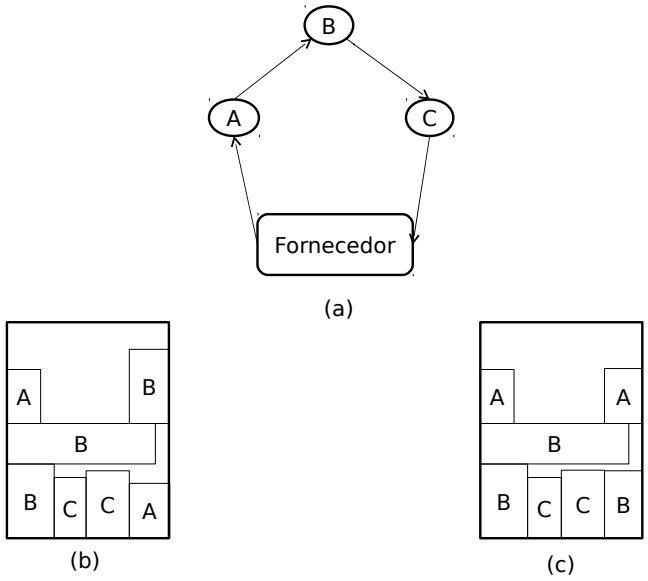


Figura 2.1: O problema SPU.

Chamamos este problema de strip packing com restrições de descarregamento e podemos defini-lo formalmente da seguinte maneira: Dadas uma faixa S de largura L e altura infinita, e uma lista $I = (a_1, \ldots, a_n)$, onde $a_i = (h(a_i), w(a_i), c(a_i))$, $i = 1, \ldots, n$, onde $h(a_i)$, $w(a_i)$ e $c(a_i)$ são, respectivamente, a altura, largura e ordem (classe) do item a_i. Considere que a faixa (recipiente) S está com seu canto inferior esquerdo na *origem* do plano cartesiano, então um empacotamento pode ser definido como uma função $f : I \to R^2$ que posiciona cada item com seu canto inferior esquerdo na posição $(x(a_i), y(a_i))$ deste plano. O objetivo é encontrar um empacotamento que minimize $max_i\{y(a_i) + h(a_i)\}$ e que respeite as seguintes restrições:

- Todos os itens devem estar completamente contidos em S, ou seja,

$$0 \leq x(a_i) \leq L - w(a_i), 0 \leq y(a_i) \quad \forall a_i \in I.$$

- Os itens não podem estar sobrepostos, ou seja,

$$x(a_i) + w(a_i) \leq x(a_j) \quad \texttt{ou} \quad x(a_j) + w(a_j) \leq x(a_i) \quad \texttt{ou}$$
$$y(a_i) + h(a_i) \leq y(a_j) \quad \texttt{ou} \quad y(a_j) + h(a_j) \leq y(a_i) \quad \forall a_i, a_j \in I,\ i \neq j.$$

- Todos os itens devem satisfazer a restrição de descarregamento (Veja a Figura 2.2), ou seja,

$$x(a_i) + w(a_i) \leq x(a_j) \quad \texttt{ou} \quad x(a_j) + w(a_j) \leq x(a_i) \quad \texttt{ou} \quad y(a_i) + h(a_i) \leq y(a_j)$$
$$\forall a_i, a_j \in I, \text{ onde } c(a_i) > c(a_j).$$

Na Figura 2.2 são apresentados dois empacotamentos. Assuma que $c(a_j) > c(a_i)$. Na parte (a) o empacotamento não é uma solução válida para o problema SPU, porque a_j está bloqueando a_i. Por outro lado, em (b) temos um empacotamento válido para o problema SPU, onde os itens podem ser removidos em ordem (1 e 2).

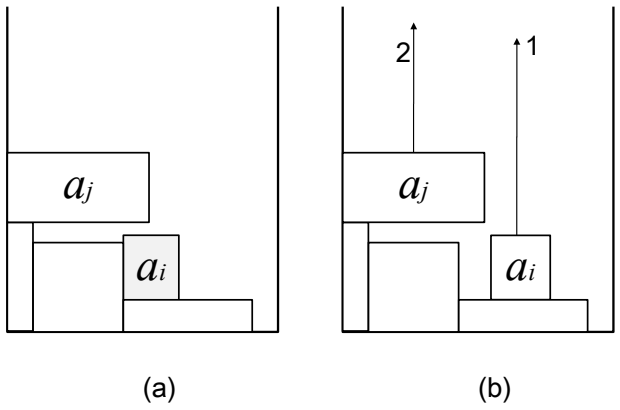


Figura 2.2: Exemplos de soluções para o SPU

Uma versão relaxada do SPU também será discutida em Seções posteriores. Nesta versão, denotada por SPUH, a restrição de remoção é relaxada, permitindo que o item também possa ser movimentado horizontalmente para ser removido do recipiente. Na Figura 2.3 são apresentadas possíveis soluções pra o SPUH. Assuma que $c(a_j) > c(a_k) > c(a_i)$. Na parte (a) o empacotamento não é uma solução válida para o problema SPUH, porque a_j está bloqueando a_i. Por outro lado, em (b) Um empacotamento válido para o problema SPUH, onde os itens podem ser removidos em ordem (1, 2 e 3). Note que a solução em (b) é inviável para o problema SPU.

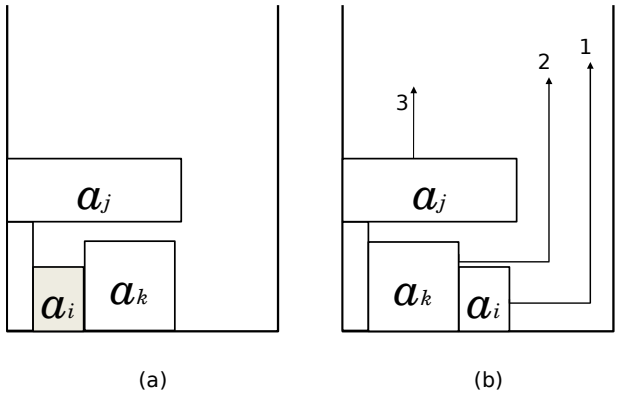


Figura 2.3: Exemplos de soluções para o SPUH

Os problemas SPU e SPUH são generalizações do clássico *Strip Packing*, pois se todos os itens pertencerem a mesma classe (apenas um cliente na rota), teremos exatamente a versão clássica do *Strip Packing* bidimensional. Portanto, estes problemas são NP-difíceis no sentido forte.

2.4 Revisão da Bibliografia

Nesta Seção descreveremos resumidamente os principais resultados de aproximação para o *Strip Packing*, bem como além de algumas heurísticas que contextualizam o problema atacado.

Em 1980, Coffman *et al.* [33] estenderam algoritmos aproximados clássicos para o *Bin Packing* unidimensional para o *Strip Packing* bidimensional. Neste trabalho eles introduziram dois algoritmos clássicos para o problema: o *Next-Fit Decreasing Height* (NFDH) e o *First-Fit Decreasing Height* (FFDH). Ambos os algoritmos são *offline*, pois assumem que os itens estão ordenados em ordem não crescente de altura. Os algoritmos funcionam de maneira semelhante: o NFDH empacota o próximo item da lista ordenada no nível (*level*) atual alinhado a base do nível e a esquerda enquanto houver espaço (o primeiro nível é a base de S). Caso o item não caiba no nível atual, este é "desativado" (não receberá mais itens) e um novo nível é criado acima do atual (como se a nova base de S fosse uma linha horizontal acima do item mais alto do nível atual) e o item é empacotado

alinhado a esquerda na base do novo nível. Enquanto o NFDH fecha os níveis e não os utiliza mais, o FFDH, por outro lado, mantém os níveis abertos, para que o próximo item seja empacotado no nível mais baixo em que caiba. Assim como no NFDH, no FFDH caso um item não caiba em nenhum nível aberto, um novo nível é criado acima de todos os outros e este novo item é empacotado nele.

Tanto o NFDH quanto o FFDH podem ser implementados em $O(n \log n)$ utilizando as estruturas de dados apresentadas em [32] para os respectivos casos unidimensionais do *Bin Packing*. Coffman demonstrou que $NFDH(I) \leq 2OPT(I) + 1$ e que $FFDH(I) \leq 1.7OPT(I) + 1$, além disto, mostrou que estes fatores de aproximação não podem ser melhorados, ou seja, que os algoritmos são justos, se desconsiderarmos as constantes aditivas.

Fugindo da ideia de níveis, em 1980 também Baker *et al.* [4] propuseram um algoritmo clássico chamado *Bottom-Left* (BL), para o qual vale que $BL(I) \leq 3OPT(I)$, e ainda mais, este fator é justo. Este algoritmo funciona da seguinte maneira: Primeiramente os itens são ordenados em ordem não crescente de largura e, em ordem, os itens são empacotados na posição mais baixa possível, alinhado a esquerda. Implementações comuns do algoritmo BL possuem complexidade de tempo $O(n^3)$, porém, Chazelle [13] apresentou uma implementação elegante com complexidade $O(n^2)$.

Estes três algoritmos supracitados têm sido reutilizados como rotinas em algoritmos para problemas diversos de empacotamento. Para os problemas atacados neste trabalho, eles podem servir como rotinas de empacotamento de cada classe de itens, porém fica difícil garantir a viabilidade do problema (exceto no caso do NFDH) dado que não há garantias das posições dos itens na faixa. Além disto, ainda há a necessidade de ordenação, que não tem relações com a ordem do item.

Com abordagens diferentes, Steinberg [40], Schiermeyer [39], propuseram algoritmos que são 2-aproximações absolutas para o *Strip Packing*. Schiermeyer [39] propôs um algoritmo chamado *Reverse-Fit* que também é baseado em níveis, porém com uma diferença básica de que as faixas podem não ser disjuntas. Steinberg [40], por sua vez, definiu sete procedimentos, cada qual divide o problema original (tratável) em outros menores e os resolve recursivamente, gerando novos problemas tratáveis.

Estes dois resultados, são de difícil adaptação para os problemas atacados neste trabalho devido a ordenação dos itens que se faz necessária e ao alto emaranhamento dos itens nas soluções.

Em 2000, Kenynon *et al.* [35] propuseram um AFPTAS para o *Strip Packing*. A solução para uma dada instância é obtida resolvendo-se um programa linear para um outro problema conhecido como *Fractional Strip Packing* [34]. Depois disso são apresentadas técnicas para se obter uma solução para o problema original. Com isso, eles mostraram que o algoritmo é uma aproximação de $\mathcal{A}(I) \leq (1+\varepsilon)OPT(I)+O(\frac{1}{\varepsilon^2})$, com complexidade

de tempo polinomial em n e em $\frac{1}{\varepsilon}$. Em 2005, Jansen *et al.* [31] propuseram um AFPTAS para o caso com rotações. Seu método é baseado no algoritmo proposto por Kenynon *et al.* [35] para o caso sem rotações e alcança a mesma aproximação. Recentemente, Jansen [30] propuseram um APTAS para o caso sem rotações, com a vantagem de possuir um fator aditivo de 1, ao invés de $O(\frac{1}{\varepsilon^2})$.

Embora estes esquemas de aproximação apresentem os melhores resultados teóricos para o *Strip Packing*, eles são de difícil adaptação para os casos com restrição de ordem, principalmente devido as reduções e ordenações utilizadas.

Para o caso *online*, os primeiros resultados são creditados a Baker *et al.* [5]. Um destes resultados foi provado para o algoritmo *First Fit Shelf*, o qual possui fator de aproximação assintótico de 1.7. Em 1997, Csirik *et al.* [15] mostraram que nenhum algoritmo baseado em níveis pode alcançar um fator de aproximação assintótico melhor que $H_\infty = 1.6913\ldots$ e, além disto, apresentaram um algoritmo baseado na série harmônica que pode se aproximar arbitrariamente deste fator. Recentemente, Han *et al.* [24] utilizaram a relação entre o *strip packing* e o *bin packing* unidimensional para mostrar que o *strip packing* admite um algoritmo *online* com fator de aproximação assintótico de 1.5888.

Além destas versões clássicas do problema, várias outras vêm sendo estudadas. Dentre estas, duas versões do problema, que consideram as restrições do famoso jogo *Tetris*, aparecem como boas estratégias para conseguir aproximações para os problemas SPU e SPUH.

Em 1997, Azar *et al.*[3] estudaram o problema de *Strip Packing Online* com restrições *Tetris*. Este problema pode ser informalmente definido da seguinte maneira: Dados n itens, apresentados de maneira *online*, devemos encontrar um empacotamento de altura induzida mínima, de maneira que um item só pode ser empacotado numa posição que seja "alcançável" por ele a partir do topo da faixa F, no momento em que ele for apresentado na entrada. O termo "alcançável" indica que o item deve ter um caminho livre para percorrer na faixa até encontrar sua posição final de empacotamento, antes que outro item seja apresentado na entrada. Uma vez empacotado, um item não pode mais ser movimentado. é importante ressaltar que apesar do seu trabalho citar as regras *Tetris*, o fator *gravidade* não é levado em consideração, ou seja, um item pode ser empacotado sem necessariamente estar suportado por outros itens. Em seu trabalho, eles estudaram os casos com orientação fixa e com rotações. Para o caso com rotações propuseram um algoritmo 4-competitivo baseado em sub-faixas horizontais, resultado este que permanece como o melhor da literatura até os dias atuais. Os autores também provaram um limitante inferior de $\Omega(\sqrt{\log \frac{1}{\varepsilon}})$ para o caso com orientação fixa, onde ε é um limitante inferior na largura dos itens. Além disto, para o caso com orientação fixa, apresentaram um algoritmo com fator de aproximação $O(\log \frac{1}{\varepsilon})$. Estes problemas estão intimamente relacionados ao SPUH e suas ideias podem ser reaproveitadas para ambos os problemas abordados neste

trabalho.

Em 2009, Fekete *et al.* [17], estudaram o mesmo problema descrito em [3], porém com duas modificações importantes: A restrição de *gravidade* é considerada e os itens são quadrados. Para esta versão do problema, eles propuseram um algoritmo 2.6154-competitivo baseado em *slots* (sub-faixas verticais). Este problema, por sua vez, está relacionado ao SPU e suas ideias podem ser diretamente aplicadas a ambos os problemas abordados neste trabalho, gerando algoritmos aproximados para estas versões restritas dos problemas SPU e SPUH.

Além destes algoritmos, que podem ser reaproveitados, algumas heurísticas já foram utilizadas para resolver o SPU. Estas são utilizadas apenas como parte do problema 2L-CVRP, preocupando-se apenas em verificar a viabilidade do empacotamento de um conjunto de itens. Na verdade, são algoritmos que buscam verificar se um certo conjunto de classes de itens (todos os itens dos clientes em uma rota de entrega) podem ser empacotados em um bin de tamanho fixo, respeitando a restrição de ordem. Nestes trabalhos o problema assemelha-se mais ao *Knapsack* ou ao *Bin Packing* e, mesmo assim, resultados específicos de empacotamento não são reportados (exceto em [22] que mostrou a ocupação média dos veículos (bins) utilizados no roteamento).

Em [29] Iori *et al.* propuseram um algoritmo exato para o problema 2L-CVRP. Seu algoritmo de empacotamento é uma heurística simples baseada no *bottom-left* e um procedimento *branch-and-bound* para checar a viabilidade dos empacotamentos. Esta solução resolveu instâncias com, no máximo 25 clientes e 91 itens em, aproximadamente 24h de tempo de CPU. Nenhum resultado específico para o algoritmo de empacotamento foi fornecido.

Gendreau *et al.* [22] propuseram uma heurística de *Busca Tabu* para o 2L-CVRP. O problema do carregamento foi resolvido utilizando-se alguns limitantes inferiores, heurísticas, busca-local e um *branch-and-bound* com tempo limitado. O algoritmo de empacotamento proposto utiliza iterativamente um procedimento baseado no algoritmo *Touching Perimeter* [36] para o *Bin Packing* bidimensional e também para o *Strip Packing* [28]. Na primeira chamada do algoritmo, os itens estão ordenados em ordem reversa de visita aos clientes (classe) e em cada chamada subsequente esta ordem trivial é alterada. Os autores reportaram algumas informações sobre o empacotamento produzido por esta estratégia. De fato, foi reportada a porcentagem de área ocupada em cada veículo. Porém este resultado não mede, de fato, a qualidade do empacotamento, já que empacotamentos muito ruins podem satisfazer a entrega em uma rota e, além disto, possíveis problemas de empacotamento entre itens de classes diferentes podem ser evitados dividindo as classes em diferentes veículos. Portanto, não é possível comparar diretamente nossos resultados com os fornecidos pelos autores.

Em [45] Zachariadis *et al.* propuseram uma heurística de *Busca Tabu* guiada para o 2L-

CVRP. Quanto ao problema de empacotamento, foram utilizadas 5 heurísticas diferentes (em ordem): As duas primeiras heurísticas simples são baseadas no *bottom-left*. A terceira e a quarta heurísticas são similares àquela utilizada por Gendreau *et al.* [22], baseada na heurística *Touching Perimeter*. A quinta, e última, heurística tenta obter um alto nível de ocupação do veículo escolhendo sempre a posição que minimiza o desperdício de área em um conjunto de espaços livres para empacotamento. Neste trabalho, os autores não mencionam resultados específicos para estas heurísticas de empacotamento.

Os melhores resultados práticos para o 2L-CVRP encontrados na literatura são creditados a uma heurística *Ant Colony Optimization*, proposta por Doerner *et al.* [21]. O algoritmo primeiramente usa alguns limitantes para o *Bin Packing* com o objetivo de provar a inviabilidade de algumas rotas. Então, se os limitantes não provarem isto, o algoritmo começa a busca por empacotamentos viáveis para esta rota. As heurísticas utilizadas são similares àquelas previamente citadas em [22] e [45] (*Bottom-left* e *Touching Perimeter*) com a adição de um algoritmo *branch-and-bound* com tempo de CPU limitado. Também neste trabalho, não foi fornecida nenhuma informação específica sobre a qualidade do empacotamento.

Se levarmos em consideração apenas resultados práticos com heurísticas para o *Strip Packing* (desconsiderando resultados de aproximação), há um algoritmo promissor para ser reutilizado no problema SPU. Em 2008, Alvarez *et al.* [1] propuseram uma heurística *GRASP reativa* para o problema *Strip Packing* com orientação fixa. A fase construtiva da heurística é guiada por uma escolha semi-gulosa baseada na largura dos itens e dos espaços livres no empacotamento atual. Desta forma, o algoritmo gera uma LRC com os melhores itens ainda não empacotados e seleciona aleatoriamente um destes para ser empacotado na solução atual. Esta escolha pode ser modificada caso algumas das estimativas usadas indique que é mais vantajoso empacotar o item mais alto ainda não empacotado. Então, este procedimento é repetido enquanto ainda há algum item não empacotado. A fase de busca local é simples, pois apenas remove e reempacota uma porcentagem dos itens na parte superior do empacotamento, utilizando para isto, o próprio algoritmo construtivo, porém determinístico. Este trabalho contém exaustivos experimentos computacionais e possui os melhores resultados da literatura para o *Strip Packing* com orientação fixa. A estratégia utilizada na fase construtiva é diretamente aplicável ao problema SPU, pois é baseada em faixas verticais para empacotar os itens, diferentemente de outras heurísticas da literatura.

Capítulo 3

Algoritmos Aproximados

Neste capítulo apresentamos os principais trabalhos na área de empacotamento *online* com restrições *Tetris*. Como já citado anteriormente, estes trabalhos estão intimamente relacionados aos problemas SPU e SPUH devido as restrições impostas na forma do item chegar a sua posição de empacotamento. Após a apresentação de cada resultado formalizaremos esta relação entre o resultado da literatura e o problema que estamos estudando.

3.1 Uma 4-Aproximação com Restrições Tetris

Nesta seção analisaremos o algoritmo aproximado proposto por Azar *et al.* [3] para resolver o problema *Strip Packing Online* com restrições *Tetris*. Neste problema, o caminho que um item percorre, do topo da faixa S até alcançar sua posição final no empacotamento, é considerado. Isto difere do modelo clássico de empacotamento, onde os itens podem ser empacotados em qualquer posição livre no recipiente. A Figura 3.1 apresenta o caminho que um item percorre para chegar a sua posição e um item bloqueado neste modelo, nas partes (a) e (b) respectivamente.

O algoritmo *Online Rectangle Packing* - ORP (Algoritmo 3.1) apresentado a seguir, permite rotações ortogonais (em 90°) nos itens. Porém, uma vez empacotado o item não pode mais ser movimentado ou rotacionado dentro do recipiente. Assume-se que a altura e a largura de cada item é limitada por 1. Além disto, sem perda de generalidade, assumimos que a largura da faixa S vale 1.

Sejam $0 < \alpha < 1$ e $0 < W < \frac{1}{2}$ duas constantes. Cada item $a_k \in I$ pode ser de um dos seguintes tipos: *NonBuffers*, que possuem $w(a_i) < W$ e *Buffers*, caso contrário. Além disto, *Buffers* dividem-se em *SmallBuffers*, onde $W \leq w(a_k) < 1 - W$, e *LargeBuffers*, caso $1 - W \leq w(a_k) \leq 1$. O recipiente S é dividido em infinitas faixas horizontais F_i com a altura $h(F_i) = \alpha^i$, $i \geq 0$. Seja F uma faixa, defina $w(F)$ a sua largura ocupada pelos itens empacotados em F (a soma das larguras dos itens em F). Cada faixa F_i será

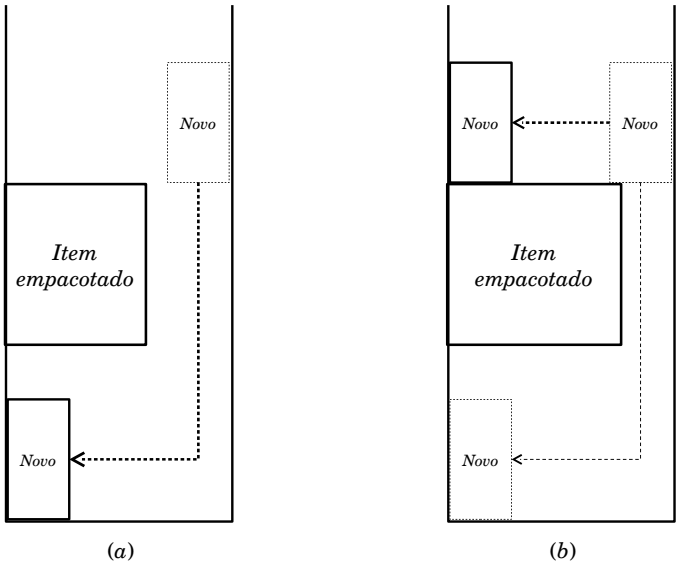


Figura 3.1: Movimentos permitidos no modelo utilizado em [3].

utilizada para empacotar os retângulos $a_k \in I$ de altura $\alpha^{i+1} < h(a_k) \leq \alpha^i$.

O algoritmo ORP opera da seguinte maneira: Ao chegar um novo item a_k na entrada, ele rotaciona-o, caso necessário, para que seu menor lado seja a sua base. Depois disto, classifica-o como um *NonBuffer* ou *Buffer*. Se for um *NonBuffer* o algoritmo busca a faixa F_i atualmente aberta para itens de altura $h(a_k)$. Se esta faixa não existir, não for alcançável ou se empacotar a_k em F_i faz com que $w(F_i) > 1 - W$, então é criada uma nova faixa, acima de todas existentes, para a_k e esta nova faixa receberá novos itens de F_i. Senão, a_k é simplesmente empacotado na faixa F_i. Por outro lado, se a_k é um *Buffer*, então uma nova faixa de altura $h(a_k)$ é criada acima de todas as outras existentes para acomodar apenas o item a_k.

3.1.1 Análise do Algoritmo ORP

Nesta Seção discutiremos sobre a complexidade, corretude e aproximação do algoritmo ORP.

Primeiramente, é fácil ver que ORP possui complexidade de tempo polinomial. Para cada *Buffer* o algoritmo simplesmente o empacota acima dos outros, operação esta, que pode ser realizada em tempo constante. Para *NonBuffers* a_k, o algoritmo busca uma faixa aberta alcançável para empacotá-lo. Esta operação pode ser feita verificando se algum *LargeBuffer* B, com $w(B) > 1 - w(a_k)$, está localizado acima da faixa aberta,

Algoritmo 3.1 *Online Rectangle Packing* - ORP

```
1: input: Lista I de itens
2: begin
3: while I ≠ ∅ do
4:    a_k ← primeiro(I).
5:    Rotacione a_k de forma que h(a_k) ≥ w(a_k).
6:    if (w(a_k) < W) then
7:       i ← ⌊log_α h(a_k)⌋.
8:       if (∄ faixa F_i alcançável or w(F_i) + w(a_k) > 1 − W) then
9:          Feche uma possível faixa F_i aberta.
10:         Crie uma nova faixa F_i acima de todas as outras.
11:      Empacote a_k na faixa F_i aberta o mais a esquerda e baixo possível.
12:   else
13:      Crie uma nova faixa B acima de todas as outras, com h(B) = h(a_k).
14:      Empacote a_k em B o mais a esquerda possível
15:   I ← I\a_k.
16: return altura total da faixa;
17: End
```

caso esta última exista. De fato, é possível olhar apenas para os *LargeBuffers* pois as faixas F para *NonBuffers* possuem $w(F) < 1 - W$ e faixas F para *SmallBuffers* também possuem $w(F) < 1 - W$ e, portanto, não impedirão os *NonBuffers* de alcançar suas respectivas faixas, já que a largura dos *NonBuffers* é menor que W. Logo, sejam n_{lb}, n_{sb} e n_f, as quantidades de *LargeBuffers*, *SmallBuffers* e faixas abertas em um instante, respectivamente. Temos que a complexidade de tempo de ORP é limitada por $(n_{lb} + n_{sb}) + (n_f \cdot n_{lb}) = O(n_{lb} \cdot n)$.

Além disto, a corretude do algoritmo (com respeito as restrições) baseia-se no fato de que um novo item é empacotado em uma faixa alcançável ou em uma nova faixa, acima de todos os itens já empacotados. Portanto, há sempre um caminho livre para o item chegar a sua posição final de empacotamento.

Para provar o fator de aproximação do algoritmo vamos utilizar o limitante da área dos itens, visto que $\sum_{k=0}^{|I|} h(a_k)w(a_k) \leq OPT(I)$. O objetivo é provar que se escolhermos $\alpha = \frac{2}{3}$ e $W = \frac{1}{4}$ teremos que $\text{ORP}(I) \leq 4OPT(I) + 3$. Para tanto, classificaremos cada faixa utilizada para empacotar *NonBuffers* em dois grupos: Faixas F_i *Full*, são as que possuem $w(F_i) \geq 1 - 2W$, e as F_i' *NonFull*, caso contrário. Considere que faixas para empacotar *Buffers* também são *Full*. Para demonstrar a aproximação do algoritmo, mostraremos que ao menos $\frac{1}{4}$ da área de S estará ocupada, exceto por $\frac{1}{1-\alpha}$ de altura.

Lema 3.1.1. *Seja F uma faixa Full, então no mínimo $\frac{1}{4}$ da sua área está ocupada.*

Demonstração. Se F é uma faixa aberta para um *Buffer* B então a fração mínima de área

ocupada desta faixa é

$$\frac{h(B)w(B)}{h(B)} = w(B) \geq W.$$

Por outro lado, se F é uma faixa usada para empacotar *NonBuffers* e sua altura é $h(F_i)$ então podemos garantir que cada item tem altura mínima de α^{i+1} e, além disto, $w(F) \geq 1 - 2W$. Logo, o mínimo de área ocupada é

$$\frac{\alpha^{i+1}w(F)}{\alpha^i} = \alpha w(F) \geq \alpha(1-2W) = \frac{1}{3},$$

substituindo $W = \frac{1}{4}$ e $\alpha = \frac{2}{3}$, temos a demonstração completa. □

Para tratar as faixas *NonFull*, faremos uma associação entre faixas *Full* e *NonFull*. Esta associação criará uma única faixa que possui no mínimo W de área ocupada, exceto por $\frac{1}{1-\alpha}$ unidades de área.

Toda faixa F_i *NonFull* (obviamente uma faixa para empacotar *NonBuffers*) será associada, unicamente com uma faixa *Full*, dependendo da maneira como F_i foi aberta:

1. F_i é a primeira faixa de altura α^i.

2. F_i foi aberta para acomodar um *NonBuffer* que não pode ser empacotado na faixa F_i' que no momento estava aberta, pois isto acarretaria que $w(F_i') > 1 - W$. Neste caso F_i está associada com F_i'.

3. F_i foi aberta para acomodar um *NonBuffer* que foi bloqueado por um *LargeBuffer* (apenas este tipo de faixa pode bloquear *NonBuffers*). Neste caso F_i está associada com a faixa deste *LargeBuffer*.

Lema 3.1.2. *Independente da maneira como uma Faixa NonFull foi aberta, sua união com outra faixa Full fará com que elas ocupem, no mínimo, W de área.*

Demonstração. As faixas que encaixam-se no caso 1 serão desconsideradas. Como pode haver apenas uma faixa deste tipo por potência de α, no máximo $\sum_{i\geq 0}\alpha^i = \frac{1}{1-\alpha} = 3$ unidades de altura serão desconsideradas pelo algoritmo, já que $\alpha = \frac{2}{3}$.

No caso 2, associaremos F_i *NonFull* com a última faixa aberta F_i' para a mesma altura. A cada faixa *Full* pode estar associada apenas uma *NonFull*. Como F_i foi aberta porque a largura de F_i' excederia $1 - W$, então temos que a fracão mínima de área ocupada por F_i e F_i' é dada por

$$\frac{\alpha^{i+1}\cdot(1-W)}{\alpha^i \cdot 2} = \alpha\frac{1-W}{2},$$

substituindo $\alpha = \frac{2}{3}$ e $W = \frac{1}{4}$, temos que $\alpha\frac{1-W}{2} = \frac{2}{3}.\frac{\frac{3}{4}}{2} = \frac{1}{4}$.

No caso 3, associaremos F_i com o *LargeBuffer* B que o impediu de alcançar a sua faixa correta. A cada *LargeBuffer* pode estar associada apenas uma faixa de *NonBuffers* de cada tipo (potência de α). Portanto, no máximo $\sum_{i\geq 0} \alpha^i = \frac{1}{1-\alpha}$ de altura será associada a B. Além disto, cada faixa F_i associada a B tem $w(F_i) > 1 - w(B)$, pois foi bloqueada por B. Logo, seja $0 \leq h \leq \frac{1}{1-\alpha}$ a altura associada, a fração mínima de área ocupada é dada por

$$\frac{h(B)w(B) + \alpha h(1 - w(B))}{h + h(B)}, \tag{3.1}$$

com $1 - W \leq w(B) \leq h(B) \leq 1$ e $0 \leq h \leq \frac{1}{1-\alpha}$. No Apêndice A.1 mostramos que se escolhermos $\alpha = \frac{2}{3}$ e $W = \frac{1}{4}$ temos que o mínimo desta função ocorre quando $h(B) = w(B) = 1$ e $h = 3$, com valor mínimo de $\frac{1}{4}$, como enunciado. No algoritmo, este é o caso em que temos um *LargeBuffer* B, com $w(B) = h(B) = 1$ e 3 unidades de altura preenchidas com faixas F_i, $i \geq 0$, com $w(F_i) = \varepsilon$, com ε tão pequeno quanto se queira. □

Com estes resultados conseguimos provar a aproximação do algoritmo ORP no Teorema 1.

Teorema 1. *Seja I uma lista de itens, então temos que $ORP(I) \leq 4OPT(I) + 3$.*

Demonstração. : Seja ORP(I) a altura da faixa gerada pelo algoritmo ORP. Sabemos que $OPT(I) \geq Area = \sum_{i=0}^{|I|} h(a_i).w(a_i)$, logo, pelo Lema 3.1.2 temos que

$$\left(\text{ORP}(I) - \frac{1}{1-\alpha}\right) \cdot W \leq \sum_{i=0}^{|I|} h(a_i).w(a_i),$$

portanto,

$$\frac{1}{W}Area \geq \text{ORP}(I) - \frac{1}{1-\alpha} \Rightarrow \text{ORP}(I) \leq \frac{1}{W}OPT(I) + \frac{1}{1-\alpha}$$

e, substituindo $\alpha = \frac{2}{3}$ e $W = \frac{1}{4}$, temos que

$$\text{ORP}(I) \leq 4OPT(I) + 3.$$

□

3.1.2 Adaptação para o problema SPUH

Veremos nesta Seção como adaptar o algoritmo ORP para resolver o problema SPUH.

Como o algoritmo ORP é *online* e sua aproximação é puramente baseada na área dos itens, se impusermos qualquer ordem arbitrária para a lista de itens, ainda assim o Lema 3.1.2 será válido. Note, porém, que se impusermos uma determinada ordem à lista de itens, teremos um algoritmo *offline*.

O algoritmo U_{ORP} (Algoritmo 3.2) é simples: Primeiramente, a lista I de entrada é ordenada em ordem não crescente de classe, gerando I'. Depois o algoritmo ORP é aplicado em I'.

Algoritmo 3.2 *Unloading Online Rectangle Packing* - U_{ORP}

1: **input:** Lista I de itens particionados em C classes.
2: **begin**
3: $I' \leftarrow I$ ordenada por ordem não crescente de classe.
4: $P \leftarrow \mathrm{ORP}(I')$.
5: **return** P;
6: **End**

U_{ORP} é obviamente polinomial, com complexidade dominada pelo algoritmo ORP. No Lema 3.1.3 mostraremos que a solução gerada pelo algoritmo U_{ORP} satisfaz as restrições do problema SPUH.

Lema 3.1.3. *Seja P um empacotamento gerado pelo algoritmo U_{ORP}, então P satisfaz as restrições do problema SPUH.*

Demonstração. Suponha, por absurdo, que o empacotamento P não satisfaz as restrições impostas pelo problema SPUH. é fácil ver que as restrições de sobreposição e limites da faixa S são sempre satisfeitas. Logo, em P existe um item a_j que não possui um caminho livre, com movimentos horizontais e verticais, para ser removido de S sem movimentar outros itens de classes superiores. Considere o momento em que a_j foi empacotado no algoritmo ORP. Neste momento, todos os itens $a_i \in I'$ com classe $c(a_i) > c(a_j)$ já haviam sido empacotados em S, devido a ordenação de I'. Portanto, como há um caminho livre entre o topo de S e a posição em que a_j será empacotado e, como todos os itens posteriores a a_j possuem classe menor, podemos utilizar este mesmo caminho para remover a_j sem movimentar itens de classes superiores. Porém isto é uma contradição. □

Portanto, como o preenchimento de área demonstrado no Lema 3.1.2 é válido independente da ordenação dos itens, podemos garantir que $\mathrm{U}_{ORP}(I) \leq 4OPT(I) + 3$. Por fim, denote $OPT_u(I)$ como o valor de um empacotamento ótimo para o problema SPUH, então vale que $OPT_u(I) \geq OPT(I)$ e, portanto, $\mathrm{U}_{ORP}(I) \leq 4OPT_u(I) + 3$, como queríamos.

Logicamente, esta mesma ideia não pode ser diretamente aplicada ao problema SPU, pois o caminho que os itens percorrem pode conter movimentos horizontais.

3.2 Uma 2.6154-Aproximação para a Versão Restrita a Quadrados

Nesta Seção analisaremos o algoritmo proposto por Fekete *et al.* [17] para resolver o problema do *Strip Packing* com restrições *Tetris* para quadrados. Diferentemente do modelo adotado no problema da Seção anterior, agora temos a restrição adicional da *gravidade*. No algoritmo ORP (Algoritmo 3.1) os itens podem ficar "flutuando no ar", de forma que a restrição de gravidade pode não estar satisfeita.

O algoritmo *Online Square Packing* OSP (Algoritmo 3.3) proposto, utiliza uma estratégia diferente do algoritmo da Seção anterior. Ao invés de utilizar faixas horizontais disjuntas (níveis) para dividir os itens, o algoritmo proposto por Fekete *et al.* [17] utiliza faixas verticais sobrepostas (*slots*). Considere duas linhas paralelas e verticais partindo da base da faixa S (recipiente). A região entre estas duas linhas é chamada de *slot* e estas linhas recebem o nome de *limite esquerdo* e *direito* do *slot*. A distância entre estes dois limites é a *largura* do *slot*.

O algoritmo OSP funciona da seguinte maneira: Divida a faixa S de largura 1 em slots de diferentes larguras. Para $j = 0, 1, 2, \ldots$ crie 2^j slots de largura $\frac{1}{2^j}$ lado a lado. Logo, são criadas uma faixa de largura 1, duas faixas de largura $\frac{1}{2}$, quatro faixas de largura $\frac{1}{4}$ e assim por diante. Note que um slot de 2^{-i} contém dois slots de largura 2^{-i-1}. (Ver Figura 3.2). Para cada quadrado a_i arredonde $l(a_i)$ (como $h(a_i) = w(a_i)$, denotaremos por $l(a_i)$ o tamanho do lado do quadrado a_i) para o menor valor de 2^{-j} maior ou igual que $l(a_i)$. Empacote a_i (com o seu tamanho original $l(a_i)$) no *slot* de largura 2^{-j} que permita-o ser empacotado o mais baixo possível, movendo-o pelo *limitante esquerdo* do *slot* até encontrar outro item. O algoritmo OSP é apresentado em (Algoritmo 3.3).

Algoritmo 3.3 *Online Square Packing* OSP

1: **input:** Lista I de quadrados
2: **begin**
3: **while** $I \neq \varnothing$ **do**
4: $a_k \leftarrow primeiro(I)$.
5: $i \leftarrow \lfloor \log_{\frac{1}{2}} l(a_k) \rfloor$.
6: Empacote a_k alinhado ao *limite esquerdo* do *slot* de largura 2^i que o permita ser empacotado o mais baixo possível.
7: $I \leftarrow I \backslash a_k$.
8: **return** altura total da faixa;
9: **End**

3.2.1 Análise do Algoritmo OSP

Nesta Seção discutiremos sobre a complexidade, corretude e aproximação do algoritmo OSP.

No que diz respeito as restrições do problema, a corretude do algoritmo OSP decorre da forma como cada item alcança a sua posição final de empacotamento. Cada item percorre um caminho, alinhado ao *limite esquerdo* de um *slot*, até encontrar outro item (sobre o qual será empacotado) ou a base de S. Primeiramente, isto garante que há um caminho para alcançar esta posição e, obviamente, assegura a restrição de *gravidade*.

A ideia ingênua de representar os *slots* explicitamente nos levaria a um algoritmo de complexidade exponencial. Para garantir uma complexidade de tempo polinomial para o algoritmo OSP devemos utilizar uma ideia parecida com a do algoritmo *bottom-left*. Primeiramente defina C como o conjunto de pontos $p_i = (x_i, y_i)$, $i \leq n$, onde n é a quantidade de itens da entrada, que denotará a superfície do empacotamento, ou seja, o conjunto de segmentos definidos pelos itens nas posições mais altas para cada ponto do eixo das abscissas da faixa S. Inicialmente C contém apenas um ponto $p_1 = (0, 0)$ (inicialmente a base de S). Para cada novo item a_k que será empacotado em um *slot* de largura 2^{-k}, o algoritmo faz o seguinte: Cria uma lista auxiliar de pontos C' que conterá cada ponto da superfície em C transladado no eixo x para o próximo múltiplo de 2^{-k}. Com isso os pontos de empacotamento são exatamente alguns dos pontos de início dos *slots* de largura 2^{-k}. Note que em uma mesma superfície pode haver mais de um *slot* de largura 2^{-k} mas apenas o ponto de início da superfície (o mais a esquerda) é utilizado, pois se o item for empacotado sobre esta superfície então ele será empacotado o mais a esquerda possível. Após criar a lista C', realiza-se uma busca para ver em qual destes pontos da lista C', a_k ficará mais próximo da base de S sem inviabilizar a solução por sobreposição dos itens. Por fim, atualiza-se a lista C com o empacotamento do item a_k na posição escolhida.

A complexidade deste algoritmo é determinada pelas buscas e construções das listas C e C'. Para cada item da instância, o algoritmo pode aumentar em 2 o tamanho de C. Além disto, uma busca em $O(n^2)$ encontrará a melhor posição possível de empacotamento, checando para cada um dos n pontos a interseção dos n itens. Por fim, como o tamanho de C é limitado por $O(n)$, temos que a complexidade do algoritmo OSP implementado desta forma, é $O(n^3)$.

Resta-nos mostrar a garantia de aproximação do algoritmo OSP. Seja a_i um quadrado empacotado num *slot* T_i de largura 2^{-k_i}. Defina δ_i como a distância entre o lado direito de a_i e o limite direito do *slot* de largura 2^{-k_i+1} que contém a_i, enquanto que $\delta'_i = min\{l(a_i), \delta_i\}$. A *Sombra* de a_i, denotada por ${a_i}^S$, é a área obtida pelo aumento da largura de a_i em δ'_i do lado direito e $l(a_i) - \delta'_i$ do lado esquerdo. Logo, ${a_i}^S$ tem área do mesmo tamanho de a_i e está completamente contida no *slot* de largura 2^{-k_i+1} que contém

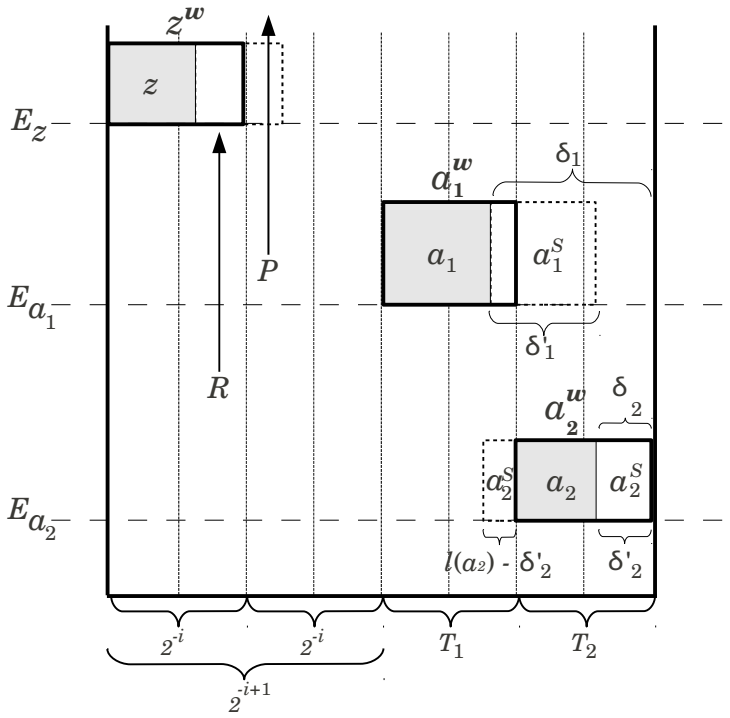


Figura 3.2: Representação dos itens e da faixa no algoritmo OSP.

a_i. Além disto, define-se ${a_i}^w = ({a_i} \cup {a_i}^S) \cap T_i$.

Considere um ponto P em T_i, onde $P \notin ({a_j} \cup {a_j}^S)\ \forall j$, ou seja, P é um ponto no slot T_i que não pertence a nenhum item e sua *sombra*. Associe P a um quadrado a_i acima de P se ${a_i}^w$ é o primeiro item alcançado por uma linha vertical a partir de P. Denota-se por F_{a_i} o conjunto de todos os pontos associados ao quadrado a_i e $|F_{a_i}|$ como sua área. O conjunto de pontos que formam os limites dos *slots* possuem área nula e, por isto, são desconsiderados na análise. Para a análise considera-se a existência de um quadrado, a_{n+1}, de tamanho 1 no topo do empacotamento. Desta forma todo ponto até a altura da solução gerada ou pertence a ${a_i}^w$ ou pertence a algum conjunto F_{a_i}.

Na Figura 3.2 estão representados os quadrados a_1, a_2 e z, juntamente com suas *sombras* e representações de δ, δ' e ${a_i}^w$. Note que o ponto R está associado ao quadrado z, pois intercepta z^w, enquanto que P não está.

Teorema 2. *O algoritmo OSP possui fator competitivo de 2.6154.*

Demonstração. A prova será baseada na área ocupada da faixa S. Como a_i e ${a_i}^S$ possuem a mesma área, pode-se limitar a área, A, produzida pelo algoritmo OSP da seguinte maneira:

$$A \leq 2\sum_{i=1}^{n} l(a_i)^2 + \sum_{i=1}^{n+1} |F_{a_i}|.$$

O limitante trivial do empacotamento ótimo é $\sum_{i=1}^{n} l(a_i)^2$ e, portanto, basta demonstrar que $|F_{a_i}| \leq 0.6154 \cdot l(a_i)^2$ para todo quadrado a_i. Para cada quadrado a_i constrói-se uma sequência de quadrados $S = B_1, B_2, B_3, \ldots, B_m$, onde $B_1 = a_i$. Define-se E_{B_j} como a "extensão" da base do quadrado B_j para a esquerda e a direita até os limites da faixa S (ver Figuras 3.2 e 3.3). Será mostrado que, através da escolha correta dos quadrados da sequência, poderemos limitar a área de F_{B_1} entre um par de extensões, E_{B_j} e $E_{B_{j+1}}$, em termos de B_{j+1} e da largura dos *slots*. Assume-se, para facilitar a notação, que o quadrado B_j está empacotado no slot T_j, de largura 2^{-k_j}. Note que F_{B_1} está contido em T_1.

Slots dividem-se em *ativos* e *inativos* (com relação a E_{B_j} e B_1):

- *Ativo*: Caso este *slot* possua um ponto abaixo de E_{B_j} associado a B_1.
- *Inativo*: Caso contrário.

A sequência de quadrados é escolhida da seguinte maneira: B_1 é o primeiro quadrado e cada quadrado B_{j+1}, $j = 1, \ldots, m-1$ é o menor quadrado que intercepta E_{B_j} em um *slot* ativo de largura 2^{-k_j}. A sequência termina quando todos os *slots* são inativos com relação a E_{B_m}.

Na Figura 3.3 pode-se ver os três primeiros quadrados B_1, B_2 e B_3 de uma sequência, onde B_{j+1} é o menor quadrado que intercepta a extensão E_{B_j}, para $j = 1, 2$. As áreas hachuradas H_2 e H_3 representam os pontos associados ao quadrado B_1 entre E_{B_1} e E_{B_2} e E_{B_2} e E_{B_3}, respectivamente. Note também que nesta sequência o *slot* T_2 de largura 2^{-k_2} é inativo com relação a qualquer extensão E_{B_j}, $j > 2$, com análise semelhante para T_3.

A respeito desta sequência de itens, as seguintes assertivas são provadas no Lema 3.2.1.

1. B_{j+1} existe para todo $j + 1 \leq m$ e $l(B_{j+1}) \leq 2^{-k_j - 1}$ para $j + 1 \leq m - 1$.
   - Isto nos garante que os itens diminuem de tamanho a medida que avançamos na sequência. Além disto, nos mostra que a sequência existe e termina quando restam apenas *slots* inativos.
2. O número de slots ativos com relação a E_{B_j} de largura 2^{-k_j} é, no máximo:

$$\begin{cases} 1, & \text{j} = 1 \\ \prod_{i=2}^{j} \left(\frac{1}{2^{k_i - 1}} 2^{k_i} - 1 \right), & j \geq 2. \end{cases}$$

   - Isto nos garante que a quantidade de *slots* ativos com relação a E_{B_j} é limitada por um fator de $(\frac{1}{2^{k_j - 1}} 2^{k_j} - 1)$ vezes o número de *slots* ativos com relação a $E_{B_{j-1}}$.

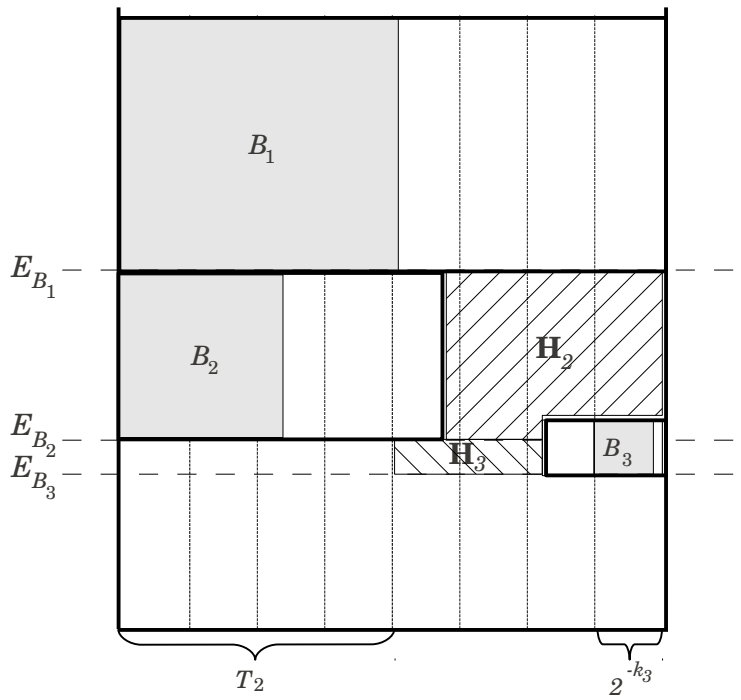


Figura 3.3: Representação de itens da sequência $B_1, \ldots, B_m$.

3. A parte da área de F_{B_1} que pertence a um *slot* ativo de largura 2^{-k_j} entre E_{B_j} e $E_{B_{j+1}}$ é, no máximo, $2^{-k_j} l(B_{j+1}) - 2l(B_{j+1})^2$.

    - Com isto conseguiremos limitar a área associada a B_1 entre duas extensões E_{B_j} e $E_{B_{j+1}}$, pois para cada B_j escolhemos B_{j+1} como o menor quadrado que intercepta E_{B_j} e portanto o menor slot T_j ativo.

Assumindo estas afirmações, podemos limitar a área de $|F_{B_1}|$ por

$$\frac{l(B_2)}{2^{k_1}} - 2l(B_2)^2 + \sum_{j=2}^{m} \left[\left(\frac{l(B_{j+1})}{2^{k_j}} - 2l(B_{j+1})^2 \right) \prod_{i=1}^{j-1} \left(\frac{2^{k_{i+1}}}{2^{k_i}} - 1 \right) \right].$$

Esta expressão é maximizada se $l(B_{i+1}) = \frac{1}{2^{k_i+2}}$. Isto implica $k_i = k_1 + 2(i-1)$, logo

$$|F_{B_1}| \leq \sum_{i=0}^{\infty} \frac{3^i}{2^{2k_i+4i+3}}.$$

A razão $\frac{|F_{B_1}|}{|B_1{}^2|}$ é maximizada se $|B_1|$ for o menor possível, ou seja, $2^{-(k_1+1)} + \epsilon$. Portanto

$$\frac{|F_{B_1}|}{|B_1{}^2|} \leq \sum_{i=0}^{\infty} \frac{3^i.2^{2k_i+2}}{2^{2k_i+4i+3}} = \sum_{i=0}^{\infty} \frac{3^i}{2^{4i+1}} = \frac{1}{2}\sum_{i=0}^{\infty}\left(\frac{3}{16}\right)^i = \frac{8}{13} = 0.6154\ldots$$

□

Lema 3.2.1. *Seja $B_1, \ldots, B_m$ uma sequência de itens definida da seguinte maneira: B_1 é o primeiro quadrado e o quadrado seguinte B_{j+1}, $j = 1, \ldots, m-1$ é o menor quadrado que intercepta E_{B_j} em um slot ativo de largura 2^{-k_j}. Então vale que:*

1. *B_{j+1} existe para todo $j+1 \leq m$ e $l(B_{j+1}) \leq 2^{-k_j-1}$ para $j+1 \leq m-1$.*
2. *O número de slots ativos com relação a E_{B_j} de largura 2^{-k_j} é, no máximo:*

$$\begin{cases} 1, & j = 1 \\ \prod_{i=2}^{j}\left(\frac{1}{2^{k_i-1}}2^{k_i}-1\right), & j \geq 2 \end{cases}$$

3. *A parte da área de F_{B_1} que pertence a um slot ativo de largura 2^{-k_j} entre E_{B_j} e $E_{B_{j+1}}$ é, no máximo, $2^{-k_j}l(B_{j+1}) - 2l(B_{j+1})^2$.*

Demonstração. Provaremos a assertiva (*1.*) por absurdo.

Suponha por contradição que algum item B_{j+1} não exista, $j+1 \leq m$. Logo, não existe nenhum item interceptando E_{B_j} em um slot ativo de largura 2^{-k_j} e podemos concluir que poderíamos empacotar B_j em uma posição mais baixa o que é uma contradição, já que o algoritmo escolhe a melhor posição possível para cada item. Portanto, sempre existe o item B_{j+1}, $j+1 \leq m$, na sequência. Suponha por absurdo também, que $j+1 < m$ e $l(B_{j+1}) > 2^{-k_j-1}$. Então, como B_{j+1} foi escolhido de tamanho mínimo, isto implicaria que todos os *slots* de tamanho 2^{-k_j} possuem um item de largura $\geq 2^{-k_j-1}$ interceptando E_{B_j}. Mas neste caso todos os slots de largura 2^{-k_j} estariam inativos em relação a B_1 e B_j seria o último item da sequência, o que é uma contradição. Portanto, para todo item B_{j+1} da sequência, vale que $l(B_{j+1}) \leq 2^{-k_j-1}$ ou $j+1 = m$.

As assertivas (*2.*) e (*3.*) serão demonstradas por indução no número de itens da sequência.

Base: Considere que B_1 está na base da faixa S, então vale que B_1 é o último item da sequência, com 0 *slots* ativos em E_{B_1} e $|F_{B_1}| = 0$. Por outro lado, se B_1 não estiver na base de S, então existe pelo menos um item B_2 na sequência (assertiva *1.*). Como $l(B_2) \leq 2^{-k_i-1}$, T_1 é o único *slot* de largura 2^{-k_1} ativo. Além disto, pode-se concluir que a parte de F_{B_1} que está entre E_{B_1} e E_{B_2} é, no máximo $2^{-k_1}l(B_2) - 2l(B_2)^2$, já que esta área abaixo de B_1 contém totalmente B_2 e sua *sombra*. (Veja a Figura 3.3).

Hipótese Indutiva: Assumiremos que até o item B_j na sequência há, no máximo, $M = (\frac{2^{k_2}}{2^{k_1}} - 1) \cdot (\frac{2^{k_3}}{2^{k_2}} - 1) \cdot \ldots \cdot (\frac{2^{k_j}}{2^{k_{j-1}}} - 1)$ *slots* ativos de largura 2^{-k_j} com respeito a E_{B_j}.
Passo: Cada um destes *slots* ativos contêm $2^{k_{j+1}-k_j}$ *slots* de largura $2^{k_{j+1}}$. Além disto, a maneira como B_{j+1} foi escolhido nos garante que T_{j+1} não estará mais ativo. Logo, a quantidade de *slots* ativos, após a escolha do quadrado B_{j+1} é $M \cdot (\frac{2^{k_{j+1}}}{2^{k_j}} - 1)$. Como sabemos também que B_{j+1} existe, podemos ver que a área em *slots* de largura 2^{k_j} associada a B_1 entre E_{B_j} e $E_{B_{j+1}}$ é, no máximo $2^{-k_j} l(B_{j+1}) - 2l(B_{j+1})^2$, como enunciado. Pois para o slot contendo B_{j+1} este valor é exato, porém para os demais slots pode ser menor, já que os itens que tocam E_{B_j} podem ser maiores que B_{j+1}. Na Figura 3.3 podemos ver a representação desta análise, considerando os itens B_1, B_2 e B_3. Nela, as áreas hachuradas são limitadas por $2 - k_j l(B_{j+1}) - 2l(B_{j+1})^2$, para $j = 1, 2$. Além disto o *slot* T_2 é inativo com relação a E_{B_3} e qualquer E_{B_k}, $k > 3$.

□

3.2.2 Adaptação para o problema SPU

Veremos nesta Seção como adaptar o algoritmo OSP para resolver o problema SPU. A ideia é semelhante à utilizada na adaptação do algoritmo ORP para o problema SPUH. Desta vez conseguimos um algoritmo 2.6154-aproximado para o caso especial do problema SPU onde as instâncias são tais que os itens são quadrados.

Neste caso, o algoritmo OSP também é *online* e sua aproximação também é baseada na área dos itens, logo, em qualquer ordem arbitrária imposta aos itens ainda valerá o Teorema 2. O algoritmo U_{OSP} (Algoritmo 3.4) para o problema SPU é *offline* pois ordena os quadrados da entrada.

O algoritmo U_{OSP} simplesmente ordena os itens da lista de entrada em ordem não crescente de classe. Depois o algoritmo OSP é aplicado nos itens ordenados.

Algoritmo 3.4 *Unloading Online Square Packing* - U_{OSP}

1: **input:** Lista I de itens particionados em C classes.
2: **begin**
3: $I' \leftarrow I$ ordenada por ordem não crescente de classe.
4: $P \leftarrow \mathrm{OSP}(I')$.
5: **return** P;
6: **End**

O algoritmo U_{OSP} tem complexidade de tempo polinomial, com complexidade dominada pelo algoritmo OSP (demonstrado da Seção anterior). No Lema 3.2.2 mostraremos que a solução gerada pelo algoritmo U_{OSP} satisfaz as restrições do problema SPU.

Lema 3.2.2. *Seja P um empacotamento gerado pelo algoritmo U_{OSP}, então P satisfaz as restrições do problema SPU.*

Demonstração. A ideia da prova é semelhante à usada no Lema 3.1.3.

Suponha, por absurdo, que o empacotamento P não satisfaz as restrições do problema SPU. As restrições de sobreposição e limites da faixa S são sempre satisfeitas pela viabilidade da solução do algoritmo OSP. Logo, existe em P um item a_j que não possui um caminho livre e vertical para ser removido de S sem movimentar itens de classes superiores. No momento em que a_j foi empacotado pelo algoritmo OSP, todos os itens $a_i \in I'$ com classe $c(a_i) > c(a_j)$ já haviam sido empacotados em S, devido a ordenação de I'. Logo, como há um caminho livre entre o topo de S e a posição em que a_j será empacotado e, como todos os itens posteriores a a_j possuem classe menor, podemos utilizar este mesmo caminho para remover a_j sem movimentar itens de classes superiores. Mas isto é uma contradição. □

Levando em consideração que a garantia do preenchimento de área demonstrado no Teorema 2 é válida para qualquer ordenação dos itens, podemos garantir que $\mathrm{U}_{OSP}(I) \leq 2.6154 \cdot OPT(I)$. Por fim, denote $OPT_u(I)$ como o valor de um empacotamento ótimo para o problema SPU, logo, pela restrição adicional, vale que $OPT_u(I) \geq OPT(I)$ e, portanto, $\mathrm{U}_{OSP}(I) \leq 2.6154 OPT_u(I)$, como enunciado.

Podemos também garantir que a aproximação vale para o problema SPUH pois a demonstração de aproximação utiliza apenas área dos itens como limitante para o ótimo.

Capítulo 4

Heurísticas para o problema SPU

Nesta Seção apresentamos as heurísticas propostas para o problema SPU. A primeira delas é um algoritmo combinatório baseado em um algoritmo para o problema *Bin Packing* bidimensional, o qual provamos ter um fator de aproximação assintótico de 6.75. Também apresentaremos uma segunda heurística baseada no algoritmo OSP (Algoritmo 3.3) e mostramos que se a instância do problema for tal que haja uma quantidade constante de classes, então o algoritmo é uma 2-aproximação. Por fim, apresentaremos uma heurística *GRASP* baseada no trabalho apresentado em [1].

4.1 Uma 6.75-aproximação

Nesta Seção apresentamos o algoritmo *Level Bin Packing* (LBP). Este algoritmo computa a solução em dois estágios. Primeiro ele empacota os itens em *bins* de largura e altura 1, utilizando níveis (faixas) para isto. Depois ele empacota cada bin gerado rotacionado, de forma que cada faixa torna-se um *slot* de altura no máximo 1 (a largura máxima de um *bin*). Devido a forma como o algoritmo LBP empacota os itens, poderemos redimensionar suas larguras para valores inferiores a 1. Por exemplo, veremos que alguns itens "grandes" serão empacotados em *bins* individuais e com largura modificada. Primeiro apresentamos no Algoritmo 4.1 o algoritmo para o empacotamento dos *bins* o qual chamamos de *Bin Packing Decreasing Order* (*BPDO*) e em seguida, apresentamos o LBP no Algoritmo 4.2.

Considere a seguinte divisão dos itens: *Buffers* são itens com pelo menos uma dimensão maior que 1/3 e *NonBuffers* os itens restantes. Denote por B_k o k-ésimo *bin* utilizado no algoritmo *BPDO*, além disto $h(B_k)$ é a altura ocupada pelas faixas criadas no *bin* B_k e $w(B_k)$ a largura ocupada no mesmo *bin*. Os *Buffers* são empacotados em faixas individuais, de altura igual à sua, ou *bins* de altura 1 e largura igual a do *buffer*. Uma faixa usada para empacotar um *Buffer* será considerada *full*.

Os *NonBuffers* são empacotados em faixas F_j de altura $h(F_j) = \frac{1}{3 \cdot 2^j}$, para $j = 0, 1, \ldots$.

Um *NonBuffer* a_i tem **tipo** F_j se é empacotado em uma faixa F_j. O algoritmo *BPDO* empacota os itens em faixas nos *bins*. Uma vez fechado, um *bin* não é mais reutilizado. Os itens são empacotados o mais a esquerda possível lado a lado. Para cada faixa F_j denote por I_{F_j}, a altura do item mais alto em F_j. As faixas mantêm sua altura original até o momento em que são fechadas e então o algoritmo faz com que $h_{F_j} = h(I_{F_j})$. Denotamos por $S(B_k)$ o conjunto de faixas empacotadas em B_k. Uma faixa usada para empacotar itens *NonBuffer* será considerada *full* se a soma da largura dos itens nela empacotados for superior a $2/3$ e *non-full* caso contrário.

O algoritmo *BPDO* (ver Algoritmo 4.1) funciona da seguinte maneira: Primeiramente, os itens são ordenados em ordem não crescente de classe c (linha 3) e rotacionados de forma que $h(a_i) \leq w(a_i)$ (linha 4). Para cada classe (linha 5) ele empacota primeiro os *Buffers* $b_0, \ldots, b_x$ em faixas individuais no último *bin* utilizado até o momento, enquanto ainda cabem (linha 6). Então ele empacota os *Buffers* restantes rotacionados, de forma que $w(b_i) \leq h(b_i)$, em *bins* individuais de largura $w(b_i)$ e fecha os bins utilizados (linha 7). Depois de empacotar os *Buffers* de uma classe, o algoritmo empacota os *NonBuffers* (linhas 8-11). Para cada item *NonBuffer* a_i, o algoritmo empacota-o numa faixa aberta F_j de altura $\frac{1}{3 \cdot 2^j}$ tal que $\frac{1}{3 \cdot 2^{j+1}} < h(a_i) \leq \frac{1}{3 \cdot 2^j}$ (linha 9). Se o item não pode ser empacotado em F_j, então esta faixa é fechada pois está *full*; uma nova faixa é criada e a_i é empacotado nela. Se após empacotar um *NonBuffer* a_i tivermos $\sum_{F \in S(B_k)} I_F > 1$ então o algoritmo remove a_i da faixa e empacota-o em uma nova faixa em um novo *bin*, fechando o *bin* em que a_i não coube (linha 11).

Algoritmo 4.1 *Bin Packing Decreasing Order* - BPDO

1: **input:** A lista L de itens divididos em C classes diferentes.
2: **begin**
3: Ordene L por ordem não crescente de classe.
4: $\forall a_i \in L$, rotacione-o de forma que $h(a_i) \leq w(a_i)$.
5: **for** ($c = C$ **downto** 1) **do**
6: Empacote os *Buffers* da classe c no *bin* atualmente aberto B_k enquanto$\sum_{F \in S(B_k)} I_F \leq 1$, cada *Buffer* em uma faixa individual de altura igual à sua.
7: Rotacione os *Buffers* restantes b_i de forma que $w(b_i) \leq h(b_i)$ e empacote-os em *bins* individuais de largura $w(b_i)$ e feche cada um destes *bins*.
8: **for** (cada *NonBuffer* a_i da classe c) **do**
9: Empacote a_i na faixa aberta F_j onde $1/(3 \cdot 2^{j+1}) < h(a_i) \leq 1/(3 \cdot 2^j)$. Se não houver uma faixa destas aberta ou a_i não couber na existente, então crie uma nova faixa F_j e empacote a_i nela. Feche a antiga F_j se ela existia e faça $h(F_j) = I_{F_j}$.
10: **if** ($\sum_{F \in S(B_k)} I_F > 1$) **then**
11: Remova o item a_i do *bin* atual B_k e empacote-o em uma nova faixa num novo *bin* B_{k+1} na base deste e feche o *bin* B_k.
12: **return** Os *bins* criados.
13: **end**.

O algoritmo LBP é apresentado a seguir (Algoritmo 4.2). Ele simplesmente chama o algoritmo *BPDO*, concatena todos os *bins* retornados formando uma faixa de altura 1 e largura igual a soma das larguras destes *bins*. A faixa é então rotacionada no sentido anti-horário, fornecendo assim, uma solução para o problema original.

Algoritmo 4.2 *Level Bin Packing* - LBP

1: **input:** $L = \{a_1, a_2, \ldots, a_n\}$
2: **begin**
3: Sejam $B_1, B_2, \ldots, B_m$ os *bins* computados pelo algoritmo *BPDO* na ordem em que foram criados.
4: Concatene $B_1, B_2, \ldots, B_k$ formando uma faixa S de altura 1 e largura $\sum_{k=1}^{m} w(B_i)$.
5: **return** S rotacionada de forma que sua largura é 1 e sua altura é $\sum_{k=1}^{m} w(B_k)$.
6: **end**.

Teorema 3. *O empacotamento gerado pelo algoritmo LBP satisfaz as restrições do problema SPU.*

Demonstração. Sejam $B_1, \ldots, B_m$ os *bins* criados pelo algoritmo *BPDO* na ordem em que foram criados. Estes *bins* foram empacotados na faixa S exatamente nesta mesma ordem. Para cada par de *bins* B_k e B_{k+1} na sequência, podemos garantir que todos os itens em B_{k+1} possuem classe menor ou igual que a classe de qualquer item em B_k, pois os itens foram empacotados em ordem não crescente de classe. Portanto, por transitividade, itens em um determinado *bin* nunca bloqueiam itens de outros *bins*. Além disto, dentro de cada *bin*, a viabilidade da solução também é garantida pela ordem imposta antes do empacotamento dos itens. Como cada item é empacotado em uma faixa na posição mais à esquerda, mas à direita do último item empacotado, este não poderá ser bloqueado por nenhum item já empacotado nesta faixa. Perceba também que faixas diferentes não interferem umas nas outras pois os itens são empacotados totalmente dentro de apenas uma faixa. □

Portanto, conseguimos garantir que o algoritmo sempre retorna soluções viáveis para o problema em questão. Na figura 4.1 podemos ver uma ilustração da ideia dos algoritmos LBP e *BPDO*. Na parte (a) desta Figura, vemos 5 bins gerados na solução do algoritmo *BPDO*. Como o primeiro item da classe 3 é um *Buffer* e o *bin* atual estava vazio no momento de empacotá-lo ele é rotacionado para que o seu lado mais estreito seja a sua base e é empacotado em um *bin* individual com largura reduzida. Após isto, novos itens *NonBuffers* da classe 3 são empacotados no segundo *bin*. As classes restantes são empacotadas seguindo o Algoritmo 4.2. Por fim, na parte (b) da figura 4.1 temos os bins gerados na parte (a) rotacionados e concatenados (empilhados) para formar a solução em uma faixa S apenas. Note que na parte (b) os *bins* B_i, $i = 2, 3, 5$ têm a sua largura

diminuída para $w(B_i)$ apenas como uma melhoria no algoritmo mas sem interferência no fator de aproximação.

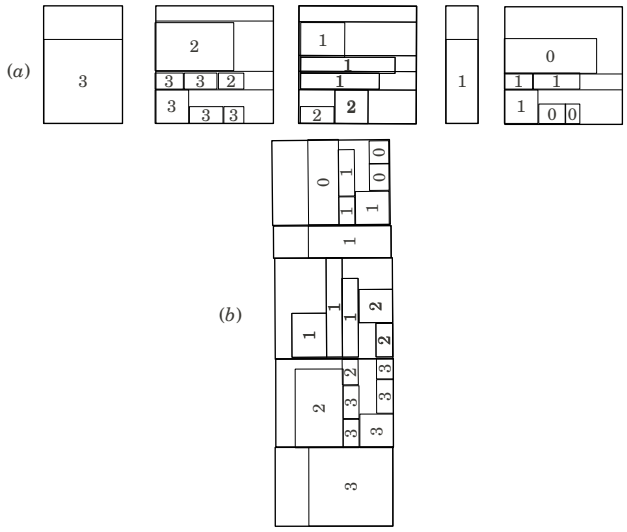


Figura 4.1: Empacotamento gerado pelo algoritmo LBP.

4.1.1 Análise do Algoritmo LBP

No que diz respeito a complexidade de tempo do algoritmo LBP, podemos ver que ela é dominada pelo algoritmo *BPDO*, o qual pode ser implementado em $O(n \lg n)$, já que além da ordenação dos itens podemos realizar as operações restantes em $O(1)$ para cada item da entrada.

Nesta Seção utilizaremos argumentos baseados no preenchimento de área para provar o fator de aproximação assintótico do algoritmo LBP.

Lema 4.1.1. *As faixas abertas para empacotar Buffers e as faixas que foram fechadas (full) têm pelo menos* $\frac{1}{3}$ *de sua área preenchida.*

Demonstração. A prova é simples e similar a do Lema 3.1.1.

Se F é uma faixa onde um *Buffer* B foi empacotado, então podemos garantir que a fração de área ocupada nela é

$$\frac{h(B)w(B)}{h(B)} = w(B) \geq \frac{1}{3}.$$

Da mesma forma, se F_i é uma faixa onde *NonBuffers* foram empacotados e sua altura é $h(F_i)$ então podemos garantir que cada item empacotado nela tem altura de pelo menos

$h(F_i)/2$ e, além disto, $w(F_i) \geq 2/3$, já que esta faixa é *full*. Logo, a fração de área ocupada nela é pelo menos

$$\frac{h(F_i)/2 \cdot 2/3}{h(F_i)} = \frac{1}{3}.$$

□

Na prova do Lema 4.1.2 assumiremos que existe, no máximo $\frac{2}{3}$ de altura ocupada por faixas *non-full*. Isto decorre do fato de que existe, no máximo, uma faixa *non-full* para cada altura $\frac{1}{3 \cdot 2^j}$ aberta a cada instante no algoritmo. Desta forma, podemos limitar a altura ocupada pelas faixas *non-full* em um *bin* por $\sum_{j=0}^{\infty} \frac{1}{3 \cdot 2^j} = \frac{2}{3}$.

Lema 4.1.2. *(i) Suponha que uma fração de altura* $1/3$ *de um bin é usada para empacotar faixas full e uma faixa non-full* F_0*. O mínimo de área ocupada ocorre quando* F_0 *é ocupada com um item* a_i *de altura* $1/6$*. A área total dos itens empacotados é, no mínimo* $1/12$*.*

(ii) Suponha que uma fração de altura $1/3$ *de um bin é usada para empacotar faixas full e non-full. O mínimo de área preenchida ocorre quando há apenas faixas non-full* F_j*, para* $j \geq 1$*. A área total dos itens empacotados neste caso é de no mínimo* $1/27$*.*

Demonstração. Primeiramente, perceba que como uma faixa *full* está pelo menos $1/3$ ocupada pelos seus itens, o total de área ocupada por itens considerando todas as faixas *full* é no mínimo $1/3$ multiplicado pela altura total destas faixas. Note também que para uma faixa *non-full* o pior caso de ocupação ocorre quando apenas um item é empacotado nela.

Em (i), o mínimo de área preenchida pelos itens ocorre quando F_0 tem apenas um item a_i. Portanto, a área dos itens nesta fração do *bin* é pelo menos

$$h(a_i)^2 + (1/3 - h(a_i)) \cdot 1/3. \tag{4.1}$$

já que a área de a_i é pelo menos $h(a_i)^2$ pois $h(a_i) \leq w(a_i)$, e o restante da altura $(1/3 - h(a_i))$ tem, no mínimo $1/3$ de área ocupada. O mínimo desta função ocorre quando $h(a_i) = 1/6$ e, portanto, o mínimo no preenchimento de área é $1/12$.

Em (ii) suponha uma configuração c, de área $A(c)$, onde há apenas faixas *non-full* F_j com apenas um item, para cada $j \geq 1$ ocupando exatamente o $1/3$ de altura (Ver Figura 4.2). Basta demonstrar então, que se substituirmos algum trecho de altura h nesta configuração por um trecho de uma faixa *full* ou uma faixa F_0 *non-full*, não diminuiremos o preenchimento de área neste $1/3$ de altura.

Considere que um trecho de altura h foi substituída por uma faixa F_0 *non-full* ou uma faixa *full* gerando uma configuração c' (Ver Figura 4.2). Se subtrairmos a área das

configurações $(A(c') - A(c))$, vemos que uma parte de altura $1/3 - h$ não é alterada e, portanto, basta calcular a diferença no trecho modificado.

Note que as faixas *full* F_i possuem $P \geq 1/3$ de área preenchida e podemos considerar então que estas estão preenchidas com altura $h(F_i)$ e largura maior ou igual a $1/3$. Além disto, Faixas F_0 *non-full* possuem pelo menos um item, e este possui largura $l > 1/6$. Por fim, veja que faixas F_i, $i \geq 1$, *non-full* com um item apenas possuem $l' \leq 1/6$ e, portanto, se subtrairmos a área preenchida nas partes diferentes em c' e c, teremos

$$A(c') - A(c) = min\{P, h \cdot l\} - h \cdot l' \geq h \cdot (l - l') \geq 0 \tag{4.2}$$

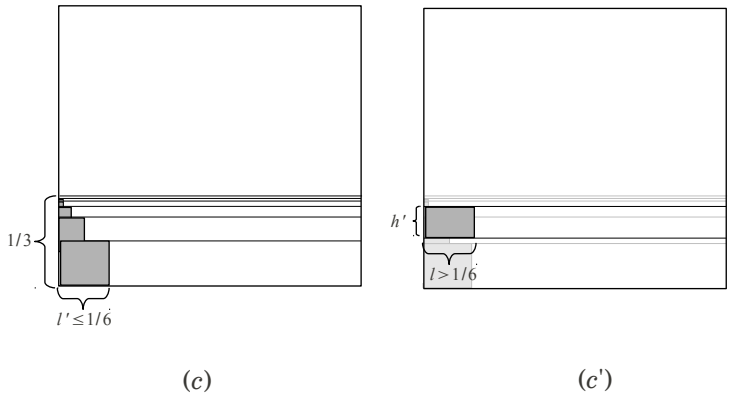


Figura 4.2: Representação das configurações c e c'.

Logo, um limite mínimo de área preenchida é dado pela configuração que só usa faixas F_i, $i \geq 1$, *non-full*, cada qual com apenas um item e de altura máxima. Note que a área mínima nestas faixas *non-full* é de

$$\sum_{j=1}^{\infty} \left(\frac{1}{3 \cdot 2^j} \right)^2 = \frac{1}{27}.$$

□

Lema 4.1.3. *Todos os bins, exceto possivelmente o último, estão em média com área preenchida de* $\frac{4}{27}$.

Demonstração. Provaremos este Lema por indução no número k de bins usados no algoritmo.

Considere o caso base com $k = 1$. Então temos apenas um *bin* e a prova é trivial. Como hipótese indutiva, assuma então que, exceto o último *bin* (B_{k-1}), os *bins* têm pelo menos $\frac{4}{27}$ de ocupação média de área. Agora, considere a maneira como o bin B_k foi aberto. Defina F e NF ambos contidos em $S(B_{k-1})$ como os conjuntos de faixas *full* e *non-full*, respectivamente, do *bin* B_{k-1}. Dividiremos esta análise em dois casos:
Caso 1: Considere que o *bin* B_k foi aberto por um item a_i *NonBuffer* porque quando empacotado em B_{k-1}, a_i fez com que tivéssemos $\sum_{F \in S(B_{k-1})} I_F > 1$. Logo, no mínimo, $1 - h(a_i)$ de altura é utilizada por faixas *full* (F) e *non-full* (NF) em B_{k-1}. Suponha que a_i é do **tipo** F_j. Podemos então assumir que a faixa F_j não faz parte do conjunto NF, pois quando empacotamos a_i em uma faixa *non-full* F_j temos que $\sum_{F_l \in S(B_{k-1})} I_{F_l} > 1$, logo $h(a_i) > I_{F_j}$, portanto $1 - h(a_i)$ já exclui a altura utilizada pela faixa F_j. Então na altura $1 - h(a_i)$ utilizada para empacotar faixas, podemos assumir que não existe F_j *non-full*.

Suponha então que o item a_i tem **tipo** F_0. Neste caso, pelo menos $(1 - 2/3)$ de altura é utilizado por faixas *full*, pois a altura máxima de a_i é $1/3$ e a altura máxima do conjunto NF também é $1/3$. Considere uma fração do *bin* de altura $1/3$ que contem NF. Pelo Lema 4.1.2 esta fração tem área ocupada maior ou igual a $1/27$. Logo, o total de área ocupada no *bin* B_{k-1} é, no mínimo $(1 - 2/3) \cdot 1/3 + 1/27 = 4/27$.

Suponha agora que a_i tem **tipo** F_j, para $j \geq 1$. Neste caso, no mínimo $(1 - 1/6 - 2/3)$ de altura é utilizado exclusivamente por faixas *full*, já que a altura máxima de a_i é $1/6$ e a altura máxima do conjunto NF é $2/3$. Logo, pelo Lema 4.1.2 a fração $1/3$ de altura utilizada por F_0 e por faixas *full*, tem área mínima ocupada de $1/12$, enquanto que a outra fração $1/3$ tem pelo menos $1/27$ de área. Portanto, neste caso, o total de área preenchida no *bin* B_{k-1} é de pelo menos $(1 - 1/6 - 2/3) \cdot 1/3 + 1/12 + 1/27 > 4/27$.
Caso 2: Considere que B_k foi aberto por um *Buffer* b tal que $w(B_k) = w(b) \leq h(b)$. Primeiramente, considere que $w(b) > 2/3$. Neste caso, á área ocupada nos dois *bins* B_{k-1} e B_k é pelo menos

$$\frac{h(b) \cdot w(b)}{1 + w(b)} \geq \frac{w(b)^2}{1 + w(b)} > \frac{4}{27},$$

já que $w(b) > 2/3$ e esta função é crescente para $1 > w(b) \geq 2/3$.

Considere agora que $2/3 \geq w(b) \geq 0$. Neste caso, há pelo menos $1/3$ de altura preenchida em B_{k-1} utilizada por faixas *full* e *non-full*. Pelo Lema 4.1.2 o pior caso de preenchimento ocorre quando este $1/3$ de altura disponível é ocupada apenas por faixas *non-full* F_j para $j \geq 1$ com área mínima de $1/27$. O restante de altura $(1 - 1/3 - w(b))$ pode ser utilizado por faixas *full*, totalizando uma altura de h' e uma faixa *non-full* F_0

com apenas um item a. O mínimo de área ocupada é dado por

$$\frac{w(b)^2 + 1/27 + h(a)^2 + \frac{1}{3}h'}{1 + w(b)}, \tag{4.3}$$

onde $1/6 < h(a) \leq 1/3$, $h(a) + h' = 1 - w(b) - 1/3$ and $2/3 \geq w(b) > 0$. Como podemos ver no Apêndice A.2 o mínimo desta função é $0.168518 > \frac{4}{27}$. □

Teorema 4. *Seja L uma lista de retângulos, então vale que $LBP(L) \leq 6.75OPT(L) + 1$.*

Demonstração. Pelo Lema 4.1.3, a ocupação média de área na faixa S é de, no mínimo $\frac{4}{27}$, exceto pelo último bin gerado no algoritmo *BPDO*. Então

$$(\mathrm{LBP}(L) - 1)\frac{4}{27} \leq \sum_{a_i \in L} w(a_i) \cdot b(a_i)$$

e portanto

$$\mathrm{LBP}(L) \leq 6.75 \sum_{a_i \in L} w(a_i) \cdot b(a_i) + 1 \leq 6.75OPT(L) + 1.$$

□

4.2 Uma 2-aproximação para o caso com número de classes constante

Nesta seção apresentaremos uma simples modificação do algoritmo OSP, apresentado por Fekete *et al.* [17] e que é estudado na Seção 3.2 deste trabalho.

Vimos que o algoritmo *online* OSP é utilizado com instâncias tais que os itens são quadrados. Nesta Seção este algoritmo é alterado com o objetivo de garantir resultados de aproximação para retângulos. Mostramos que se redefinirmos os tamanhos dos itens e ordenarmos a lista L por ordem não crescente de largura, conseguimos uma 2-aproximação assintótica para o problema na sua versão com orientação fixa (resultado igual ao do algoritmo *NFDH*, porém com aproximação puramente baseada no preenchimento de área). A este algoritmo demos o nome de *Slot Class* (SC). A ideia básica do algoritmo *Slot Order* (SO) apresentado nesta Seção é empacotar cada classe, em ordem, com este algoritmo modificado SC. Com isto conseguiremos um fator de aproximação 2 para o problema, porém com um fator adicional de t, o número de classes da instância. Isto é um problema, pois desta forma o algoritmo pode ter aproximação arbitrariamente ruim. Entretanto, assumindo que t seja limitado superiormente por uma constante, uma restrição que é razoável na prática, aí sim teremos uma 2-aproximação assintótica.

Primeiramente apresentamos o algoritmo SC (Algoritmo 4.3). Este funciona de maneira bem simples: primeiramente redimensiona a largura dos itens e ordena a lista L de entrada em ordem não crescente de largura, depois utiliza o algoritmo OSP para gerar o empacotamento com estes itens.

Algoritmo 4.3 *Slot Class* - SC

1: **input:** Uma lista de itens L.
2: **begin**
3: Ordene a lista de itens L em ordem não crescente de largura.
4: Para todo item a_i, seja k tal que $2^{-k-1} < w(a_i) \leq 2^{-k}$, faça $w(a_i) = 2^{-k}$
5: Seja P o empacotamento gerado pelo algoritmo OSP(L), considerando que $l(a_i) = w(a_i)$.
6: **return** P com os itens em suas larguras originais.
7: **end**

Como já mencionado, este algoritmo é utilizado como rotina no algoritmo SO (Algoritmo 4.4). Este algoritmo também é simples: No início, os itens são particionados em conjuntos representando cada classe. Após isto, cada conjunto é empacotado com o algoritmo SC. Para facilitar a análise, considera-se que um nível é criado para cada classe empacotada. Logo, ao fim de cada execução do algoritmo SC, uma divisória horizontal é traçada acima do item que determina a altura do empacotamento até o momento.

Algoritmo 4.4 *Slot* - SO

1: **entrada:** Uma lista de itens L.
2: **begin**
3: Particione os itens de L pela sua classe em conjuntos $c_1, \ldots, c_t$.
4: **for** $(i = 1$ to $t)$ **do**
5: Seja P_i o empacotamento gerado pelo algoritmo SC(c_i).
6: Empacote P_i no nível atual e feche este nível.
7: **return** altura total da faixa.
8: **end**

O algoritmo SO claramente é viável do ponto de vista da restrição do problema SPU, já que cada nível contém itens de apenas uma classe e estes estão ordenados em ordem não crescente de classe.

4.2.1 Análise do Algoritmo SO

Primeiramente, a complexidade do algoritmo é dominada pelo algoritmo SC que por sua vez, tem complexidade semelhante a do algoritmo OSP, ou seja, $O(n^3)$.

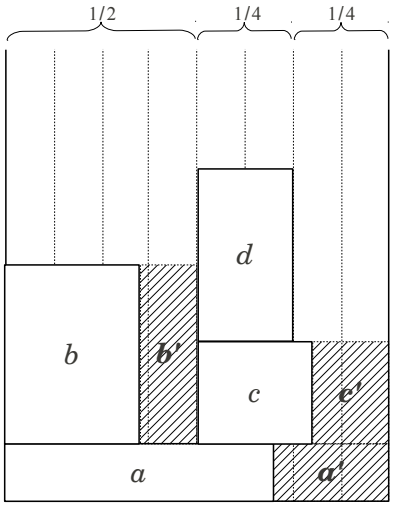


Figura 4.3: Representação dos itens empacotados pelo algoritmo SC.

Quanto ao fator de aproximação, vamos primeiramente provar que o algoritmo SC garante uma ocupação de área de pelo menos 1/2 exceto por uma unidade de altura no topo do empacotamento. Assumindo isto, mostraremos que o algoritmo SO é 2-aproximado com uma constante aditiva limitada por t, o número de classes do problema.

Considere que a_i, o i-ésimo item empacotado pelo algoritmo OSP, foi empacotado em um *slot* S_i de largura 2^{-k_i}, onde $2^{-k_i-1} < w(a_i) \leq 2^{-k_i}$. Defina a_i^C (o complemento de a_i) como a área obtida pelo aumento da largura de a_i em $2^{-k_i} - w(a_i)$. Faça $a_i^P = a_i \cup a_i^C$ e, portanto, podemos notar que a_i^p é um retângulo com $w(a_i^P) = 2^{-k_i}$, completamente contido no *slot* S_i (Ver Figura 4.3). Nesta figura, temos os itens a, b, c e d, cada qual com sua representação do complemento denotada por a' para facilitar a visualização da Figura, logo, neste caso $a^P = a \cup a'$ por exemplo. Note que $w(a_i^P) < 2w(a_i)$. Para cada item a_i, defina $\beta(a_i) = \{a_j^P \mid$ *a base de* a_j^P *intercepta o topo de* a_i^P *em mais de um ponto*$\}$. Logo, $\beta(a_i)$ pode ser visto como o conjunto de itens que "cobrem" a_i^P (Ver Figura 4.3). Nesta Figura podemos ver que $\beta(a) = \{b, c\}$, $\beta(c) = \{d\}$ e $\beta(b) = \beta(d) = \varnothing$.

Ao final do algoritmo SC, antes de retornar a largura original dos itens, considere os itens a_i^P ao invés dos itens a_i. Nos próximos lemas (4.2.1 e 4.2.2) provaremos duas assertivas que serão utilizadas para provar a aproximação do algoritmo SC no Teorema 5. Considere que a *altura induzida* por um item a_i é definida como $y(a_i) + h(a_i)$.

Lema 4.2.1. *No empacotamento gerado pelo algoritmo SC, todo item* a_i^P *ou está sobre a base da faixa* S *ou está empacotado sobre um único item* a_j^P.

Demonstração. Se a_i^P estiver empacotado sobre a base de S a prova segue trivialmente, senão, caso a_i^P esteja sobre pelo menos um item a_j^P então $w(a_i) \leq w(a_j)$ já que os itens são empacotados em ordem decrescente de largura. Por um lado, se $w(a_i^P) = w(a_j^P)$, então como a_i está sobre a_j vale que $S_i = S_j$, ou seja a_i está empacotado no mesmo slot de a_j, e como os itens possuem largura igual a do slot o lema segue. Por outro lado, se $w(a_i^P) < w(a_j^P)$ então S_j possui $\frac{2^{-k_j}}{2^{-k_i}}$ *slots* de largura $w(a_i^P)$ e, além disto, todos estão totalmente contidos no *slot* S_j por definição, portanto S_i está totalmente contido em S_j e portanto a_i está sobre somente o item a_j. □

Lema 4.2.2. *Para cada i, $1 < i \leq n$, após ser empacotado o item a_i, a faixa está totalmente preenchida até pelo menos a altura $y(a_i)$ pelos itens $a_{i'}^P$, $1 \leq i' \leq i-1$.*

Demonstração. Por contradição, escolha a_i como o primeiro item que foi empacotado, tal que a faixa não está totalmente preenchida pelos itens $a_{i'}^P$, $i' < i$, até $y(a_i)$. Logo, isto faz com que exista um item a_j^P tal que $y(a_j)+h(a_j) < y(a_i)$ e que não está totalmente coberto, ou seja, $\sum_{a_k^P \in \beta(a_j)} w(a_k^P) < w(a_j^P) = w(S_j)$. Seja a_k^P o último item empacotado sobre a_j^P, então $w(S_j) \geq w(S_k) \geq w(S_i)$. Logo, se a_k ocupou o último *slot* de largura $w(S_k)$ contido em S_j então temos uma contradição pois a_j^P estaria totalmente coberto. Desta forma, temos pelo menos um *slot* de largura $w(S_k)$ acima de a_j^P e como $w(S_k) \geq w(a_i^P)$ então a_i pode ser empacotado completamente dentro do slot S_j e em uma altura menor que $y(a_i)$ o que é uma contradição. □

Teorema 5. *Seja L uma lista de retângulos, então vale que $SC(L) < 2OPT(L) + 1$.*

Demonstração. Seja a_i o item que induz a altura do empacotamento final, ou seja, $\mathrm{SC}(L) = y(a_i)+h(a_i)$. Então pelo Lema 4.2.2, até a altura $y(a_i)$ a faixa está totalmente preenchida pelos itens a_j^P. Logo, observa-se que ao retornar um item ao seu tamanho original removemos, no máximo metade da área deste, pois, para qualquer item a_i, vale que $2^{-k_1} < 2w(a_i)$ e a altura permanece inalterada. Desta forma

$$\mathrm{SC}(L) - h_i \;\leq\; \sum_{i=1}^{n} w(a_i^P) \cdot h(a_i^P) < \sum_{i=1}^{n} 2 \cdot w(a_i) \cdot h(a_i) = 2 \cdot \sum_{i=1}^{n} w(a_i) \cdot h(a_i) \tag{4.4}$$

e como a altura de um item é limitado por 1, concluímos que

$$\mathrm{SC}(L) \;<\; 2OPT(L) + 1.$$

□

Com base no Teorema 5 provamos o Teorema 6. A ideia básica é garantir a área de empacotamento em cada nível gerado.

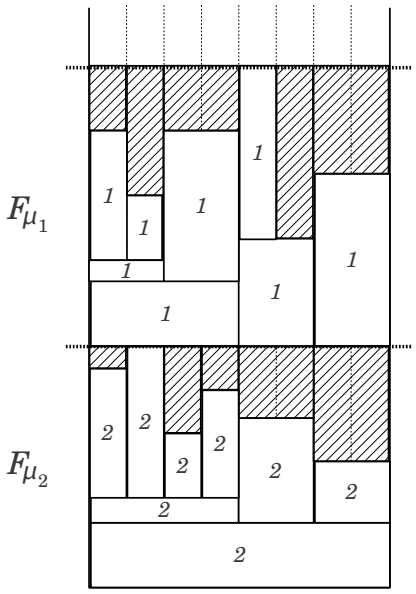


Figura 4.4: Representação dos Níveis F_{μ_i} no algoritmo SO.

Teorema 6. *Seja L uma lista de retângulos particionados em t classes, então vale que $SO(L) < 2OPT(L) + t$.*

Demonstração. Pelo Teorema 5, para cada nível gerado no algoritmo 4.4 para uma determinada classe i de itens, vale que ele está totalmente preenchido pelos itens a_j^P, exceto pela "última unidade de altura do nível". Neste caso, delimitaremos a área de cada nível para uma classe i como uma faixa horizontal F_{μ_i}, $1 \leq i \leq t$. Denotaremos a área não preenchida pelos itens a^P por N_{μ_i} para cada faixa F_{μ_i} (Veja a Figura 4.4 com um exemplo para uma instância com duas classes apenas). Pelo Teorema 5, podemos ver que $N_{\mu_i} \leq 1$, $1 \leq i \leq t$ (representados pela área hachurada na Figura 4.4).

Como temos um número t de classes, temos o mesmo número t de níveis. Desta forma

$$\mathrm{SO}(L) \;<\; \sum_{i=1}^{n} w(a_i^P) \cdot h(a_i^P) + \sum_{i=1}^{t} N_{\mu_i} \leq 2 \cdot \sum_{i=1}^{n} w(a_i) \cdot h(a_i) + t \tag{4.5}$$

e portanto

$$\mathrm{SO}(L) \;<\; 2OPT(L) + t$$

□

Portanto, para instâncias onde o número de classes é limitado superiormente por uma constante temos uma 2-aproximação assintótica.

4.2.2 Heurísticas no Algoritmo SO

O algoritmo SO da maneira como está definido em 4.4 é 2-aproximado, porém com o termo t adicional. Em situações práticas este fator adicional pode ser crítico. Considere por exemplo o caso em que há apenas um item por classe (cliente). Neste caso, o algoritmo simplesmente empilha os itens alinhados à esquerda na faixa S. Portanto, fizemos duas modificações para melhorar esta questão prática do algoritmo. Para os casos onde os itens podem ser rotacionados, a mesma garantia de aproximação é válida, já que esta é baseada apenas na área dos itens.

A primeira modificação foi simplesmente rotacionar os itens de forma que $w(a_i) \geq h(a_i)$. Neste caso, tentamos diminuir o impacto do termo t. Outra modificação realizada é não dividir as classes em níveis, considerando um empacotamento semelhante ao do algoritmo OSP. Neste caso, o algoritmo é semelhante ao U_{OSP} na Seção 3.2.2, porém com os itens estendidos e com uma ordem de largura nos itens. Note que aqui também, o limite superior de aproximação continua válido pois o empacotamento gerado desta forma não é pior do que o algoritmo original.

Nos testes realizados neste trabalho consideraremos as versões SO^u e SO^{ur} para os casos sem e com rotações, respectivamente. O algoritmo SO^u com apenas a segunda modificação e o SO^{ur} com as duas.

Estes algoritmos possuem o mesmo fator de aproximação do SO.

4.3 Heurísticas *GRASP*

Nesta Seção descrevemos duas heurísticas *GRASP* para o problema SPU, uma para a versão com orientação fixa (G) e outra para a versão com rotações ortogonais (G_r). As heurísticas são semelhantes, modificando-se apenas a orientação escolhida para os itens na versão com rotações. Por isto, descrevemos primeiramente as etapas da heurística e depois as especializações para os problemas considerados. As heurísticas propostas são baseadas na heurística *GRASP Reativa* proposta por Alvarez *et al.* [1] para o problema *Strip Packing* sem rotações. Apresentamos primeiro o algoritmo construtivo e depois o algoritmo de busca local, para só então definir a heurística para cada problema.

4.3.1 Algoritmo Construtivo

O algoritmo construtivo utiliza faixas verticais para representar possíveis posições de empacotamento dos itens. Diferentemente dos *slots* vistos até agora, os quais possuem largura fixa, as faixas utilizadas neste algoritmo têm sua largura definida pela largura dos itens já empacotados. Neste algoritmo não são feitas rotações nos itens.

O algoritmo construtivo (Algoritmo 4.5) utiliza duas listas: L, para guardar os itens que ainda não foram empacotados e E para representar os espaços onde os itens podem ser empacotados em um determinado instante. Cada espaço e é representado pela tupla $(y(e), w(e), c(e))$, onde $y(e)$ é a altura (nível) do espaço, $w(e)$ é a largura e $c(e)$ é o valor mínimo de classe de um item empacotado abaixo deste espaço, para ajudar a lidar com a restrição de descarregamento. Inicialmente a lista L contém todos os itens da entrada e a lista E contém apenas a base da faixa S, ou seja, $E = \{< 0, w(S), \infty >\}$ (linha 3). A cada rodada o algoritmo constrói a sua *lista restrita de candidatos* (LRC) com os itens que provavelmente serão empacotados nestes espaços sem inviabilizar o empacotamento do ponto de vista da restrição de descarregamento (linhas 5 a 7). Isto é feito primeiramente com uma lista auxiliar LRC^* contendo itens a_i tal que para cada a_i vale que quando somamos as larguras de itens com classe maior do que a classe de a_i, esta largura é menor ou igual a um fator ρ (linha 5), o qual será discutido na Seção 4.3.2. A LRC é formada a partir da LRC^* selecionando os itens que podem ser empacotados na posição mais baixa possível, sem inviabilizar a solução (linhas 6 e 7). Neste caso, a posição mais baixa possível pode ser formada por uma sequência de espaços de alturas diferentes, de forma que estes espaços consigam acomodar ao menos um item da lista LRC^*. Então de maneira aleatória, com probabilidade proporcional a largura dos itens, selecionamos um item da LRC e o empacotamos na posição mais baixa possível sem inviabilizar a solução parcial, utilizando o Algoritmo 4.6 para decidir a posição do item dentro do espaço E' (linhas 8 a 11). Por fim, as listas L e E são atualizadas (linhas 12 a 14). Este processo é repetido até que $L = \varnothing$.

Na Figura 4.5 podemos ver um exemplo de uma rodada do algoritmo construtivo. Na parte (a) temos alguns itens de classes 7, 8, 9 e 10 já empacotados. Na lista L restam itens de classe 1 a 8, porém apenas 3 deles satisfizeram a seleção para a lista LRC^*, os quais são exibidos na parte (b). Dentre os itens em LRC^*, podemos ver que as posições mais baixas de empacotamento podem ser alcançadas pelos itens de classe 7 e 8, utilizando $E' = e_3, e_4, e_5$ e, por isto a LRC conterá apenas os itens da classe 7 e 8. Depois, em (c.), o item de classe 8 é selecionado aleatoriamente e empacotado em E'. Nas partes (a) e (c.) podemos ver também a representação dos espaços e_i. O espaço e_1 por exemplo, é representado por $e_i =< y(a_7) + h(a_7), w(a_7), 7 >$, onde a_7 é o único item de classe 7 empacotado.

O algoritmo construtivo é apresentado em 4.5.

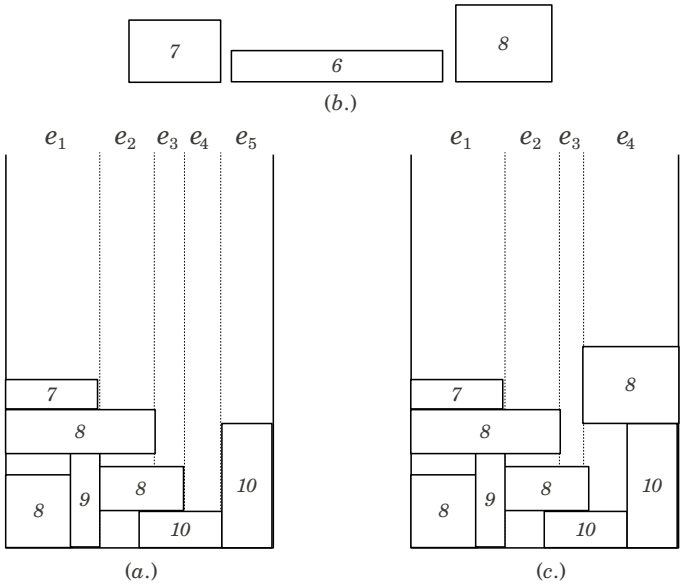


Figura 4.5: Representação e escolhas do algoritmo construtivo.

Algoritmo 4.5 Algoritmo Construtivo

1: **input:** Uma lista de itens L.
2: **begin**
3: $E = \{< 0, W, \infty >\}$.
4: **while** $L \neq \varnothing$ **do**
5: $LRC^* \leftarrow \{a_i | \sum_{c(a_k)>c(a_i)} w(a_k) \leq \rho\}$, onde $a_k \in L$.
6: Seja $E' \subseteq E$ o conjunto de espaços subsequentes que permitem a algum item $a_i \in LRC^*$ ser empacotado o mais baixo possível (Respeitando a restrição de descarregamento).
7: $LRC \leftarrow \{a_i | a_i \in LRC^*$ e $w(a_i) \leq \sum_{e \in E'} w(e)\}$.
8: Selecione um item $a_s \in LRC$ com probabilidade $p = \frac{w(a_s)}{\sum_{a_k \in LRC} w(a_k)}$.
9: Denote por $w(E') = \sum_{e \in E'} w(e)$, $l_{max} = max\{l(e) | e \in E'\}$, e $c_{min} = min\{c(e) | e \in E'\}$.
10: Selecione a posição $\{esquerda, direita\}$ do item a_s no espaço $< l_{max}, w(E'), c_{min} >$ utilizando o procedimento *escolha* (Algoritmo 4.6).
11: Empacote a_s na posição selecionada.
12: $L = L - \{a_s\}$.
13: $E = E - E' + \{< l_{max} + h(a_s), w(a_s), c(a_s) >\}$.
14: Se $w(E') > w(a_s)$ então um segundo espaço $e' =< l_{abaixo}, w(E') - w(a_s), c_{abaixo} >$ deve ser inserido em E, onde *abaixo* é o espaço que não foi totalmente coberto por a_s.
15: **end**

Algoritmo 4.6 Escolha

1: **input:** Um item a_i, uma lista E de espaços e E' a lista de espaços sobre os quais a_i será empacotado.
2: **begin**
3: Seja e_l e e_r os espaços imediatamente a esquerda e a direita de E' respectivamente.
4: **if** $e_r = \varnothing$ **then**
5: Retorne *direita*.
6: **if** $e_l = \varnothing$ **then**
7: Retorne *esquerda*.
8: **if** $y(e_r) = max_{y(e)}\{e \in E'\} + h(a_i)$ **then**
9: Retorne *direita*.
10: **if** $y(e_l) = max_{y(e)}\{e \in E'\} + h(a_i)$ **then**
11: Retorne *esquerda*.
12: **if** $y(e_r) = y(e_l)$ **then**
13: Retorne a posição em que a_i fica mais próximo de um dos lados de S.
14: **if** $y(e_r) < y(e_l)$ **then**
15: Retorne *esquerda*.
16: Retorne *direita*.
17: **end**

Além da estratégia escolhida para o valor a probabilidade p (linha 8 do Algoritmo 4.5) algumas outras abordagem foram testadas. O valor da ordem poderia ser combinado com valores de altura ou largura afim de dar mais prioridade a itens que vão ter um menor impacto na restrição de descarregamento. Por exemplo $p = \frac{c(a_i)+kw(a_i)}{\sum_{a_k \in LRC} c(a_i)+kw(a_i)}$, onde k é uma constante, mostrou alcançar resultados promissores, porém não tão bons quanto a escolha simples pela largura.

O algoritmo construtivo seria determinístico se sempre escolhêssemos o item com maior probabilidade p.

4.3.2 Escolhendo o valor de ρ

O valor da variável ρ pode ser interpretado como um grau de liberdade da heurística com relação a restrição de descarregamento, pois quanto maior o valor de ρ, maior será a probabilidade de um item de classe baixa ser empacotado antes dos seus sucessores (na ordem remoção dos itens). Por exemplo, se usarmos esse fator como 0 induziremos o empacotamento dos itens em ordem decrescente do valor de classe. Note que a estratégia para o valor ρ determina a qualidade da lista LRC, um fator primordial na heurística.

Vários experimentos foram realizados para descobrir o melhor valor de ρ. Primeiro nós testamos as seguintes estratégias para construir a lista LRC:

- *LRC* contendo dentre os itens ainda não empacotados, apenas aqueles com o maior valor de classe. Este é o mais conservador.
- *LRC* contendo dentre os itens ainda não empacotados, apenas aqueles com o maior e o segundo maior valor de classe.
- *LRC* contendo dentre os itens ainda não empacotados, aqueles com o maior valor de classe e cada item com o segundo maior valor de classe selecionado com 50% de probabilidade.

Os testes mostraram que o modelo conservador é a melhor estratégia de todas. A partir disto pudemos ver que itens das classes menores deveriam ser escolhidos cuidadosamente com uma estratégia melhor, já que podem facilmente tornar o empacotamento inviável.

Assim, nas próximas três estratégias usamos o valor de ρ que impõe um limitante para a máxima largura aceitável de itens que podem bloquear um item a_i em questão.

- $\rho = w(S) - w(a_i)$.
- $\rho = \frac{w(S)-w(a_i)}{2}$.
- $\rho = \lambda(w(S) - w(a_i))$. Onde λ é um valor aleatório que foi testado em vários intervalos.

Estas estratégias buscam selecionar itens que provavelmente não criarão soluções inviáveis. Por exemplo, a primeira solução impõe que um item a_i seja selecionado se a soma das larguras dos itens das classes maiores que $c(a_i)$ mais $w(a_i)$ é menor do que a largura da faixa $w(S)$. Para a terceira estratégia usamos quatro distribuições uniformes para λ ($\{[0.1; 0.9], [0.2; 0.8], [0.3; 0.7], [0.4; 0.6]\}$) e a melhor foi $\lambda \in [0.4; 0.6]$. Nos testes realizados para a escolha dos parâmetros, a segunda estratégia atingiu os melhores resultados de forma consistente.

4.3.3 Busca Local

Esta fase é aplicada a cada solução gerada pelo algoritmo construtivo, tendo como objetivo melhorar o valor da solução encontrada.

Testes computacionais demonstraram que o algoritmo construtivo tem um bom desempenho por si só, porém seu maior problema ocorre nos últimos itens empacotados, os quais geralmente aumentam muito o valor da solução. Em geral, qualquer escolha mal feita na etapa final do algoritmo pode influenciar muito negativamente a qualidade da solução, pois não há itens o bastante para preencher espaços gerados pelo empacotamento de novos itens.

Para resolver este problema, utilizamos uma estratégia semelhante a proposta por Burke *et al.* [12] onde eles utilizaram a heurística *Bottom-Left* para "finalizar" (empacotar os últimos k itens da entrada) o empacotamento gerado por uma heurística *simulated annealing*. Dada uma solução gerada pelo algoritmo construtivo, a busca local gerando no máximo 3 vizinhos removendo os últimos $k\%$, $k \in \{10, 20, 30\}$, de itens empacotados, ou o mínimo para que a altura induzida pelo empacotamento diminua, e os empacota novamente utilizado o algoritmo construtivo determinístico. Por fim, é escolhida a melhor solução encontrada dentre os vizinhos e a solução original. Como a questão da restrição de descarregamento já foi tratada no algoritmo construtivo, o qual testa a viabilidade de cada empacotamento, esta escolha parece ser apropriada pois é semelhante a estratégia usada por Alvarez *et al.* [1].

Algumas outras estratégias também foram abordadas, como por exemplo impor diferentes ordens a lista de itens e empacotá-los nesta ordem assim como [45] e [22], porém, a estratégia escolhida obteve um desempenho melhor. Isto provavelmente deve-se ao fato de que, além de ter um desempenho melhor, a estratégia escolhida gera menos soluções inviáveis.

4.3.4 As Heurísticas G e G_r

Nesta Seção descrevemos as heurísticas *GRASP* baseadas nos algoritmos das Seções 4.3.1 e 4.3.3. As heurísticas basicamente aplicam o algoritmo construtivo e depois utilizam a busca para encontrar soluções melhores. A estratégia de *GRASP Reativa* não foi utilizada já que a escolha aleatória p não depende de nenhum parâmetro a ser reajustado.

A primeira heurística G é aplicada para o caso com orientação fixa. Ela segue basicamente a ideia da meta-heurística *GRASP* com os algoritmos construtivo e de busca guardando a melhor solução encontrada até o momento.

A segunda heurística G_r pode rotacionar os itens. Depois de experimentar várias maneiras para o fazer, escolhemos uma estratégia simples. O algoritmo G_r rotaciona todos os itens a_i de forma que $(h(a_i) \geq w(a_i))$ durante a fase construtiva, priorizando a restrição de descarregamento, enquanto que na fase de busca, os itens são rotacionados na posição inversa $(h(a_i) \leq w(a_i))$, priorizando a altura induzida.

Primeiramente, poderíamos tentar empacotar os itens com a melhor orientação possível, buscando minimizar a altura induzida da solução. Por outro lado, esta escolha pode piorar muito o resultado do algoritmo, por exemplo nos casos onde a instância é formada por poucos itens, altos e finos. Se pensarmos de outra forma, poderíamos tentar empacotar os itens sempre minimizando o impacto da restrição de descarregamento no empacotamento, entretanto, isto poderia gerar soluções ruins devido ao foco na restrição e não na altura induzida.

Então dentre várias abordagens testadas decidimos priorizar a restrição de ordem primeiro (na fase construtiva) e depois pensar na altura induzida (na fase de busca).

Capítulo 5

Experimentos Computacionais

Nesta seção descreveremos os testes realizados com implementações dos algoritmos LBP, SO^u, SO^{ur}, G_r e G, assim como as instâncias utilizadas nos testes e os resultados obtidos.

Os algoritmos foram implementados utilizando a linguagem C e executados em um processador Intel Core 2 Duo com 2.4 GHz. Seguindo o padrão utilizado em alguns trabalhos da literatura, utilizamos como critério de parada das heurísticas *GRASP*, um tempo limite de 60 segundos ou 1000 iterações. Os algoritmos LBP, SO^u e SO^{ur} não possuem critérios de parada já que são algoritmos que geram apenas uma solução, de maneira rápida e determinística. Além destes critérios de parada, executamos testes com um limite de apenas 5 iterações para as heurísticas G_r e G para medir a eficiência em casos onde elas necessitem executar muito rápido (como nos problemas 2L-CVRP).

Três limitantes inferiores diferentes foram usados para o problema SPU já que não conhecemos valores ótimos para as instâncias consideradas. O primeiro é o limitante trivial de área utilizado para alcançar a aproximação dos algoritmos LBP e SO. O segundo é o valor do ótimo para o *Strip Packing Problem*, pois para algumas instâncias este valor é conhecido. Por fim, podemos analisar a estrutura dos itens afim de encontrar um limitante inferior que considere a ordem dos itens. Se um par de itens a_i e a_j possuírem $c(a_i) \neq c(a_j)$ e $w(a_i)+w(a_j) > w(S)$, então $h(a_i)+h(a_j)$ é um limitante para o custo da solução. Como existe a ordem entre os itens, utilizamos também como limitante inferior a sequência mais alta (soma das alturas) de itens que devem ser obrigatoriamente empacotados um acima do outro. Denotaremos este último limitante inferior por lb_3. Além disto, nos experimentos realizados sempre utilizamos o valor máximo dentre os três limitantes citados.

Dois tipos de medidas de qualidade foram utilizadas:

- A ocupação média da faixa S (Nestes casos as heurísticas *GRASP* foram limitadas a apenas 5 iterações).
- A razão de aproximação com relação ao limitante do ótimo, dada pela altura da

solução encontrada sobre o limitante inferior para o ótimo.

Os algoritmos foram aplicados em 8 conjuntos de instâncias $(S_1, \ldots, S_8)$. Um dos conjuntos foi adaptado das instâncias utilizadas nos trabalhos relacionados ao problema clássico do 2L-CVRP. Nestas instâncias removemos as informações de roteamento e usamos apenas informações sobre dimensões dos itens e seus valores de ordem. Os 7 conjuntos restantes foram adaptados de instâncias para o *Strip Packing Problem* vastamente utilizadas na literatura. Neste caso, particionamos sistematicamente os itens das instâncias em classes a fim de reutilizá-los no problema SPU.

O conjunto S_1 foi retirado do conjunto de instâncias para o 2L-CVRP utilizado em [29, 22, 45, 21]. Cada instância deste conjunto é formada por um grafo de rotas e os itens que devem ser entregues aos clientes neste grafo. Desta forma, removemos de cada instância as informações de roteamento e utilizamos apenas os dados relacionados ao problema de empacotamento (os itens com suas dimensões e clientes). As 180 instâncias disponíveis foram divididas em 5 subconjuntos de acordo com o número de clientes, itens e suas respectivas dimensões. Cada subconjunto $S_{11}, S_{12}, \ldots, S_{15}$ é formado por 36 instâncias com características semelhantes. Cada instância contém entre 15 e 255 clientes e entre 15 e 786 produtos. Para maiores informações sobre cada classe e o conjunto em geral veja [28]. Este conjunto tem uma importância diferenciada pois é adaptado de instâncias do 2L-CVPR e pode ser adquirido em `http://www.or.deis.unibo.it/research.html`.

Das instâncias para o *Strip Packing Problem* encontradas na literatura, selecionamos 7 conjuntos. Para cada instância destes 7 conjuntos geramos 10 tipos de instâncias variando o número de classes dado por $\lceil \frac{kn}{10} \rceil$ classes (clientes), para $k = 1, 2, \ldots, 10$. Com isto podemos medir o impacto do aumento do número de classes no resultado do algoritmo. Por exemplo, para uma instância com 200 itens foram criadas 10 tipos de instâncias com, $20, 40, \ldots, 180$ e 200 classes. Depois, geramos aleatoriamente 10 instâncias de cada um destes tipos, com pelo menos um item por classe, ou seja, para cada instância da literatura, geramos 100 cópias diferentes. Além disto, mantivemos os valores originais de largura da faixa S.

Os conjuntos $S_2, \ldots, S_8$ são:

- S_2 contendo de 3 instâncias criadas por Christofides *et al.* [14].
- O conjunto S_3 com 12 instâncias geradas por Burke *et al.* [11], com tamanho entre 10 e 500 itens.
- O conjunto S_4 composto por 10 instâncias proposto por Bengtsson [9]. Cada instância contendo entre 25 e 200 itens.
- S_5 com 21 instâncias criadas por Hopper*et al.* [26]. Cada instância contendo entre 16 e 197 itens.

- O conjunto S_6 com 420 instâncias geradas a partir do método proposto por Wang *et al.* [42]. Este conjunto é particionado em 2 subconjuntos: S_{61} (com tamanhos e formatos semelhantes) e S_{62} (com grandes variações tanto nos tamanhos como nos formatos), com 210 instâncias cada, variando de 25 a 500 itens.

- O conjunto S_7 com 70 instâncias proposto por Hopper [27]. Este conjunto também é particionado em 2 subconjuntos de tamanhos iguais, o primeiro S_{71} (guilhotinado) e o segundo S_{72} (não guilhotinado) com tamanhos entre 17 e 199 itens.

- S_8 com 16 instâncias proposto por Beasley [6] [7]. Assim como no conjunto S_7 as instâncias são divididas em subconjuntos S_{81} (guilhotinado) contendo 4 instâncias com tamanho 250 e S_{82} (não guilhotinado) formado por 12 instâncias com tamanhos entre 17 e 22.

No total geramos 55380 instâncias a fim de comparar e demonstrar a eficiência dos algoritmos propostos no trabalho.

Quanto ao tempo de processamento, em cerca de 90% das instâncias os algoritmos LBP, SO^u e SO^{ur} executaram em menos de 1 segundo. As exceções são algumas instâncias do conjunto S_1 com mais de 500 itens onde o algoritmo levou entre 1 e 2 segundos. Por outro lado as heurísticas *GRASP* foram limitadas pelo tempo (60s) em instâncias a partir de 400 itens. Resultados mais detalhados podem ser encontrados no Apêndice A.3.

Nas tabelas de resultados apresentadas nesta seção, nós apresentamos os valores médios de aproximação e preenchimento por subconjunto de instâncias estudado. Isto pode esconder alguns resultados ruins para certos subconjuntos. Porém devido ao número de instâncias, deixamos os resultados mais detalhados para o Apêndice A.3. Com as medidas apresentadas nesta seção podemos avaliar o desempenho geral dos algoritmos, além de ver como a quantidade de classes (clientes) influencia o custo das soluções.

Subconjunto	LBP	SO^{ur}	G_r	SO^u	G
S_{11}	25.984125	1.226453	1.000000	1.226453	1.000000
S_{12}	2.354272	1.572328	1.119889	1.565922	1.163831
S_{13}	2.214308	1.582067	1.108992	1.579744	1.146797
S_{14}	2.043642	1.598569	1.107019	1.584100	1.140322
S_{15}	1.943778	1.492667	1.097533	1.506039	1.127356

Tabela 5.1: Aproximação dos algoritmos LBP, SO^{ur}, G_r, SO^u e G para o conjunto S_1

As tabelas 5.1 e 5.2 apresentam os resultados para o conjunto S_1. O valor anômalo para o algoritmo LBP no subconjunto S_{11} pode ser explicado pelas peculiaridades das instâncias deste conjunto. Nestas o conjunto de itens é formado por n quadrados de lado

Subconjunto	LBP	SO^{ur}	G_r	SO^u	G
S_{11}	5.0833%	63.3611%	77.4531%	63.3611%	77.4531%
S_{12}	42.5556%	63.5278%	86.2672%	63.7778%	80.6281%
S_{13}	45.3611%	63.0556%	86.7%	63.25%	82.755%
S_{14}	48.9722%	62.4167%	86.915%	63.1667%	83.5581%
S_{15}	51.7778%	66.7778%	88.1308%	66.3611%	85.8278%

Tabela 5.2: Ocupação média de área dos algoritmos LBP, SO^{ur}, G_r, SO^u e G para o conjunto S_1

1 enquanto que a largura e altura da faixa S é bem maior que isto. Logo o algoritmo, após rotacionar os bins, gera pilhas de itens pequenos, enquanto o ótimo seria empacotá-los lado a lado. Mesmo assim, o algoritmo LBP ainda garante o fator de aproximação pois utiliza apenas um bin.

Para o conjunto S_1 as heurísticas *GRASP* obtiveram consistentemente os melhores resultados. Além disto, para as instâncias do subconjunto S_{11} o valor ótimo sempre foi encontrado, apesar da média de preenchimento de área ser de apenas 77.4531%. As instâncias do subconjunto S_{12} possuem muitos itens "muito altos e muito finos" ou "muito baixos e muito largos". Logo, obtiveram piores resultados de aproximação pois não encontramos um bom valor de limitante inferior.

Apesar do limitante de área ser o mais utilizado nos testes, o limitante lb_3 obteve resultados muito bons em algumas instâncias. Considere por exemplo, a instância *N1Burke* [11] que possui uma faixa de largura 40 e 10 itens. Neste teste com um item por classe, geramos aleatoriamente a instância da Tabela 5.3. é fácil ver que o limitante de área desta instância é 40 já que $\sum_{i=1}^{10} h(i)w(i)/40 = 40$. Entretanto, se utilizarmos o limitante lb_3, veremos que os itens de ordem 1, 4 e 5 devem ser empacotados exatamente nesta ordem, um acima do outro, devido a suas larguras e classes. Logo, podemos limitar a altura mínima por $h(a_1) + h(a_4) + h(a_5) = 46 > 40$ (Veja a Figura 5.1 parte (a), onde os itens são representados pelo seu valor $c(a_i)$). Na parte (b) da Figura 5.1 vemos um empacotamento ótimo com valor igual ao do limitante inferior lb_3. Nas partes (c), (d), (e) e (f) da mesma figura vemos os resultados dos algoritmos G, LBP, SO^{ur} e SO^u, respectivamente. O resultado da heurística G_r foi semelhante ao da G. Além disto, esta instância nos mostra que se aplicarmos uma simples heurística de gravidade ao algoritmo LBP teremos um resultado comparável aos dos outros algoritmos. é interessante notar também que para esta instância específica, o algoritmo SO^{ur} foi superado pelo SO^u.

O limitante lb_3 foi bom para instâncias "patológicas" ou instâncias com itens relativamente grandes se comparados a largura da faixa S. Há instâncias do subconjunto S_{62} para as quais o limitante de área vale 200, enquanto que o limitante lb_3 vale 304. Nestes casos,

uma solução que possuía valor 1.5450 de aproximação utilizando a área como limitante, mostrou-se ser praticamente ótima, alcançando razões menores que 1.02 quando utilizamos o limitante lb_3. Entretanto, o limitante lb_3 só obteve bons resultados para versões do problema sem rotações.

item	$c(a_i)$	$h(a_i)$	$w(a_i)$
a_1	1	6	7
a_2	2	6	7
a_3	3	4	4
a_4	4	16	40
a_5	5	24	24
a_6	6	20	4
a_7	7	20	5
a_8	8	4	5
a_9	9	8	7
a_{10}	10	4	7

Tabela 5.3: Instância *N1Burke* gerada.

De um modo geral, os conjuntos S_8 (tabelas 5.4 e 5.5) e S_2 (tabelas 5.10 e 5.11) foram mais incisivamente influenciados pelo aumento do número de classes. Nestes casos o preenchimento da faixa S alcançado diminui a medida que o número de classes aumenta. Nos outros conjuntos isto ocorreu de maneira muito discreta ou mesmo o inverso, como ocorreu algumas vezes para o algoritmo LBP.

As heurísticas *GRASP* obtiveram os melhores resultados de maneira consistente e o algoritmo LBP os piores. Além disto, os melhores resultados ocorreram com o conjunto S_4 e os piores para o conjunto S_6. Isto ocorreu devido ao formato dos itens em S_6 ser muito variado e "patológico" como citado na literatura e isto dificulta a busca por bons limitantes, apesar do limitante inferior lb_3 ter encontrado alguns valores muito bons para este conjunto.

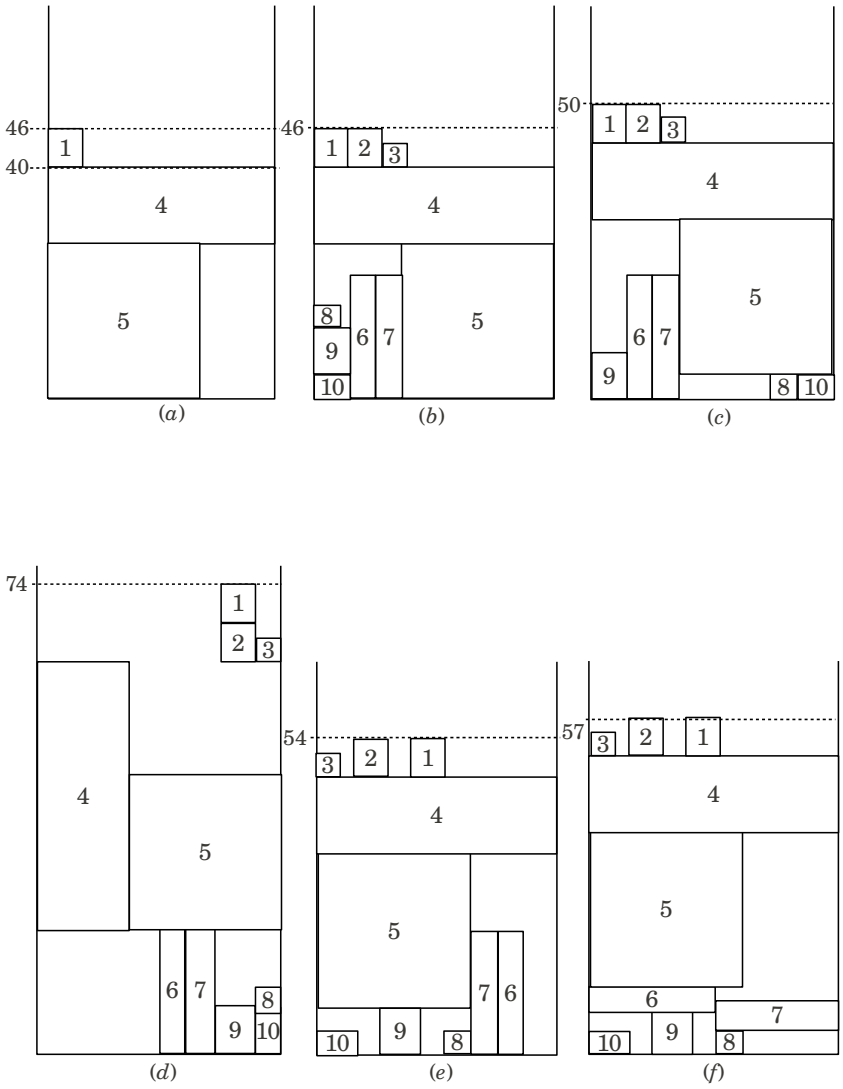


Figura 5.1: Exemplo de aplicação do limitante inferior lb_3 e soluções encontradas pelos algoritmos.

% de classes	LBP	SO^{ur}	G_r	SO^u	G
10%	1.513673	1.354004	1.054463	1.569507	1.156282
20%	1.496253	1.387987	1.075605	1.658317	1.187746
30%	1.572955	1.415532	1.081815	1.702120	1.227620
40%	1.567958	1.423347	1.094264	1.723947	1.235946
50%	1.589539	1.434959	1.105973	1.746141	1.234141
60%	1.552563	1.432957	1.099326	1.764682	1.215088
70%	1.568864	1.433089	1.104802	1.742528	1.212644
80%	1.552544	1.441044	1.104891	1.750084	1.183001
90%	1.568234	1.439814	1.114307	1.750717	1.174322
100%	1.577692	1.443485	1.112562	1.731063	1.142481

Tabela 5.4: Aproximação dos algoritmos LBP, SO^{ur}, G_r, SO^u e G para o conjunto S_8

% de classes	LBP	SO^{ur}	G_r	SO^u	G
10%	68.4938%	74.2375%	86.4697%	64.225%	79.5776%
20%	68.4563%	72.625%	85.6678%	60.7875%	76.4525%
30%	64.1813%	71.275%	84.2036%	59.1125%	74.887%
40%	64.4313%	70.8563%	82.9284%	58.525%	72.8404%
50%	63.3063%	70.35%	83.1173%	57.7125%	72.0452%
60%	64.4375%	70.375%	82.8872%	57.2125%	71.977%
70%	64.375%	70.5063%	80.9488%	57.8313%	70.2536%
80%	65.0125%	70.0938%	82.4872%	57.5125%	70.2894%
90%	64.3625%	70.1188%	81.4388%	57.5812%	69.9721%
100%	63.975%	70.0063%	81.615%	58.2188%	69.8229%

Tabela 5.5: Ocupação média de área dos algoritmos LBP, SO^{ur}, G_r, SO^u e G para o conjunto S_8

% de classes	LBP	SO^{ur}	G_r	SO^u	G
10%	1.807064	1.427667	1.079656	1.465677	1.085570
20%	1.750334	1.435489	1.081197	1.459650	1.090265
30%	1.739296	1.438347	1.079657	1.452276	1.094684
40%	1.737143	1.432140	1.083589	1.447633	1.095555
50%	1.706140	1.435915	1.085563	1.455600	1.097090
60%	1.691735	1.432658	1.085033	1.449756	1.101335
70%	1.684230	1.434967	1.085571	1.444989	1.099153
80%	1.682065	1.436586	1.084651	1.443321	1.100341
90%	1.674856	1.432122	1.089103	1.457225	1.107456
100%	1.674072	1.438388	1.089360	1.445680	1.086685

Tabela 5.6: Aproximação dos algoritmos LBP, SO^{ur}, G_r, SO^u e G para o conjunto S_4

% de classes	LBP	SO^{ur}	G_r	SO^u	G
10%	55.92%	70.02%	88.2238%	67.95%	87.0175%
20%	57.47%	69.6%	88.172%	68.23%	86.752%
30%	57.77%	69.47%	88.3605%	68.57%	87.1419%
40%	57.99%	69.7%	88.218%	68.85%	86.9921%
50%	58.77%	69.5%	87.7883%	68.42%	86.3211%
60%	59.34%	69.69%	88.3182%	68.72%	86.6574%
70%	59.62%	69.51%	88.3527%	68.92%	84.8954%
80%	59.82%	69.49%	88.6524%	69.06%	86.6638%
90%	59.88%	69.63%	86.1079%	68.36%	86.3033%
100%	59.91%	69.4%	88.0891%	68.88%	86.1528%

Tabela 5.7: Ocupação média de área dos algoritmos LBP, SO^{ur}, G_r, SO^u e G para o conjunto S_4

% de classes	LBP	SO^{ur}	G_r	SO^u	G
10%	2.079518	1.366882	1.146684	1.637426	1.209746
20%	2.036734	1.385701	1.141762	1.645387	1.226171
30%	1.986354	1.393261	1.149936	1.679248	1.238127
40%	1.994499	1.403589	1.149952	1.690075	1.226963
50%	2.015651	1.414644	1.157273	1.679643	1.246556
60%	2.051661	1.420751	1.154938	1.702259	1.241662
70%	2.002029	1.415734	1.162863	1.673464	1.244838
80%	1.994399	1.424808	1.168685	1.692001	1.220286
90%	2.013087	1.413696	1.164138	1.696155	1.238788
100%	2.010259	1.426643	1.168221	1.685716	1.215087

Tabela 5.8: Aproximação dos algoritmos LBP, SO^{ur}, G_r, SO^u e G para o conjunto S_3

% de classes	LBP	SO^{ur}	G_r	SO^u	G
10%	49.1584%	73.05%	80.4697%	61.7917%	76.4237%
20%	50.1667%	72.2333%	81.3194%	61.3833%	74.5247%
30%	51.3333%	71.8666%	81.4227%	60.1917%	74.0806%
40%	51.3583%	71.2917%	81.8119%	59.9583%	74.2154%
50%	50.6417%	70.8%	81.6731%	60.1917%	73.6146%
60%	49.7583%	70.525%	81.1147%	59.3667%	73.1193%
70%	50.7833%	70.7083%	81.4764%	60.4333%	71.6385%
80%	51.2333%	70.2167%	81.3683%	59.7083%	73.9457%
90%	50.5667%	70.8666%	79.5208%	59.7833%	73.4861%
100%	50.6667%	70.1667%	81.5709%	59.9167%	73.4415%

Tabela 5.9: Ocupação média de área dos algoritmos LBP, SO^{ur}, G_r, SO^u e G para o conjunto S_3

% de classes	LBP	SO^{ur}	G_r	SO^u	G
10%	2.022786	1.393246	1.075700	1.483997	1.093187
20%	1.845573	1.430093	1.092100	1.539447	1.133467
30%	1.818720	1.459500	1.098773	1.537643	1.138843
40%	1.772077	1.486590	1.106620	1.534057	1.144673
50%	1.685900	1.482424	1.112367	1.542603	1.149360
60%	1.719033	1.474273	1.112593	1.551353	1.152973
70%	1.677370	1.493333	1.124757	1.547436	1.156480
80%	1.729900	1.485260	1.121597	1.569720	1.142420
90%	1.656187	1.505910	1.122720	1.571000	1.134527
100%	1.727550	1.505547	1.135717	1.548137	1.077124

Tabela 5.10: Aproximação dos algoritmos LBP, SO^{ur}, G_r, SO^u e G para o conjunto S_2

% de classes	LBP	SO^{ur}	G_r	SO^u	G
10%	50.0667%	71.2666%	85.8537%	67.2333%	85.6243%
20%	54.8333%	69.5%	84.9243%	64.5333%	82.3517%
30%	55.7333%	67.9333%	84.3733%	64.8333%	81.092%
40%	57.4333%	66.6333%	84.7943%	64.9%	82.183%
50%	59.7667%	66.9667%	83.527%	64.4%	81.184%
60%	59.3667%	67.4%	82.5513%	64.1333%	80.8177%
70%	60.4333%	66.5667%	83.0067%	64.4333%	73.4638%
80%	59.6667%	66.8333%	83.145%	63.2667%	81.261%
90%	61.3667%	66.1%	75.9744%	63.2667%	80.726%
100%	59.3667%	65.9333%	82.2697%	64.2%	78.144%

Tabela 5.11: Ocupação média de área dos algoritmos LBP, SO^{ur}, G_r, SO^u e G para o conjunto S_2

% de classes	LBP	SO^{ur}	G_r	SO^u	G
10%	2.121170	1.482359	1.133008	1.670358	1.170693
20%	2.046448	1.500828	1.141887	1.680843	1.196902
30%	2.052969	1.505814	1.144158	1.688042	1.210092
40%	2.048075	1.513822	1.146557	1.684614	1.215968
50%	2.035063	1.516000	1.148515	1.687279	1.220500
60%	2.039090	1.520028	1.148351	1.688044	1.225563
70%	2.023077	1.514322	1.150422	1.692021	1.225373
80%	2.020477	1.518901	1.154508	1.687129	1.237086
90%	2.032064	1.518551	1.153922	1.691243	1.235596
100%	2.038427	1.522986	1.156572	1.691914	1.243994

Tabela 5.12: Aproximação dos algoritmos LBP, SO^{ur}, G_r, SO^u e G para o conjunto S_7

% de classes	LBP	SO^{ur}	G_r	SO^u	G
10%	47.7514%	67.5529%	82.444%	60.2743%	78.5103%
20%	49.4914%	66.8443%	82.145%	60.03%	77.0036%
30%	49.3472%	66.6344%	82.0738%	59.7885%	76.2622%
40%	49.4215%	66.3115%	82.1704%	59.9157%	75.8699%
50%	49.7829%	66.2043%	82.215%	59.81%	75.7431%
60%	49.6729%	66.0444%	82.1541%	59.7828%	75.6115%
70%	50.0471%	66.2872%	82.1854%	59.6271%	75.4836%
80%	50.07%	66.0787%	82.1054%	59.8214%	75.3816%
90%	49.8158%	66.1057%	81.939%	59.63%	75.3243%
100%	49.6729%	65.9215%	81.8746%	59.6357%	74.8345%

Tabela 5.13: Ocupação média de área dos algoritmos LBP, SO^{ur}, G_r, SO^u e G para o conjunto S_7

% de classes	LBP	SO^{ur}	G_r	SO^u	G
10%	2.108377	1.448955	1.107310	1.647837	1.146293
20%	2.152525	1.465310	1.115140	1.644740	1.167346
30%	2.140054	1.471140	1.120292	1.657825	1.188220
40%	2.112863	1.471486	1.118088	1.654788	1.187225
50%	2.124667	1.472861	1.121428	1.666356	1.196380
60%	2.116104	1.474872	1.118894	1.661680	1.202426
70%	2.137732	1.478033	1.123217	1.669851	1.212248
80%	2.171282	1.474998	1.120038	1.644333	1.208238
90%	2.128607	1.475075	1.125077	1.649717	1.214269
100%	2.137567	1.478086	1.133415	1.656293	1.204721

Tabela 5.14: Aproximação dos algoritmos LBP, SO^{ur}, G_r, SO^u e G para o conjunto S_5

% de classes	LBP	SO^{ur}	G_r	SO^u	G
10%	48.5285%	69.2572%	83.4129%	61.2048%	78.2316%
20%	47.6667%	68.481%	83.2209%	61.4762%	77.4631%
30%	48.2428%	68.2095%	83.1405%	60.9381%	77.0926%
40%	48.7762%	68.2333%	83.1642%	60.9905%	76.7528%
50%	48.419%	68.1143%	83.5299%	60.6524%	76.6433%
60%	48.6428%	68.0428%	83.2435%	61.0191%	75.9185%
70%	48.1095%	67.9238%	83.3512%	60.6238%	75.0681%
80%	47.3429%	68.0571%	83.3775%	61.4191%	75.8498%
90%	48.3667%	68.1048%	82.5436%	61.2714%	76.2034%
100%	48.2143%	67.8571%	83.1326%	60.8857%	76.2183%

Tabela 5.15: Ocupação média de área dos algoritmos LBP, SO^{ur}, G_r, SO^u e G para o conjunto S_5

% de classes	LBP	SO^{ur}	G_r	SO^u	G
10%	2.075329	1.520628	1.180391	1.790572	1.248258
20%	2.061508	1.539635	1.190671	1.784375	1.268749
30%	2.066374	1.544600	1.196927	1.771767	1.271074
40%	2.067475	1.548056	1.198655	1.771918	1.276033
50%	2.060256	1.548783	1.202151	1.773831	1.281227
60%	2.062761	1.549584	1.202551	1.766135	1.280723
70%	2.058465	1.549917	1.205042	1.767488	1.286123
80%	2.059818	1.551093	1.205060	1.769142	1.284798
90%	2.055278	1.550903	1.208117	1.770972	1.290980
100%	2.056807	1.550770	1.209893	1.772558	1.293560

Tabela 5.16: Aproximação dos algoritmos LBP, SO^{ur}, G_r, SO^u e G para o conjunto S_6

% de classes	LBP	SO^{ur}	G_r	SO^u	G
10%	48.7018%	65.7981%	78.8798%	56.9177%	74.1243%
20%	48.9963%	64.9725%	78.8043%	57.2324%	73.3525%
30%	48.9049%	64.7707%	78.6871%	57.627%	73.3073%
40%	48.8835%	64.6222%	78.6632%	57.6879%	73.2544%
50%	49.0449%	64.5842%	78.4407%	57.6112%	72.9281%
60%	48.9652%	64.5447%	78.5972%	57.8079%	73.2069%
70%	49.0712%	64.5483%	78.4757%	57.7762%	72.9974%
80%	49.0497%	64.5088%	78.5081%	57.7314%	73.0007%
90%	49.1662%	64.5039%	78.3512%	57.71%	72.8271%
100%	49.1024%	64.5112%	78.2465%	57.6225%	72.5882%

Tabela 5.17: Ocupação média de área dos algoritmos LBP, SO^{ur}, G_r, SO^u e G para o conjunto S_6

Capítulo 6

Resumo dos Resultados

Nesta Seção apresentamos um resumo dos resultados de aproximação obtidos neste trabalho além de uma discussão sobre os resultados dos testes computacionais realizados com os algoritmos aproximados e a heurística.

Os resultados de aproximação foram resumidos na Tabela 6.1. Note que nesta tabela estão contidos todos os resultados comentados durante o trabalho, mesmo as reduções apresentadas nas seções 3.1.2 e 3.2.2. Note também que os resultados para o problema SPU podem ser utilizados para o problema SPUH pois o SPU é um generalização do SPUH.

Problema	Algoritmo	Aproximação	Versão
SPUH	U_{ORP}	4	
	SO	$2 + k$	Sem Rotações
	U_{OSP}	2.6154	Quadrados
SPU	LBP	6.75	
	SO	$2 + k$	Sem Rotações
	U_{OSP}	2.6154	Quadrados

Tabela 6.1: Resumo dos resultados de aproximação.

Quanto aos resultados dos experimentos, as heurísticas *GRASP* obtiveram os melhores resultados consistentemente, mesmo nos casos onde utilizamos apenas 5 iterações como critério de parada. Por outro lado, a heurística teve um custo computacional bem mais elevado. É interessante notar que os resultados do Algoritmo LBP ficaram muito abaixo do fator de aproximação provado.

Na prática, a melhor escolha seria utilizar a heurística *GRASP* com um limite baixo de iterações, já que o algoritmo construtivo por si só, encontra boas soluções.

Capítulo 7

Conclusões

Neste trabalho estudamos problemas de empacotamento com restrições de descarregamento considerados NP-difíceis. Estes problemas possuem aplicações nas áreas de logística e roteamento. Duas versões foram estudadas, o problema SPUH onde os itens podem ser removidos da faixa utilizando-se movimentos verticais e horizontais e o problema SPU, onde utiliza-se apenas um movimento vertical para remover os itens. Estudamos alguns algoritmos para problemas de empacotamento em faixa *online* que podem ser utilizados na resolução do problema estudado. Propusemos heurísticas para os problemas e, além disto, provamos que duas destas heurísticas possuem garantias de aproximação. Os algoritmos LBP e SO propostos para o problema SPUH, possuem fator de aproximação 6.75 e $2 + k$, respectivamente, onde k é o número de classes da instâncias. Testes computacionais demonstraram que a heurística *GRASP* proposta obteve melhores resultados para as instâncias de teste consideradas.

O trabalho também abre várias direções para pesquisas futuras, tanto no campo teórico quanto prático, como por exemplo:

- **Teórico**:
  - Diminuir o fator de aproximação do algoritmo LBP: Como vimos nos resultados práticos, a razão entre o custo da solução encontrada pelo algoritmo e o custo ótimo, ficou muito abaixo do fator 6.75 provado. A priori, o fator de aproximação provado não parece ser justo, logo se buscarmos instâncias "ruins" para o algoritmos talvez vejamos onde a análise pode ser melhorada.
  - Desenvolvimento de algoritmos aproximados específicos para o SPUH: O único algoritmo apresentado para este problema foi o algoritmo proposto por [3]. Um bom ponto de partida pode ser modificar um pouco este mesmo algoritmo.
  - Desenvolvimento de algoritmos aproximados para versões restritas dos problemas SPU e SPUH: Neste trabalho, resolvemos apenas um única versão restrita

(quantidade de classes limitada por uma constante). Podemos considerar também itens de tamanho limitado, outras formas de remoção dos itens, entre outras.

- Considerar o problema 2L-CVPR do ponto de vista de aproximação: Talvez o caso métrico (onde os nós do grafo de clientes obedece a desigualdade métrica) do problema possua alguma aproximação.

- **Prático**:

  - Considerar outras meta-heurísticas: Na literatura outras meta-heurísticas foram utilizadas para o *Strip-Packing* como citado anteriormente. Podemos também considerar outros algoritmos básicos como o clássico *Bottom-Left*.
  - Testar os algoritmos com instâncias maiores: Na literatura há trabalhos que tratam instâncias com até 15000 itens [37]. Neste caso a heurística *GRASP* deve ter um limite de tempo um pouco mais alto.

Referências Bibliográficas

[1] R. Alvarez-Valdes, F. Parreño, and J. M. Tamarit. Reactive grasp for the strip-packing problem. *Computers and Operations Research*, 35:1065 – 1083, 2008.

[2] S. Arora and C. Lund. *Hardness of approximations*. PWS Publishing, 1997.

[3] Y. Azar and L. Epstein. On two dimensional packing. *Journal of Algorithms*, 25(2):290 – 310, 1997.

[4] Brenda S. Baker, Jr. E. G. Coffman, and Ronald L. Rivest. Orthogonal packings in two dimensions. *SIAM Journal on Computing*, 9(4):846–855, 1980.

[5] Brenda S. Baker and Jerald S. Schwarz. Shelf algorithms for two-dimensional packing problems. *SIAM Journal on Computing*, 12(3):508–525, 1983.

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

[7] J. E. Beasley. An exact two-dimensional non-guillotine cutting tree search procedure. *Operations Research*, 33:49–64, 1985.

[8] Jesús David Beltrán, Jose Eduardo Calderón, Rayco Jorge Cabrera, José A. Moreno Pérez, and J. Marcos Moreno-vega. Grasp/vns hybrid for the strip packing problem. In *In 1st International Workshop on Hybrid Metaheuristics*, pages 79–90, 2004.

[9] Bengt-Erik Bengtsson. Packing Rectangular Pieces - A Heuristic Approach. *The Computer Journal*, 25(3):353–357, 1982.

[10] A. Borodin and R. El-Yaniv. *Online Computation and Competitive Analysis*. Cambridge University Press, 1998.

[11] E. K. Burke, G. Kendall, and G. Whitwell. A new placement heuristic for the orthogonal stock-cutting problem. *Oper. Res.*, 52:655–671, August 2004.

[12] Edmund K. Burke, Graham Kendall, and Glenn Whitwell. A simulated annealing enhancement of the best-fit heuristic for the orthogonal stock-cutting problem. *INFORMS J. on Computing*, 21:505–516, July 2009.

[13] B. Chazelle. The bottomn-left bin-packing heuristic: An efficient implementation. *IEEE Trans. Comput.*, 32:697–707, August 1983.

[14] Nicos Christofides and Charles Whitlock. An Algorithm for Two-Dimensional Cutting Problems. *OPERATIONS RESEARCH*, 25(1):30–44, 1977.

[15] János Csirik and Gerhard J. Woeginger. Shelf algorithms for on-line strip packing. *Information Processing Letters*, 63(4):171 – 175, 1997.

[16] H. Dyckhoff. A typology of cutting and packing problems. *European J. Operational Research*, 44:145–159, 1990.

[17] S. P. Fekete, T. Kamphans, and N. Schweer. Online square packing. In *WADS '09: Proceedings of the 11th International Symposium on Algorithms and Data Structures*, pages 302–314, Berlin, Heidelberg, 2009. Springer-Verlag.

[18] Thomas A. Feo and Mauricio G. C. Resende. A probabilistic heuristic for a computationally difficult set covering problem. *Operations Research Letters*, 8(2):67 – 71, 1989.

[19] Thomas A. Feo and Mauricio G.C. Resende. Greedy randomized adaptive search procedures. *Journal of Global Optimization*, 6:109–133, 1995.

[20] 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.

[21] Guenther Fuellerer, Karl F. Doerner, Richard F. Hartl, and Manuel Iori. Ant colony optimization for the two-dimensional loading vehicle routing problem. *Comput. Oper. Res.*, 36:655–673, March 2009.

[22] Michel Gendreau, Manuel Iori, Gilbert Laporte, and Silvaro Martello. A tabu search heuristic for the vehicle routing problem with two-dimensional loading constraints. *Networks*, 51:1097–0037, 2008.

[23] Fred Glover. Tabu search and adaptive memory programing – advances, applications and challenges. In *Interfaces in Computer Science and Operations Research*, pages 1–75. Kluwer, 1996.

[24] Xin Han, Kazuo Iwama, Deshi Ye, and Guochuan Zhang. Strip packing vs. bin packing. *CoRR*, abs/cs/0607046, 2006.

[25] D. S. Hochbaum. *Approximation Algorithms for NP-Hard Problems*. PWS Publishing Company, 1997.

[26] E. Hopper and B. C. H. Turton. An empirical investigation of meta-heuristic and heuristic algorithms for a 2d packing problem. *European Journal of Operational Research*, 128:34–57, 2000.

[27] Eva Hopper and B. C. H. Turton. Problem generators for rectangular packing problems. *Studia Informatica Universalis*, 2:123–136, 2002.

[28] M. Iori, S. Martello, and M. Monaci. *Metaheuristic Algorithms for the Strip Packing Problem*, volume 78 of *Applied Optimization*, chapter 7. Springer, 2003.

[29] Manuel Iori, Juan-Jose Salazar-Gonzalez, and Daniele Vigo. An Exact Approach for the Vehicle Routing Problem with Two-Dimensional Loading Constraints. *TRANSPORTATION SCIENCE*, 41(2):253–264, 2007.

[30] Klaus Jansen and Roberto Solis-Oba. New approximability results for 2-dimensional packing problems. In Ludek Kucera and Antonín Kucera, editors, *Mathematical Foundations of Computer Science 2007*, volume 4708 of *Lecture Notes in Computer Science*, pages 103–114. Springer Berlin / Heidelberg, 2007.

[31] Klaus Jansen and Rob van Stee. On strip packing with rotations. In *Proceedings of the thirty-seventh annual ACM symposium on Theory of computing*, STOC '05, pages 755–761, New York, NY, USA, 2005. ACM.

[32] D. S. Johnson. *Near-optimal bin packing algorithms*. PhD thesis, MIT, Cambridge, MA, 1973.

[33] 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.

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

[35] Claire Kenyon and Eric Rémila. A near-optimal solution to a two-dimensional cutting stock problem. *Math. Oper. Res.*, 25:645–656, November 2000.

[36] Andrea Lodi, Silvano Martello, and Daniele Vigo. Heuristic and metaheuristic approaches for a class of two-dimensional bin packing problems. *INFORMS J. on Computing*, 11:345–357, April 1999.

[37] E. Pinto and J. F. Oliveira. Algorithm based on graphs for the non-guillotinable two-dimensional packing problem. In *Second ESICUP Meeting*, Southampton, May 2005. ESICUP, ESICUP.

[38] Marcelo Prais and Celso C. Ribeiro. Reactive GRASP: An Application to a Matrix Decomposition Problem in TDMA Traffic Assignment. *INFORMS J. on Computing*, 12(3):164–176, 2000.

[39] Ingo Schiermeyer. Reverse-fit: A 2-optimal algorithm for packing rectangles. In Jan van Leeuwen, editor, *Algorithms - ESA '94*, volume 855 of *Lecture Notes in Computer Science*, pages 290–299. Springer Berlin / Heidelberg, 1994.

[40] A. Steinberg. A strip-packing algorithm with absolute performance bound 2. *SIAM J. Comput.*, 26:401–409, April 1997.

[41] V. Vazirani. *Approximation Algorithms*. Springer-Verlag, 2001.

[42] Pearl Y. Wang and Christine L. Valenzela. Data set generation for rectangular placement problems. *European Journal of Operational Research*, 134(2):378 – 391, 2001.

[43] G. Wäscher, H. Haussner, and H. Schumann. An improved typology of cutting and packing problems. *European Journal of Operational Research*, 183(3):1109–1130, December 2007.

[44] D. Williamson. Lecture notes on appoximation algorithms. Technical Report 21409, T. J. Watson Research Center (IBM Research Division), Michigan State University, Yorktown Heights, New York, 1998.

[45] Emmanouil E. Zachariadis, Christos D. Tarantilis, and Christos 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.

Apêndice A

A.1 Minimizando a Função (3.1) do Lema 3.1.2

A fim de facilitar a notação renomearemos as variáveis ($h(B) \leftrightarrow x$, $w(B) \leftrightarrow y$ e $h \leftrightarrow z$) e, além disto, consideraremos $\alpha = \frac{2}{3}$. Logo, devemos minimizar a seguinte função

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

onde $\frac{3}{4} \leq y \leq x \leq 1$ e $0 \leq z \leq 3$ (os limites da região analisada).

Os pontos críticos de f podem ser encontrados com $\frac{\partial f}{\partial x} = 0$, $\frac{\partial f}{\partial y} = 0$ e $\frac{\partial f}{\partial z} = 0$, onde encontramos $p = (x, y, z) = (x, \frac{2}{5}, \frac{3}{2}x)$. Porém, calculando a *hessiana* para f em p , vemos que seu valor é menor que 0 e, portanto, p é um *ponto de sela*.

Logo, devemos analisar os limites da região imposta pelas restrições:

1. $y = \frac{3}{4} \Rightarrow f_1(x, z) = \frac{9x+2z}{12(x+z)}$, que não possui pontos críticos e, portanto, tem seu mínimo em alguma das bordas da sua região ($x = \frac{3}{4}$, $x = 1$, $z = 0$, $z = 3$). Com mínimo em $x = \frac{3}{4}$ e $z = 3$, com valor $\frac{17}{60}$;

2. $y = x \Rightarrow f_2(x, z) = \frac{x^2 - \frac{2}{3}z(x-1)}{x+z}$, que tem um único ponto crítico em $z = 3x$, e, portanto, equivale a $\frac{x^2 - \frac{2}{3}3x(x-1)}{x+3x} = \frac{2-x}{4}$, que é decrescente em x, portanto, o mínimo ocorre em $x = 1$ e vale $\frac{1}{4}$;

3. $x = 1 \Rightarrow f_3(y, z) = \frac{y(-2z+3)+2z}{3(z-1)}$, que tem um ponto crítico em $p = (y, z) = (-2, \frac{3}{2})$, que não pertence ao domínio da função f, tem seu mínimo em alguma das bordas da sua região ($y = \frac{3}{4}$, $y = 1$, $z = 0$, $z = 3$). Com mínimo em $y = 1$ e $z = 3$, com valor $\frac{1}{4}$;

4. $z = 0 \Rightarrow f_4(y) = y$, cujo mínimo vale $\frac{3}{4}$;

5. $z = 3 \Rightarrow f_5(x, y) = \frac{y(x-2)+2}{x+3}$, que tem um ponto crítico em $p = (x, y) = (2, 2)$, com as mesmas consequências de f_3. Logo, tem seu mínimo em alguma das bordas da

sua região ($x = \frac{3}{4}$, $x = 1$, $y = \frac{3}{4}$, $y = 1$). Com mínimo em $x = 1$ e $y = 1$, com valor $\frac{1}{4}$;

Portanto o mínimo da função ocorre quando $x = y = 1$ e $z = 3$, com valor $\frac{1}{4}$.

A.2 Minimizando a Função (4.3) do Lema 4.1.3

Novamente renomeando as variáveis para facilitar a notação devemos minimizar a seguinte função

$$g(x, y, z) = \frac{x^2 + y^2 + \frac{z}{3} + 1/27}{1 + x}$$

na região $1/6 < y \leq 1/3$, $y + z = 2/3 - x$ e $2/3 \geq x > 0$.

As derivadas parciais de g ($\frac{\partial g}{\partial x} = 0$, $\frac{\partial f}{\partial y} = 0$ e $\frac{\partial f}{\partial z}$) nos levam sempre a absurdos ou pontos fora da região analisada, logo, devemos analisar as bordas desta região impostas pelas restrições:

1. $y = \frac{1}{6} \Rightarrow g_1(x, z) = \frac{x^2 + \frac{z}{3} + 7/108}{1+x}$ que também não possui pontos críticos. Analisando a borda desta função veremos que ela alcança seu mínimo quando $(x, z) = \left(\frac{13}{6\sqrt{3}} - 1, \frac{3}{2} - \frac{13}{6\sqrt{3}}\right)$, levando a um mínimo de $\frac{1}{9}(13\sqrt{3} - 21) \approx 0.168518$;

2. $y = \frac{1}{3} \Rightarrow g_2(x, z) = \frac{x^2 + \frac{z}{3} + 4/27}{1+x}$ que não possui pontos críticos. Com as mesmas ideias empregadas na função g_1 chegamos ao ponto de mínimo $(x, z) = \left(\frac{\sqrt{\frac{43}{3}}}{3} - 1, \frac{4}{3} - \frac{\sqrt{\frac{43}{3}}}{3}\right)$, levando a um mínimo de $\frac{1}{9}(2\sqrt{129} - 21) \approx 0.190626$;

3. $x = \frac{2}{3} \Rightarrow g_3(y, z) = \frac{y^2 + \frac{z}{3} + \frac{13}{27}}{\frac{5}{3}}$ que não possui pontos críticos e nos leva a restrição $z = -y$ que inviabiliza o problema;

4. $x = 0 \Rightarrow g_4(y, z) = y^2 + \frac{z}{3} + \frac{1}{27}$ que não possui pontos críticos. Com ideias semelhantes as utilizadas nas funções g_1 e g_2 temos que o mínimo ocorre quando $y = 1/6$ e tem valor $\frac{25}{108} \approx 0.23148$;

5. $z = \frac{2}{3} - (x + y) \Rightarrow g_5(x, y) = \frac{x^2 + y^2 - \frac{x}{3} - \frac{y}{3} + \frac{7}{27}}{1+x}$ que tem um ponto crítico válido em $(x, y) = \left(\frac{13}{6\sqrt{3}} - 1, \frac{1}{6}\right)$ e tem valor igual ao da função g_1.

Portanto o mínimo da função ocorre quando $x = \frac{13}{6\sqrt{3}} - 1$, $y = \frac{1}{6}$ e $z = \frac{3}{2} - \frac{13}{6\sqrt{3}}$, com valor $\frac{1}{9}(13\sqrt{3} - 21) \approx 0.168518$.

A.3 Resultados dos experimentos

Os resultados completos dos experimentos realizados estão publicados no seguinte endereço `http://www.students.ic.unicamp.br/~ra089044/dissert/results.html`. Na Tabela 1 apresentamos os resultados para o conjunto S_1. Na tabela 2 apresentamos a média dos valores (aproximação e preenchimento) por instância original (com 10 cópias) dos conjuntos S_2 a S_8. Nas tabelas apresentamos os resultados dos algoritmos LBP, SO^{ur}, G_R, SO^{u} e G. As colunas G (5) e G_r (5) foram realizados com 5 iterações apenas e G (1000) e G_R (1000) com 1000 iterações.