Problema da Mochila com Itens Irregulares

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por Aline Marques Del Valle e aprovada pela Banca Examinadora.

Campinas, 16 de dezembro de 2010.

Eduardo Candido Xavier (Orientador)

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

Del Valle, Aline Marques
D389p Problema da mochila com itens irregulares/Aline Marques Del Valle-- Campinas, [S.P. : s.n.], 2010.

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

1.Problema da mochila. 2.Problema de empacotamento. 3.Algoritmos. 4.Heurística. I. Xavier, Eduardo Candido. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Título em inglês: Irregular knapsack problems

Palavras-chave em inglês (Keywords): 1. Knapsack problems. 2. Packing problems. 3. Algorithms. 4. Heuristic.

Área de concentração: Teoria da Computação

Titulação: Mestre em Ciência da Computação

Banca examinadora: Prof. Dr. Eduardo Candido Xavier (IC - UNICAMP)
Prof. Dr. Yuri Abitbol de Menezes Frota (IC – UFF)
Prof. Dr. Flávio Keidi Miyazawa (IC - UNICAMP)

Data da defesa: 16/12/2010

Programa de Pós-Graduação: Mestrado em Ciência da Computação

TERMO DE APROVAÇÃO

Dissertação Defendida e Aprovada em 16 de dezembro de 2010, pela Banca examinadora composta pelos Professores Doutores:

Prof. Dr. Yuri Abitbol de Menezes Frota
Centro Tecnológico / UFF

Prof. Dr. Flávio Keidi Miyazawa
IC / UNICAMP

Prof. Dr. Eduardo Cândido Xavier
IC / UNICAMP

Instituto de Computação
Universidade Estadual de Campinas

Problema da Mochila com Itens Irregulares

Aline Marques Del Valle[1]

Dezembro de 2010

Banca Examinadora:

- Eduardo Candido Xavier (Orientador)
- Yuri Abitbol de Menezes Frota
  Instituto de Computação - Universidade Federal Fluminense
- Flávio Keidi Miyazawa
  Instituto de Computação - Universidade Estadual de Campinas
- Luis Augusto Angelotti Meira
  Departamento de Ciência e Tecnologia - Universidade Federal de São Paulo
- Orlando Lee
  Instituto de Computação - Universidade Estadual de Campinas

[1]Suporte financeiro de: Bolsa Capes (processo 01-P.04388-10) 2008–2009, Bolsa do CNPq (processo 133457/2009-5) 2009–2010.

Resumo

Nesta dissertação, estudamos problemas de empacotamento com itens irregulares. Estamos particularmente interessados no Problema da Mochila Bidimensional: dados um recipiente de tamanho W x H e uma lista de itens bidimensionais, o objetivo é empacotar um subconjunto dos itens de forma a maximizar a área dos itens empacotados. Existem diversos trabalhos que lidam com problemas para itens e recipientes bidimensionais com forma regular (retangular). No entanto, são poucos os estudos que tratam de itens com formas irregulares. Nós propomos algoritmos de empacotamento para itens irregulares em recipientes limitados baseados no uso de No-Fit-Polygon (NFP). Este trabalho apresenta uma heurística GRASP para a versão restrita do Problema da Mochila: uma solução inicial gulosa é gerada e, em seguida, utiliza-se um algoritmo de busca local para melhorar solução atual. Uma estratégia híbrida também foi proposta para versão irrestrita do Problema da Mochila. Ela divide-se em passos de empacotamento de itens irregulares e empacotamento de itens regulares.

Testamos os algoritmos com instâncias adaptadas do problema de *Strip Packing*. O GRASP obteve empacotamentos ótimos para várias instâncias testadas e, mesmo para as instâncias em que o algoritmo não obteve resultados ótimos, os empacotamentos obtidos tiveram boa taxa de ocupação, com valores relativamente próximos do ótimo. O tempo de execução do algoritmo foi razoável. Na estratégia híbrida, obtiveram-se empacotamentos bons para a maioria das instâncias, com taxa de ocupação acima de 90% e tempos de execução relativamente baixos.

Abstract

In this work, we study packing problems dealing with two dimensional irregular items. We are particularly interested in the knapsack version of the problem: given a container with size W x H and a list of two dimensional items, the goal is to pack a subset of items such that the total area of packed items is maximized. There are several works that deal with problems for the case where items and containers have regular shapes (rectangular). However, only a few studies deal with items with irregular shapes. We propose algorithms for packing irregular items in limited containers based on the use of No-Fit-Polygon (NFP). This work presents a GRASP algorithm for the restricted version of the Knapsack Problem: first, a greedy initial solution is generated, then, the local search algorithm is used to improve the current solution. A hybrid strategy has also been proposed for the unrestricted version of the Knapsack Problem. It is divided into steps of packing irregular items and packing regular items.

We tested the algorithms using adapted instances for the Strip Packing problem. The GRASP algorithm achieved optimal packings for several of the tested instances, and, even for those that the algorithm did not, the obtained packings had a good occupancy rate, with values relatively close to the optimum. The runtime of the algorithm was reasonable. In the hybrid strategy, we obtained good packings for most of the instances, with occupancy rates above 90% and relatively low execution times.

Agradecimentos

Agradeço primeiramente a Deus por me dar a oportunidade de estar aqui hoje. Aos meus pais, João Carlos e Celina, meus irmãos, Carla e Tiago, e meu cunhado, Éder, agradeço todo respeito, compreensão, companheirismo, carinho e amor. Agradeço à caçulinha da família, Isabela, que veio renovar as esperanças e trazer muita alegria. Agradeço também a todos colegas e amigos do IC com quem convivi, que tornaram minha caminhada mais leve. Tenho muito a agradecer e a muitas pessoas, dos quais não vou citar nomes para não ser injusta.

Meus agradecimentos especiais a:

Prof. Ricardo Martins de Abreu Silva, que me abriu caminhos nunca antes pensados, me incentivando à pesquisa e ao mestrado.

Prof. Eduardo Candido Xavier, meu orientador, por sua paciência, incentivo, amizade e orientação ao longo desses dois anos.

Meu amigo Lehilton, por sua preciosa ajuda em todas as etapas do trabalho.

D. Mara, por ter sido uma mãe ao me acolher em sua casa.

À CAPES e ao CNPQ que contribuíram para que esta pesquisa acontecesse.

A todos, meu muito obrigada!

Sumário

Lista de Tabelas

Lista de Figuras

Capítulo 1

Introdução

Um dos principais objetivos da indústria moderna é a diminuição de custos na busca de lucro e competitividade. Esse objetivo está diretamente relacionado à diminuição dos custos variáveis. Por exemplo, a matéria-prima representa uma parcela significativa dos custos, com impacto tanto econômico, quanto ecológico no processo de produção [16]. Portanto, uma maneira de reduzir custos é a otimização do uso da matéria prima. As indústrias de tecido, couro, papel, móveis, vidros e folhas de metal são exemplos de ramos industriais em que é fundamental o uso eficiente de matéria-prima.

Outro exemplo de custo variável é o transporte de produtos e materiais. Normalmente, vários produtos são encaixotados e devem ser transportados em contêineres de dimensões fixas. Como cada viagem necessária para transportar um contêiner representa um custo adicional, é importante diminuir o número de viagens necessárias e, portanto, o número de contêineres necessários para transportar todos produtos.

Os exemplos acima correspondem a problemas muito comuns na indústria: o *problema do corte* – dada a matéria-prima e um conjunto de formas para o corte do material, as formas devem ser posicionadas maximizando a área ocupada e minimizando o desperdício da matéria-prima – e o *problema do empacotamento em contêineres* – dado um conjunto de itens e respectivos contêineres, os itens devem ser organizados maximizando o número de itens empacotados e minimizando o número de contêineres necessários. Os problemas acima podem ser tratados de maneira bastante similar e, doravante, serão descritos simplesmente como *problemas de empacotamento*.

Os problemas que têm como objetivo maximizar ou minimizar uma função objetivo definida em algum domínio são chamados problemas de otimização. Se o domínio de um problema for finito, então ele é chamado de problema de otimização combinatória. Problemas de empacotamento podem ser formulados como problemas de otimização combinatória. Isso significa que podemos enumerar e testar todas as combinações de itens possíveis, na busca do melhor empacotamento. Entretanto, a ideia ingênua de testar todas

as combinações para encontrar o empacotamento ótimo pode ser inviável na prática, pois o domínio pode ser muito grande.

Vários problemas de empacotamento pertencem à classe de problemas NP-difíceis. Isso é, sob a hipótese de que P $\neq$ NP, não existem algoritmos de tempo polinomial que resolvam de maneira exata esse tipo de problema. Embora não haja provas de que tais algoritmos não existam, não se espera que eles sejam encontrados [27]. Na prática, no entanto, mesmo na falta de uma solução ótima do problema, muitas vezes podemos utilizar a melhor solução encontrada. Em resposta a essas dificuldades, surgiram vários métodos, que são utilizados para tentar resolver, ou encontrar soluções boas (não necessariamente ótimas) para problemas de otimização combinatória. Dentre esses métodos, destacamos programação inteira, heurísticas e algoritmos aproximados.

No método de programação linear inteira (PLI), formulamos um programa linear, composto por uma função objetivo e por um conjunto de restrições, e assumimos que todas as variáveis são inteiras. Há várias técnicas de tratamento de PLI, como arredondamento, relaxação, *Branch & Bound* e *Branch & Cut*. Mais detalhes sobre programação inteira podem ser encontrados em [31, 32].

Um algoritmo de aproximação é um algoritmo que encontra uma solução α-aproximada para o problema em tempo polinomial. Isso quer dizer que, embora a solução não seja necessariamente ótima, ela dista no máximo em um fator α do valor ótimo. Mais detalhes sobre algoritmos de aproximação podem ser encontrados em [29].

Heurísticas são técnicas de resolução de problemas que não garantem que a solução resultante seja ótima, mas produzem soluções de boa qualidade. Entre essas técnicas, há a Busca Tabu, algoritmos genéticos, GRASP, etc. Em geral, não podemos afirmar quão boa será uma solução gerada por uma heurística, ao contrário de soluções geradas por algoritmos aproximados.

Em nosso trabalho, propomos uma heurística GRASP para o Problema da Mochila com itens irregulares. Utilizamos heurísticas porque elas têm sido utilizadas com sucesso para obtenção de soluções de boa qualidade para problemas de empacotamentos [2, 7, 15, 16, 21, 22, 23, 24, 25, 28]. Já algoritmos de aproximação e PLI são difíceis de serem projetados para problemas de empacotamento, especialmente quando os itens são irregulares.

Propomos também uma estratégia híbrida para resolver a versão irrestrita do Problema da Mochila com itens irregulares. A estratégia divide-se em passos de empacotamento de itens irregulares e empacotamento de itens regulares. Propomos um algoritmo guloso para empacotar itens irregulares em retângulos e utilizamos os algoritmos de Cintra *et al.* [10] para o problema *Rectangular Knapsack* para empacotar os retângulos. Utilizamos os algoritmos de Cintra *et al.* [10], pois são algoritmos exatos e estamos interessados em produzir os melhores resultados possíveis.

1.1 Organização da Dissertação

Nas próximas seções, descrevemos os problemas de empacotamentos, inclusive o Problema da Mochila, e apresentamos alguns trabalhos relacionados. No Capítulo 2, descrevemos o conceito de No-Fit-Polygons, utilizado para encontrar posicionamentos viáveis para itens, e apresentamos os algoritmos de Cuninghame-Green [11] e de Burke *et al.* [8] para o cálculo de No-Fit-Polygons de itens convexos e de itens convexos e não-convexos, respectivamente. No Capítulo 3, propomos uma rotina básica para empacotamento de itens irregulares. Utilizamos a heurística de escolher o posicionamento viável de um item que minimize a área retangular do empacotamento e, para isso, apresentamos o algoritmo de busca estendida de Adamowicz e Albano [1]. Propomos também duas abordagens para a geração dos No-Fit-Polygons utilizados na rotina de empacotamento. No Capítulo 4, propomos um algoritmo para o Problema da Mochila com itens irregulares baseado na meta-heurística GRASP e no empacotamento de sequências de itens pela rotina proposta no Capítulo 3. No Capítulo 5, propomos uma estratégia híbrida para o caso irrestrito do Problema da Mochila com itens irregulares. Essa estratégia é baseada no empacotamento de itens irregulares em retângulos, feito pela rotina de empacotamento proposta, e no empacotamento desses retângulos, utilizando algoritmos para itens regulares. No Capítulo 6, testamos nossos algoritmos e apresentamos os resultados obtidos. No Capítulo 7, concluímos nosso trabalho e ressaltamos alguns possíveis trabalhos futuros.

1.2 Problemas de Empacotamentos

Um problema de empacotamento é composto por 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 objetivo é empacotar itens dentro de recipientes, de forma a maximizar ou minimizar uma função objetivo. Tanto os itens quanto os recipientes podem assumir formas regulares ou irregulares. Por exemplo, podemos representar formas regulares por retângulos e formas irregulares por polígonos em geral. O empacotamento deve ser feito de tal maneira que os itens não se sobreponham e que as capacidades do recipiente sejam respeitadas.

Problemas de empacotamentos podem se apresentar tendo uma quantidade variável de itens e um recipiente de dimensões fixas, ou em uma quantidade fixa de itens e um recipiente de dimensões variáveis [30]. Nesse contexto, descrevemos os problemas *Bin Packing*, *Knapsack Problem* e *Strip Packing*.

O problema *Bin Packing* unidimensional tem como entrada uma lista de itens $L = (a_1, \ldots, a_m)$, tal que cada item a_i tem tamanho $s(a_i)$, e um número B, que indica o tamanho do recipiente. Assumimos que, para todo item $a_i \in L$, vale que $s(a_i) \leq B$.

Este problema consiste em empacotar todos os itens de L no menor número possível de recipientes, ou seja, devemos achar uma partição $P_1, \ldots, P_q$ de L tal que q seja mínimo e $\sum_{a_i \in P_j} s(a_i) \leq B$ para cada parte P_j. Podemos considerar versões multidimensionais desse problema.

No *Knapsack Problem*, ou Problema da Mochila, consideramos um recipiente de tamanho B e uma lista de itens $L = (a_1, \ldots, a_m)$, tal que cada item tem tamanho $s(a_i)$ e valor $p(a_i)$. O objetivo do problema é empacotar um subconjunto dos itens de forma a maximizar o valor dos itens empacotados. Formalmente, queremos encontrar $C \subseteq L$ tal que $\sum_{a_i \in C} s(a_i) \leq B$ e $\sum_{a_i \in C} p(a_i)$ seja máximo. O problema, como foi definido, é conhecido como Problema da Mochila Restrito, ou Mochila 0/1. Neste caso, cada item pode ser empacotado apenas uma vez. Na versão do Problema da Mochila Irrestrita, um item $a_i \in L$ pode ser empacotado várias vezes. Como no caso do *Bin Packing*, também podemos considerar versões multidimensionais desse problema.

No problema *Strip Packing*, consideramos uma lista de itens retangulares $L = (a_1, \ldots, a_m)$, tal que cada item a_i tem dimensões $x(a_i)$ e $y(a_i)$, e uma faixa 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 utilizada para empacotar todos os itens. Também podem ser consideradas versões em que os itens possuem formas irregulares, ou versões com objetos de três ou mais dimensões.

Segundo Wäscher *et al.* [30], são escassas referências para problemas de empacotamentos com itens irregulares, principalmente para os que envolvem maximização da função objetivo. O Problema da Mochila é um exemplo desses problemas, que busca maximizar o valor dos itens empacotados. Neste trabalho, propomos uma heurística GRASP e uma Estratégia Híbrida para o Problema da Mochila bidimensional para itens irregulares.

1.3 Trabalhos Relacionados

Problemas de empacotamentos podem ser classificados de acordo com seus objetivos e dependendo das características dos recipientes e dos itens considerados. Os problemas podem lidar com recipientes de dimensões fixas ou variáveis, assumir a existência de um único recipiente ou vários deles e assim por diante. Podemos considerar itens cujas geometrias das formas são iguais ou diferentes, regulares ou irregulares. Dyckhoff [12] apresentou uma classificação abrangente para problemas de empacotamentos que foi revisada por Waescher *et al.* [30]. Dessa maneira, os problemas de empacotamento são classificados de acordo com suas características:

- *Quanto à dimensionalidade*: uma, duas, três ou mais dimensões.

- *Quanto aos objetivos do problema*: maximizar valores como o custo ou minimizar valores como o desperdício, por exemplo.
- *Quanto à variedade dos itens*: os itens são iguais, fracamente heterogêneos, ou fortemente heterogêneos.
- *Quanto à quantidade de recipiente*: há apenas um recipiente ou vários recipientes.
- *Quanto à geometria dos itens*: os itens são regulares (retângulos, círculos, caixas, cilindros, bolas, etc) ou irregulares (polígonos em geral).

Em particular, os problemas de empacotamento bidimensional *Strip Packing*, *Knapsack Problem* (Problema da Mochila) e *Bin Packing* são amplamente estudados. Nas subseções seguintes, apresentamos sucintamente algumas abordagens utilizadas na literatura para lidar com esses problemas para o caso com itens irregulares.

1.3.1 Strip Packing Irregular

O problema *Strip Packing* é tratado em diversos trabalhos tanto no caso em que os itens são todos retangulares, quando o problema é chamado de *Retangular Strip Packing*, quanto no caso de itens irregulares, quando o problema é chamado de *Irregular Strip Packing*.

Oliveira *et al.* [25] apresentaram um algoritmo guloso para o *Irregular Strip Packing*. A cada iteração, um item candidato é escolhido para fazer parte da solução parcial e é posicionado da melhor maneira possível. São consideradas todas as posições viáveis para um item, que podem ser encontradas por meio de No-Fit-Polygons. A posição viável escolhida é a que minimiza a área da envoltória retangular de dois itens; ou que minimiza o comprimento da envoltória retangular de dois itens; ou, ainda, que maximiza a sobreposição entre a envoltória retangular de dois itens. Adamowicz e Albano [1] propuseram um algoritmo para encontrar essa posição.

Gomes e Oliveira [15] propuseram uma estratégia para resolver o *Irregular Strip Packing* empacotando itens em uma sequência definida de acordo com uma heurística de baixo nível e, em seguida, testando trocas de dois itens nessa sequência. Inicialmente, os itens são colocados em uma sequência que obedece a um determinado critério, como ordem decrescente de área, de altura, de largura, ou ordenados aleatoriamente. Utilizando a heurística de posicionamento Bottom-Left combinada com o uso de No-Fit-Polygons e Inner-Fit-Rectangle, os itens são posicionados no recipiente em ordem. A partir de uma solução inicial, um procedimento de busca local é realizado. Esse procedimento consiste em obter novas sequências, trocando-se dois itens da sequência corrente, e gerar os empacotamentos correspondentes. Cada solução vizinha é testada, à procura de soluções melhores.

Gomes e Oliveira [16] propuseram um algoritmo híbrido, composto por Simulated Annealing e por algoritmos de Separação e Compactação baseados em Programação Linear, para resolver o *Irregular Strip Packing*. Similarmente ao algoritmo anterior [15], os itens são posicionados por meio da heurística Bottom-Left combinada com o uso de No-Fit-Polygons e Inner-Fit-Rectangle. Nesse algoritmo, a troca de posição de itens no recipiente é feita por meio de Simulated Annealing. O algoritmo de Separação é empregado para remover possíveis sobreposições e, em seguida, o algoritmo de Compactação é utilizado para juntar itens eventualmente afastados. O objetivo é encontrar soluções melhores e mais compactas.

Burke *et. al* [7] resolveram o problema *Irregular Strip Packing* por meio da heurística Bottom-Left e dos mecanismos de busca local Hill Climbing e Busca Tabu. Assim como o algoritmo de Gomes e Oliveira [15], os itens são colocados em uma tupla, ordenados por área ou por tamanho, e posicionados seguindo a ordem da tupla. No entanto, é proposta uma nova heurística Bottom-Left que não faz o uso de No-Fit-Polygons no posicionamento dos itens. Nessa nova heurística, um item é posicionado no canto inferior esquerdo do recipiente e é movido vertical e horizontalmente, até que se encontre uma posição viável. O movimento vertical é contínuo, obtido por funções trigonométricas que verificam se a posição do item é viável e computam o descolamento vertical que leva o item a uma posição viável. Se na posição horizontal atual não for encontrada uma posição viável, o item é movido horizontalmente à direita em um valor fixo. Para os mecanismos de busca, foram testadas várias opções de trocas para gerar os vizinhos.

1.3.2 Knapsack Problem Irregular

Tay *et al.* [28] apresentaram um Algoritmo Genético (AG) para posicionar itens em um recipiente de forma irregular, de maneira a maximizar a área ocupada. Os itens são empacotados sequencialmente, até que não seja mais possível colocar nenhum item no recipiente. A cada item empacotado, um novo recipiente é calculado, extraindo o item da borda do recipiente atual. Para empacotar um item, o AG trabalha com as variáveis x, posição do item ao longo da borda do recipiente, e θ, rotação do item, que são consideradas em suas representações binárias. Inicialmente, é criada uma população de itens idênticos espalhados em diversas posições e rotações diferentes, de maneira que cada item tenha pelo menos um vértice tocando a borda. A cada iteração do AG, as variáveis x e θ sofrem *crossovers* e mutações, na busca do item com posição e rotação que minimizem a área ocupada. Anand *et al.* [2] também usaram um AG para posicionar itens em recipientes de dimensões fixas, maximizando o valor dos itens empacotados.

Martins e Tsuzuki [21, 22, 23, 24] propuseram um algoritmo para o problema de posicionamento rotacional de múltiplos itens em um recipiente de dimensões fechadas, de

forma a maximizar a área ocupada. Os itens são empacotados sequencialmente de acordo com parâmetros θ, f e r. A cada item empacotado, os seguinte passos são realizados: a rotação θ é aplicada; regiões livres de colisão são calculadas considerando os itens já empacotados; o parâmetro f é utilizado para escolher uma região livre de colisão; e o parâmetro r é utilizado para definir a posição, no perímetro da região escolhida, em que o item será empacotado. Para encontrar regiões livres de colisão, é feito o uso de No-Fit-Polygons e Inner-Fit-Rectangle. Inicialmente, os itens são ordenados de acordo com alguma heurística, como Random Permutation, Larger First, ou Weight Sort. Para obter soluções melhores, a técnica de Simulated Annealing é aplicada a cada iteração do algoritmo, alterando um parâmetro de algum item, ou mudando a ordem de dois itens.

1.3.3 Bin Packing Irregular

Na literatura, existem várias abordagens para o problema *Strip Packing* com itens irregulares. Para o problema *Bin Packing*, a maioria dos trabalhos envolvem itens retangulares. Ressaltamos, no entanto, que podemos utilizar uma abordagem de *Strip Packing* para tratar o problema de *Bin Packing*. Primeiramente, resolvemos o problema do *Irregular Strip Packing* para o mesmo conjunto de itens irregulares em uma faixa da mesma largura do recipiente. Em seguida, dividimos o empacotamento obtido na faixa em diversos recipientes, de acordo com a altura do recipiente.

Encontramos várias referências para o caso onde os itens são retângulos. Chung *et al.* [9] propuseram um algoritmo de duas fases para o *Bin Packing* retangular chamado Hybrid First-Fit. Na primeira fase, o algoritmo resolve um *Strip Packing* usando a estratégia baseada em níveis First-Fit Decreasing Height. O resultado obtido na primeira fase é o empacotamento dos itens em retângulos de diferentes alturas, correspondentes a cada um dos níveis. Em seguida, é utilizado o algoritmo de *Bin Packing* unidimensional First-Fit Decreasing para distribuir os retângulos em recipientes: cada retângulo é colocado no primeiro recipiente que o comporte, ou, se o retângulo não puder ser colocado em nenhum existente, um novo recipiente é utilizado. Outros algoritmos para o *Bin Packing* com itens retangulares podem ser encontrados nos trabalhos de Berkey e Wang [6], Frenk e Galambos [14] e Lodi *et al.* [19].

Capítulo 2

No-Fit-Polygon

Na construção de soluções viáveis para problemas de empacotamentos, temos que lidar com a geometria dos itens. A geometria está presente em todas as interações entre itens. Por exemplo, para detectar sobreposições e para calcular distâncias de translações para resolução das sobreposições.

Bennell e Oliveira [5] descrevem abordagens relacionadas à geometria, para verificar se os itens tocam-se, sobrepõem-se, ou estão separados. Entre essas abordagens, citam o método *rasther*, a trigonometria direta, o No-Fit-Polygon e a função *Phi*. O método *rasther* divide o recipiente em áreas discretas, reduzindo a informação geométrica e codificando os itens por meio de um *grid*, que é representado por uma matriz. Esse método elimina e identifica sobreposições entre itens por meio de contagem de células do *grid*. Quando os itens são representados por polígonos, a trigonometria direta também pode ser utilizada. A função D, em especial, é usada para verificar o relacionamento entre as arestas do polígono e para identificar interseções entre elas e, portanto, a sobreposição de itens. O No-Fit-Polygon tem se tornado a abordagem mais popular para lidar com a geometria. Verificando a posição do ponto de referência de um item orbital em relação ao No-Fit-Polygon correspondente, é possível identificar se os itens tocam-se, sobrepõem-se ou estão separados. A função *Phi* é composta de expressões matemáticas que representam a posição mútua de dois itens. Se o valor da função *Phi* for maior que zero, os itens estão separados ou, se o valor for igual a zero, os itens tocam-se e, se o valor for menor que zero, os itens sobrepõem-se.

O No-Fit-Polygon tem o mesmo efeito das operações trigonométricas para se determinar pontos viáveis de empacotamentos. O cálculo do No-Fit-Polygon pode ser mais rápido, pois ele pode ser pré-calculado para pares de itens, o que reduz significativamente o tempo de execução. Utilizamos o No-Fit-Polygon em nosso trabalho e descrevemo-lo na Seção 2.2.

Cuninghame-Green [11] propôs um algoritmo simples para o cálculo do No-Fit-Polygon

para polígonos convexos. Mahadevan [20] propôs uma abordagem orbital para o cálculo do No-Fit-Polygon de polígonos convexos e não-convexos. Mais tarde, Burke *et al.* [8] propuseram uma nova abordagem orbital baseando-se em [20]. Neste trabalho, usamos os algoritmos de [11] e [8], que serão discutidos nas Seções 2.3 e 2.4, respectivamente. Outras abordagens para o cálculo do No-Fit-Polygon, como a Soma de Minkowski, podem ser encontradas em [4]. Definições importantes para o entendimento deste capítulo serão dadas na Seção 2.1

2.1 Conceitos

Nesta seção, definimos os conceitos de *Item*,*Vetor de Translação*, *Toucher* e *Posição Relativa de Duas Arestas*, que serão utilizados no decorrer deste Capítulo.

Definição 2.1.1 *Itens são objetos, n-dimensionais, com formas regulares ou irregulares. No espaço bidimensional, podem ser representados por polígonos, com arestas* $((x_i, y_i), (x_{i+1}, y_{i+1}))$ *orientadas ligando pontos consecutivos* (x_i, y_i) *e* (x_{i+1}, y_{i+1}).

Definição 2.1.2 *Vetores de Translação são vetores que definem o tamanho, a direção e o sentido para o deslocamento de um objeto de um ponto a outro.*

Definição 2.1.3 *Toucher é um par de arestas de polígonos diferentes que se tocam em um ponto.*

Definição 2.1.4 *A posição relativa de duas arestas orientadas* UV *e* AB *define a posição de* UV *em relação a* AB*: se* UV *está à direita, à esquerda ou é paralela a* AB.

A função D pode ser utilizada para encontrar esta relação usando os pontos da aresta UV em relação à aresta AB. Dados uma aresta orientada AB e um ponto P, é possível verificar se P está à direita, à esquerda ou sobre a aresta AB. A função D pode ser definida pela seguinte fórmula [4]:

$$D_{ABP} = ((X_A - X_B)(Y_A - Y_P) - (Y_A - Y_B)(X_A - X_P))$$

A partir do valor encontrado para a função D é possível definir a posição de P em relação à aresta AB:

- Se $D_{ABP} > 0$ então o ponto P está à esquerda da aresta AB
- Se $D_{ABP} < 0$ então o ponto P está à direita da aresta AB
- Se $D_{ABP} = 0$ então o ponto P está sobre a aresta AB

A Figura 2.1(a), Figura 2.1(b) e 2.1(c) ilustram os casos em que P está à direita, à esquerda e sobre a aresta AB, respectivamente.

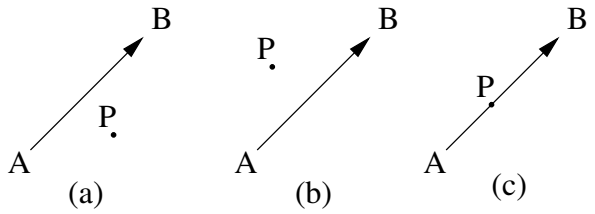


Figura 2.1: Posição do ponto P em relação à aresta AB: (a) Direita. (b) Esquerda. (c) Sobre

2.2 Descrição

O conceito de No-Fit-Polygon é usado para descrever regiões viáveis de posicionamento de duas formas. Em problemas de empacotamentos, é usado como estratégia de posicionamento para assegurar leiautes viáveis, ou seja, sem sobreposição de itens.

O No-Fit-Polygon pode ser obtido por meio de um conjunto de translações de uma forma representado matematicamente por vetores. É possível associar a cada forma um ponto de referência. O conjunto de translações da forma e, consequentemente, do ponto de referência, resultam em um conjunto de vetores que formam o No-Fit-Polygon.

Dados dois polígonos, A e B, o No-Fit-Polygon correspondente pode ser encontrado transladando um polígono ao redor das arestas do outro. Um polígono permanece fixo e o outro orbita ao redor das arestas do anterior, assegurando que os polígonos sempre se tocam, mas nunca se interceptam. O ponto de referência de um polígono é qualquer ponto interno ou externo ao polígono. Na Figura 2.2, A é o polígono fixo, B é o polígono orbital e o ponto de referência do polígono B é o vértice de B com maior ordenada. Quando o polígono B translada ao redor do polígono A, o ponto de referência de B forma um caminho denominado NFP_{AB}.

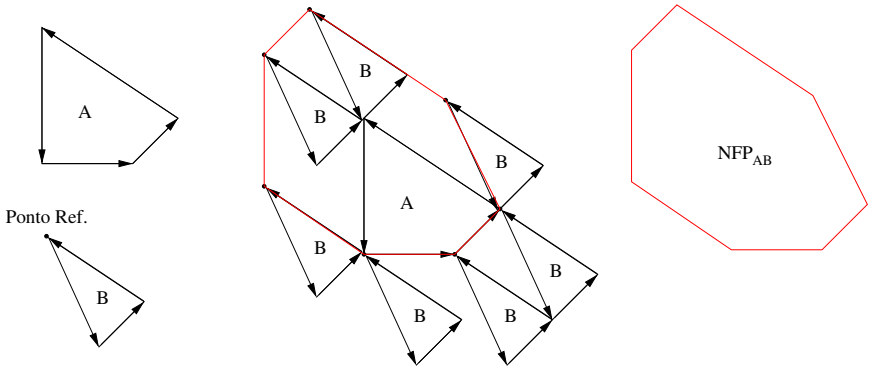


Figura 2.2: No-Fit-Polygon dos polígonos A e B.

Para testar se dois polígonos se sobrepõem, usamos o NFP_{AB} e o ponto de referência

do polígono B. Se o ponto de referência é posicionado dentro do NFP_{AB}, então o polígono B sobrepõe o polígono A. Se o ponto de referência é posicionado na borda do NFP_{AB}, então os polígonos A e B tocam-se. Finalmente, se o ponto de referência está fora do NFP_{AB}, então A e B não se tocam e não se sobrepõem, isto é, estão separados.

2.3 Algoritmo para o Cálculo de NFP para Polígonos Convexos

Cuninghame-Green [11] apresenta uma abordagem para produzir configurações de espaço-obstáculo para polígonos convexos. Quando os polígonos não são convexos, é calculada a envoltória convexa dos polígonos e, em seguida, são calculados os No-Fit-Polygons.

Para a construção do No-Fit-Polygon, Cuninghame-Green [11] adota um polígono fixo com orientação anti-horária, A, e um polígono orbital com orientação horária, B (Figura 2.3(a)). Transladam-se todas as arestas de A e B para um único ponto, o que é equivalente a ordenar as arestas de A e B por suas inclinações (Figura 2.3(b)). Concatenam-se as arestas no sentido anti-horário para produzir o No-Fit-Polygon, como mostra a Figura 2.3(c). Resumidamente, o algoritmo consiste basicamente em ordenação de arestas e reordenação por meio de translação.

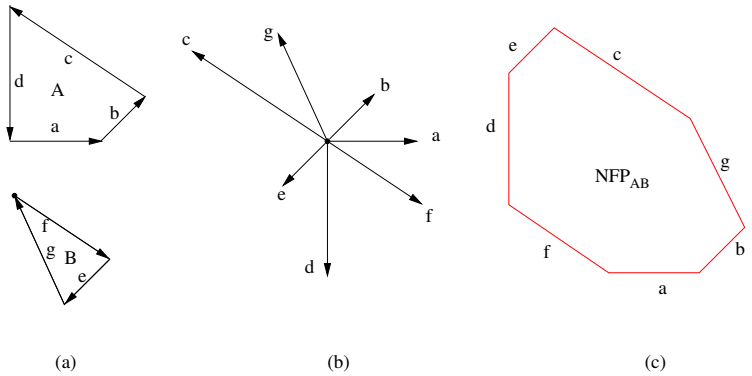


Figura 2.3: Algoritmo para o No-Fit-Polygon de polígonos convexos. Fonte: [11].

2.4 Algoritmo para o Cálculo de NFP para Polígonos Convexos e Não-Convexos

Nesta seção, apresentamos o algoritmo orbital proposto por Burke *et al.* [8] para o cálculo do No-Fit-Polygon. O Algoritmo pode ser dividido em duas fases. Na primeira, um

polígono orbital desliza ao redor do polígono fixo para gerar a borda externa do No-Fit-Polygon de dois polígonos. Como os polígonos podem não ser convexos, pode haver concavidades não percorridas pelo polígono orbital. Essas concavidades podem gerar buracos que não são obtidos na primeira fase. Na segunda fase, identificam-se os possíveis pontos de início para encontrar buracos do No-Fit-Polygon.

No movimento orbital, assumem-se dois polígonos no sentido anti-horário, A e B, para gerar o NFP_{AB}. O ponto de referência usado no movimento de translação de B é o vértice de B, (x, y), com maior ordenada (maior valor de y) e, caso haja vários vértices com ordenadas iguais, aquele com a maior abscissa (maior valor de x) é escolhido como ponto de referência. A posição inicial do ponto de referência de B coincide com o vértice de A com menor ordenada e, caso haja vários vértices com ordenadas iguais, coincide com aquele de menor abscissa (Figura 2.4). A escolha desses vértices garante que A e B sempre se tocam, mas nunca se interceptam. Nesta posição, pode-se começar a gerar o NFP_{AB}.

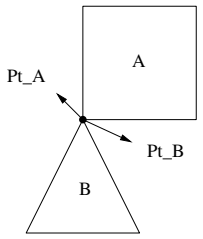


Figura 2.4: Posição inicial dos polígonos A e B para o cálculo do No-Fit-Polygon.

2.4.1 Primeira fase

A primeira fase do algoritmo é iterativa, sendo que, em cada iteração, uma aresta para gerar o No-Fit-Polygon é escolhida. Para construir o NFP_{AB}, o polígono B parte da posição inicial e vai transladando ao redor do polígono A, usando o vetor de translação escolhido em cada iteração, até retornar a origem. Os passos para a criação do NFP_{AB} podem ser divididos em *detecção de pares de arestas que se tocam*, *criação de possíveis vetores de translação*, *busca de uma translação viável*, *poda das translações viáveis* e *aplicação da translação viável*.

Chamamos pares de arestas que se tocam de *touchers* e representamos um *toucher* por uma aresta do polígono fixo, A, e uma aresta do polígono orbital, B. Todo *toucher* possui um ponto em comum entre suas arestas. Para encontrar os *touchers*, podemos verificar se existem pontos em comum entre todas as arestas do polígono A e todas as arestas do polígono B. A Figura 2.5 apresenta os *touchers* (a_1, b_2), (a_1, b_3), (a_4, b_2) e (a_4, b_3) dos polígonos A e B.

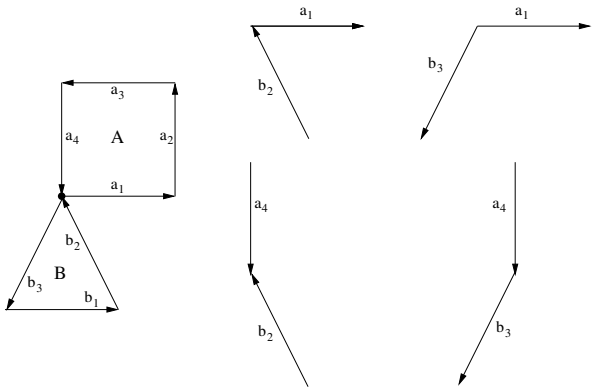


Figura 2.5: *Touchers* entre os polígonos A e B.

A criação dos possíveis vetores de translação leva em consideração os *touchers*, pois o vetor de translação é derivado de uma aresta de A ou de uma aresta de B. A Figura 2.6(a) ilustra a translação do polígono orbital através do vetor a_1, derivada do polígono A e a Figura 2.6(b) mostra a translação através do vetor $-b_3$, derivada do polígono B. Como a aresta de translação b_3 pertence ao polígono B, poderíamos transladar o polígono A através da aresta b_3. No entanto, para manter o polígono A fixo, transladamos o polígono B, invertendo a direção da aresta de translação, obtendo $-b_3$.

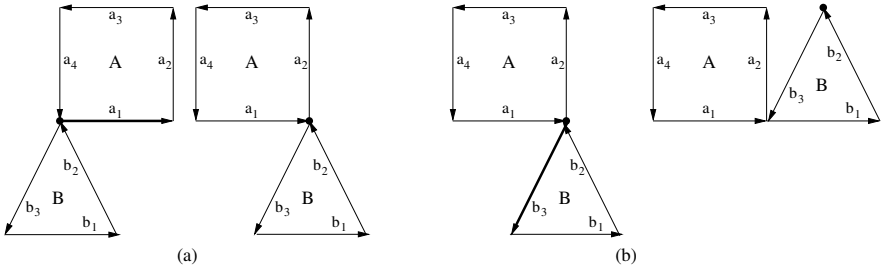


Figura 2.6: Vetor de translação: (a) Derivado de a_1. (b) Derivado de b_3.

Os vetores de translação são obtidos a partir dos *touchers*. Há três possibilidades de combinação das arestas (Figura 2.7):

I. Ambas as arestas tocam-se em um vértice.

II. Um vértice da aresta orbital toca no meio da aresta fixa.

III. Um vértice da aresta fixa toca no meio da aresta orbital.

O caso I requer a correta identificação da aresta da qual o vetor de translação é derivado: da aresta fixa ou da aresta orbital. Ela pode ser identificada baseando-se nos

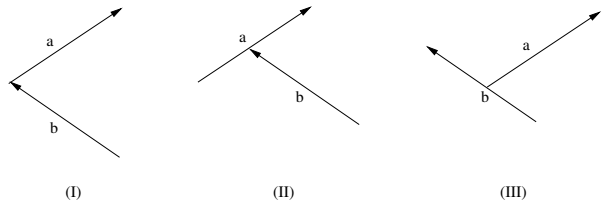


Figura 2.7: Tipos de pares de arestas que se tocam.

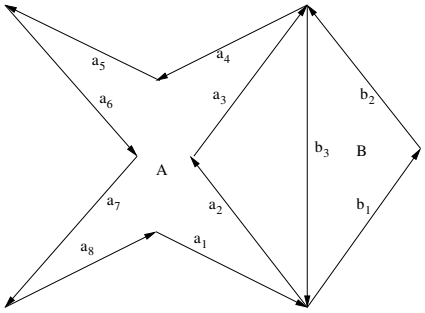
Caso	**Vértice das Arestas**		**Posição da aresta orbital em relação a aresta fixa**	**Vetor de Translação derivado de**
	Fixa	**Orbital**		
1	Início	Início	Esquerda	Aresta Orbital
2	Início	Início	Direita	Aresta Fixa
3	Início	Fim	Esquerda	-
4	Início	Fim	Direita	Aresta Fixa
5	Fim	Início	Esquerda	-
6	Fim	Início	Direita	Aresta Orbital
7	Fim	Fim	-	-
8	-	-	Paralela	Ambas as arestas

Tabela 2.1: Derivando possíveis translações quando as arestas se tocam em um vértice.

vértices das arestas do *toucher* e do teste da posição da aresta orbital em relação à aresta fixa, que verifica se a aresta orbital está à direita, à esquerda ou paralela à aresta fixa. A Tabela 2.1 apresenta os possíveis vetores de translação derivados nas várias circunstâncias possíveis. Um traço na última coluna da tabela significa que não é possível obter um vetor de translação do *toucher* correspondente, pois transladar o polígono orbital através de a ou $-b$ levaria à sobreposição entre o polígono fixo e o polígono orbital.

A Figura 2.8 mostra os *touchers* e os possíveis vetores de translação derivados dos polígonos A e B, quando ambas as arestas tocam-se em um vértice. Os vetores de translação foram obtidos a partir do ponto em comum das arestas do *toucher* e por meio do teste da posição da aresta orbital em relação à aresta fixa. Com base na Tabela 2.1, encontramos as possíveis translações. Para o *toucher* (a_2, b_1), por exemplo, o ponto em comum das arestas ocorre no início da aresta a_2 e no início da aresta b_1, sendo que b_1 esta à direita de a_2. De acordo com a tabela, identificamos o caso 2, em que o vetor de translação é derivado da aresta fixa. Nesse caso, a translação é derivada de a_2.

No caso II, quando um vértice da aresta orbital toca o meio da aresta fixa, o vetor de translação é definido como o vetor que sai do ponto em comum das arestas do *toucher* e chega ao vértice final da aresta fixa. No caso III, quando um vértice da aresta fixa toca o meio da aresta orbital, o vetor de translação é definido como o vetor que sai do vértice final da aresta orbital e chega ao ponto em comum das arestas do *toucher*. Nestes



Toucher	Caso	Vetor de Translação
(a_1, b_1)	5	-
(a_1, b_3)	7	-
(a_2, b_1)	2	a_2
(a_2, b_3)	4	a_2
(a_3, b_2)	7	-
(a_3, b_3)	6	$-b_3$
(a_4, b_2)	3	-
(a_4, b_3)	1	$-b_3$

Figura 2.8: Possíveis vetores de translação de A e B.

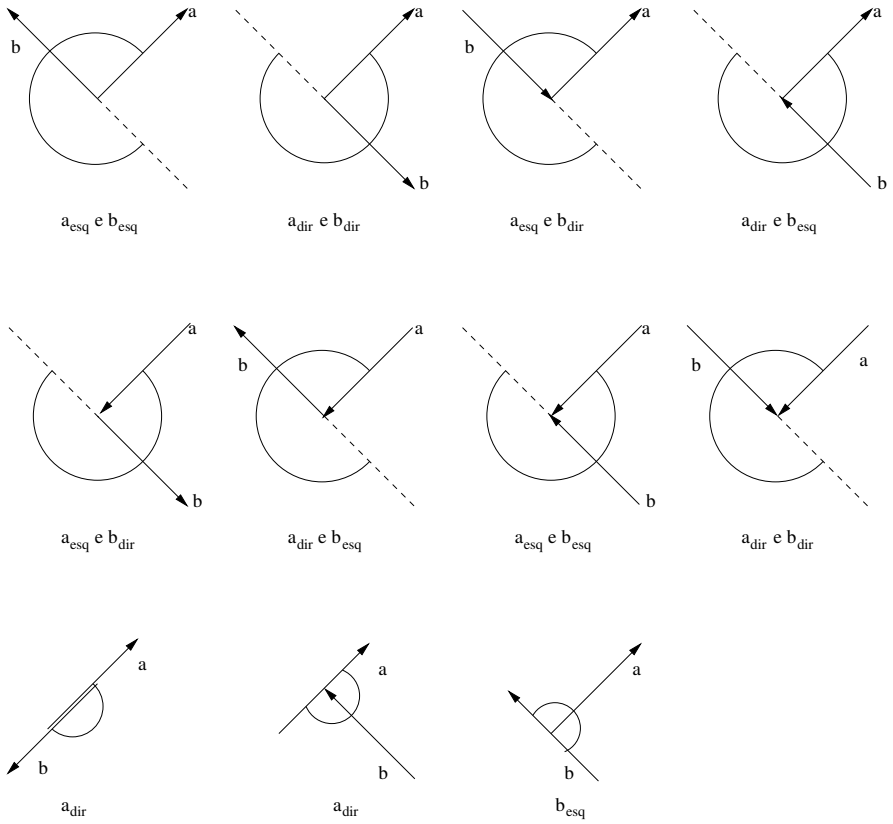
casos, o teste da posição relativa da aresta orbital em relação à aresta fixa não precisa ser feito, porque o vetor de translação sempre é derivado da aresta fixa e da aresta orbital, respectivamente.

Uma vez encontrados os possíveis vetores de translação, é necessário identificar quais vetores são viáveis. Os vetores de translação viáveis são aqueles que não resultam em uma interseção imediata. Para exemplificar a situação, considere a Figura 2.8. Há dois possíveis vetores de translações, a_2 e $-b_3$. Se adotarmos a_2 como vetor de translação, ao transladarmos o polígono B, ocorrerá uma interseção entre a_3 e b_3. Logo, o vetor de translação a_2 é inviável, pois o movimento de translação do polígono B resulta em interseção imediata.

Para determinar se um vetor de translação é viável, o conjunto de *touchers* é novamente utilizado. Para tal, translada-se o vetor de translação para junto do ponto de interseção de cada *toucher*. Dependendo do ângulo de inclinação do vetor, é possível determinar se o movimento correspondente à translação causa interseção imediata. A Figura 2.9 identifica regiões angulares viáveis que podem ocorrer para todos os *touchers* possíveis. Se, para todos os *touchers*, o vetor de translação pertencer à região viável, então o vetor de translação é viável.

Baseando-se no esquema da Figura 2.9, podem-se encontrar as regiões viáveis de *touchers* de dois polígonos quaisquer. Na Figura 2.8, encontramos os possíveis vetores de translação, a_2 e $-b_3$. Para exemplificar como verificar se um vetor é ou não viável, a Figura 2.10 mostra por que o vetor de translação a_2 não é viável. Temos as regiões viáveis para os *touchers* (indicadas pelos arcos) e transladamos o vetor a_2 para junto de todos os *touchers*. Entretanto, nas Figuras 2.10 (5) e (8) o vetor a_2 não pertence à região viável. Logo, a_2 é inviável.

Usualmente, há apenas um vetor de translação viável. Entretanto, vários vetores de translações viáveis podem ser encontrados. Para a escolha de uma dessas translações, Burke *et al.* [8] mantêm a aresta usada para gerar o movimento de translação anterior. Dentre os possíveis vetores de translação, o vetor viável escolhido é o que está mais próximo



Um vetor de translação que tiver direção na região marcada com semi-círculo é viável.

Figura 2.9: Regiões angulares viáveis para translações.

(na ordem das arestas) da translação anterior. Na Figura 2.11, o polígono B chegou ao centro do polígono pela translação a_3. Para a próxima translação, os possíveis vetores são a_4, $-b_1$, a_7, $-b_4$, a_{10}, $-b_3$, a_{13} e $-b_2$. Seguindo a ordem das arestas, a aresta a_4 é a mais próxima de a_3 e é o vetor de translação escolhido.

Ao invés de manter a translação anterior, mantemos o *toucher* que gerou a translação anterior, pois ele já possui as arestas dos polígonos A e B e a partir dele podemos descobrir de qual aresta e de qual polígono a translação foi derivada. Denotamos por $T_A(a_{i_a}, b_{j_a})$ a translação anterior originada pelo *toucher* formado por a_{i_a} e b_{j_a}. Também, denotamos por $T_V(a_i, b_j)$ a translação viável originada pelo *toucher* formado por a_i e b_j.

Se a translação anterior é derivada do polígono A, encontra-se, em um conjunto dos *touchers* viáveis, aquele cuja aresta a_i é a mais próxima da aresta a_{i_a}, de acordo com a ordem das arestas no polígono fixo, A. Se a translação anterior é derivada do polígono B, encontra-se a aresta b_j mais próxima da aresta b_{j_a}, de acordo com a ordem das arestas do polígono orbital, B.

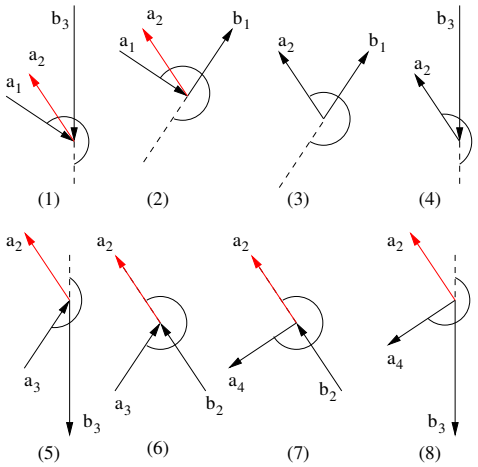


Figura 2.10: Eliminando o vetor de translação a_2.

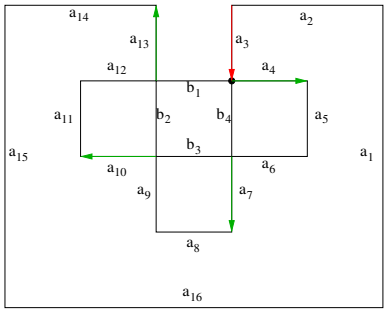


Figura 2.11: Escolha de um vetor de translação entre vários. Fonte: [8].

Depois da escolha do vetor de translação viável, o último passo antes de transladar o polígono B é a poda. Isso é necessário porque transladar B através do vetor de translação inteiro pode resultar em interseções entre os polígonos A e B.

A poda consiste em projetar o vetor de translação em cada vértice do polígono B, de forma que o ponto inicial do vetor coincida com o vértice de B, e testar se existem interseções entre o vetor transladado e as arestas do polígono A. Quando há interseção, uma nova translação é calculada por meio da seguinte fórmula:

$$\textit{Nova Translação} = \textit{Ponto de Interseção - Ponto Inicial da Translação Original} \quad (2.1)$$

Deve-se também projetar o vetor de translação em cada vértice do polígono A, de forma que o ponto final do vetor coincida com o vértice de A, e testar se existem interseções entre o vetor transladado e as arestas do polígono B. Em caso de interseção, uma nova translação é calculada por meio da seguinte fórmula

Nova Translação = Ponto Final da Translação Original - Ponto de Interseção (2.2)

Todas as novas translações geradas eliminam interseções entre os polígonos A e B. A translação escolhida é a que possui menor norma. As Figuras 2.12(a) e 2.13(a) mostram casos em que o uso do vetor de translação inteiro resulta em interseção entre os polígonos. As Figuras 2.12(b) e 2.13(b) mostram a projeção do vetor de translação nos vértices dos polígonos B e A, respectivamente.

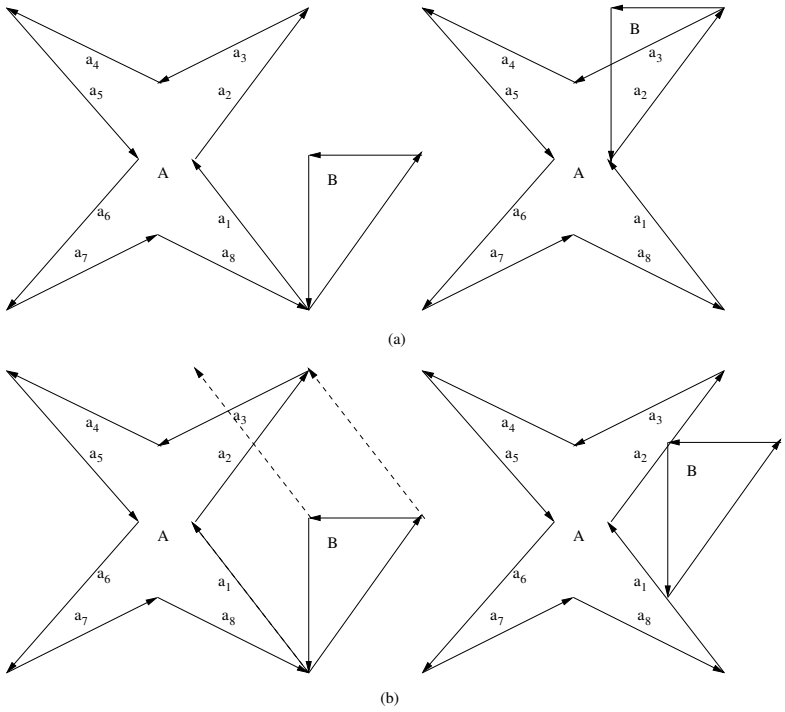


Figura 2.12: (a) Escolha incorreta do vetor de translação. (b) Poda do vetor de translação projetado no polígono B.

O último passo de uma iteração é adicionar o vetor de translação podado ao final da solução parcial do No-Fit-Polygon e transladar o polígono B por esse vetor. Isto moverá o polígono B para o próximo ponto de decisão, onde o processo reinicia a partir da detecção de pares de arestas que se tocam (*touchers*).

2.4.2 Segunda Fase

Na Seção 2.4.1, descrevemos uma abordagem orbital, em que um polígono orbita ao redor de outro usando comparação e translação de arestas. Entretanto, esta abordagem não

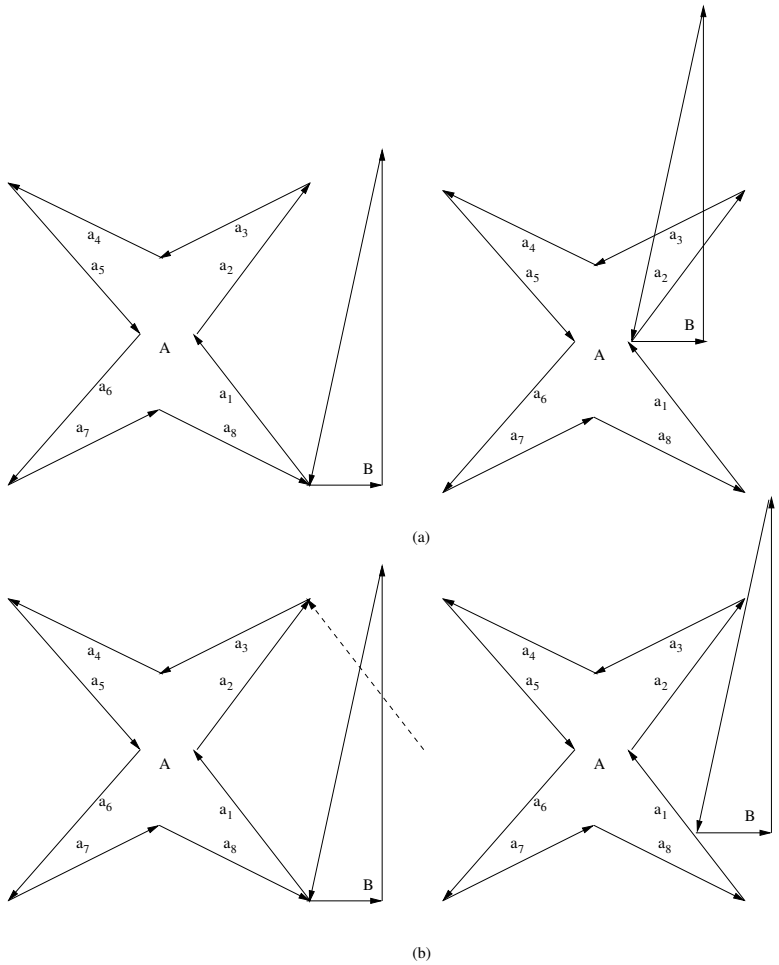


Figura 2.13: (a) Escolha incorreta do vetor de translação. (b) Poda do vetor de translação projetado no polígono A.

gera o No-Fit-Polygon completo de polígonos que possuem concavidades ou buracos. A Figura 2.14(b) mostra que o algoritmo orbital sozinho não encontra o No-Fit-Polygon completo. Os polígonos podem possuir concavidades que geram novas regiões do No-Fit-Polygon, mas que nunca são encontradas porque a entrada da concavidade do polígono fixo é muito estreita. Outra abordagem é necessária para tratar essas possibilidades.

Nesta seção, descrevemos a abordagem usada por Burke *et al.* [8] para encontrar *posições iniciais viáveis*, que não causam interseção entre os polígonos. A partir de uma posição inicial, a abordagem anterior pode ser usada para gerar o interior do No-Fit-Polygon. Por exemplo, se o polígono B puder ser colocado no lugar indicado na Figura 2.14(c), o algoritmo orbital pode ser empregado para gerar a região interna do No-Fit-Polygon. Posições iniciais viáveis são encontradas por meio de uma modificação na abordagem descrita na Seção 2.4.1 e, mais uma vez, o processo envolve a translação do polígono B. Entre-

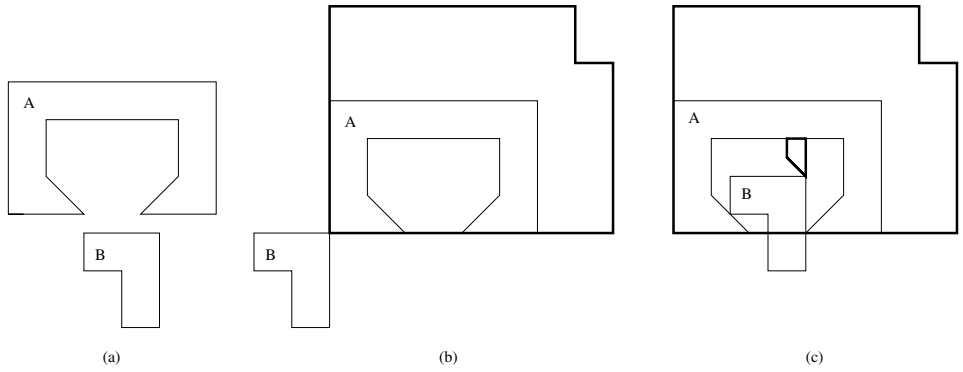


Figura 2.14: (a) Polígonos. (b) No-Fit-Polygon usando a abordagem orbital. (c) No-Fit-Polygon completo.

tanto, o objetivo não é construir o No-Fit-Polygon, mas resolver interseções enquanto transladamos B ao longo de uma aresta e encontrar pontos iniciais viáveis.

Dados um polígono fixo, A, e um polígono orbital, B, o objetivo de cada iteração da segunda fase é encontrar posições iniciais viáveis nas arestas do polígono A que ainda não foram percorridas pelo polígono B na primeira fase. Dada uma aresta e do polígono A, podem-se detectar as possíveis posições de início do polígono orbital B ao longo de e. O processo envolve transladar o polígono B de forma que cada um de seus vértices seja alinhado com o vértice inicial da aresta e. Para cada posição, verifica-se:

- Se os polígonos não se interceptam nessa posição, então esta é uma posição inicial para os dois polígonos.
- Se o polígono B intercepta o polígono A, então mais testes devem ser feitos e o polígono B deve ser transladado pela aresta e até que uma posição que não cause interseção seja encontrada, ou que o final da aresta e seja alcançado.

Caso haja interseções, o primeiro teste a ser feito é verificar se as duas arestas do polígono B que estão conectadas a e estão à direita, à esquerda ou paralelas a e. Se ambas as arestas de B estão à esquerda de e, elas transladariam para dentro do polígono A e podem ser eliminadas imediatamente, pois elas nunca produziriam uma posição inicial viável quando deslizassem ao longo de e.

A Figura 2.15(a) mostra um exemplo onde as arestas, b_3 e b_4, são eliminadas da translação ao longo de a_2 porque b_4 está à esquerda de a_2. A Figura 2.15(b) mostra um exemplo onde ambas as arestas, b_4 e b_5, estão à direita da aresta a_2 e formam um alinhamento válido para encontrar uma posição inicial.

Assumindo que ambas as arestas conectadas a e estão à direita ou paralelas à aresta e e que os polígonos A e B interceptam-se, pode-se tentar resolver a sobreposição transladando

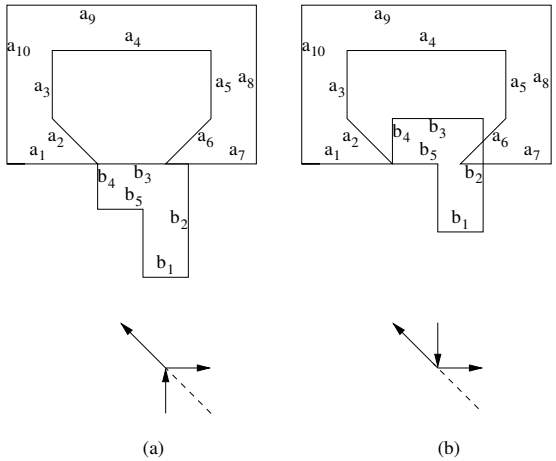


Figura 2.15: Alinhamento do vértice do polígono B na aresta a_2 do polígono S: (a) Alinhamento inválido. (b) Alinhamento válido.

o polígono orbital ao longo do vetor de translação e. Como na seção anterior, a poda é aplicada neste vetor. O vetor de translação resultante pode ser aplicado ao polígono B, que translada ao longo da aresta e. Se os dois polígonos continuam a interceptar-se, então o processo é repetido. Dessa vez, um novo vetor de translação é derivado do ponto que as arestas conectam-se ao vértice final da aresta e. Se a interseção é resolvida, então o ponto de referência do polígono B é uma posição inicial possível. Se a aresta e é percorrida por inteiro e ainda há interseção, então não há posições iniciais viáveis ao longo da aresta e e do par de arestas de B conectados a e. O próximo vértice de B é considerado, até que todas as arestas e vértices possíveis sejam examinados.

Também é importante examinar as arestas do polígono B com os vértices do polígono A para encontrar posições iniciais viáveis. O processo é análogo: os papéis dos polígonos A e B invertem-se e a aresta e passa a representar uma aresta do polígono orbital, B. No entanto, para manter o polígono A fixo, ao invés de transladar o polígono A através das translações derivadas de e, transladamos o polígono B no sentido oposto da translação. Transladamos o polígono B, de forma que cada vértice do polígono A esteja alinhado com o vértice inicial de e. Se não houver interseção entre polígonos, então esta é uma posição inicial viável. Caso contrário, movemos B no sentido oposto da translação obtida de e, como descrito acima.

A Figura 2.16 mostra a resolução de uma interseção entre os polígonos A e B quando usamos a aresta a_2 e o vértice que conecta as arestas b_4 e b_5. A Figura 2.16(a) mostra o processo de poda e identifica o ponto de interseção entre vetor de translação projetado em um vértice do polígono A e o polígono B. Representamos esse ponto por Pt. O polígono B é então transladado pelo vetor de translação que sofreu poda. A posição resultante é mostrada na Figura 2.16(b). Depois de transladar o polígono B, os polígonos não se

interceptam mais e um ponto inicial viável foi encontrado. O ponto inicial é o ponto de referência de B representado na Figura 2.16(b) pelo ponto Ref_B. A abordagem orbital descrita na seção anterior pode ser empregada mais uma vez para gerar novos buracos (*loops* internos) do No-Fit-Polygon.

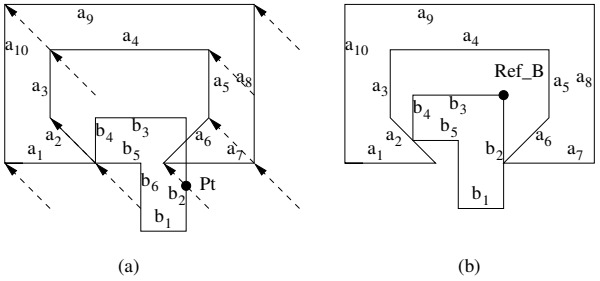


Figura 2.16: Encontrando um ponto de início viável.

Para o cálculo completo do No-Fit-Polygon, primeiro calcula-se a borda externa do NFP_{AB} usando a abordagem descrita na Seção 2.4.1 e, em seguida, encontram-se as posições iniciais das arestas não percorridas de ambos os polígonos, A e B. Para cada uma destas arestas tentamos encontrar um ponto inicial viável. Para cada ponto inicial viável é gerado um *loop* interno do NFP usando também a abordagem descrita na seção 2.4.1. A busca por posições iniciais viáveis repete-se até que todas as arestas tenham sido percorridas, ou até que posições iniciais viáveis não sejam mais possíveis, quando o No-Fit-Polygon completo é retornado.

2.4.3 Pseudocódigo

Nas seções 2.4.1 e 2.4.2, foram descritos o algoritmo que calcula o No-Fit-Polygon e o algoritmo que encontra o próximo ponto inicial para o cálculo dos buracos do No-Fit-Polygon. Aqui, apresentamos os pseudoscódigos desses algoritmos.

Algoritmo 1 NoFitPolygon

Entrada: $PolygonA$, $PolygonB$

1: $Pt_{A(xmin,ymin)}$ = vértice de A de menor ordenada;
2: $Pt_{B(xmax,ymax)}$ = vértice de B de maior ordenada;
3: IFP = posição inicial viável;
4: Bool $bStartPointAvailable$ = true;
5: Point $NFPLoopStartRefPoint$;
6: Point $PolygonB\_RefPoint$;
7: Array[Line[]] $nfpEdges$; // de vetor de segmentos para armazenar as arestas do NFP
8: Int $loopCount$ = 0; // contador para o número de *loops* do NFP
9: **Begin**

10: Colocar os polígonos na IFP usando a translação $Pt_{A(xmin,ymin)}$ - $Pt_{B(xmax,ymax)}$
11: $NFPLoopStartRefPoint = Pt_{B(xmax,ymax)}$;
12: $PolygonB\_RefPoint = Pt_{B(xmax,ymax)}$;
13: **while** (*bStartPointAvailable*) **do**
14: *bStartPointAvailable* = false;
15: // Encontra os segmentos que se tocam em um ponto, gera *touchers*
16: Touchers[] *toucherStructures* = FindToucher(A, B);
17: // Elimina *touchers* inviáveis, que causam interseções imediatas
18: Touchers[] $feasibleTouchers$ = CanMove(A, B, *toucherStructures*);
19: // Poda translações viáveis contra os polígonos A e B
20: Touchers[] $trimmedTouchers$ = Trim($feasibleTouchers$, A, B);
21: // Ordena as translações podadas pelo tamanho
22: Touchers[] $lenghtSortedTouchers$ = Sort($trimmedTouchers$);
23: // Translada B pela maior translação viável
24: B.Translate($lenghtSortedTouchers$[0].Translation);
25: // Adiciona a translação a $nfpEdges$ e marca a aresta percorrida como estática
26: $nfpEdges$[*loopCount*.Add($lengthSortedTouchers$[0].Translation.TRanslation];
27: A.MarkEdge($lengthSortedTouchers$[0].StaticEdgeID);
28: // Se completou um *loop*
29: **if** $NFPLoopStartRefPoint = PolygonB\_RefPoint$ **then**
30: Point *nextStartPoint*;
31: // Encontra próximo ponto de início viável, recalcula o $PolygonB\_RefPoint$
32: *bStartPointAvailable* = **FindNextStartPoint**(A, B, &*nextStartPoint*, &$PolygonB\_RefPoint$);
33: **if** *bStartPointAvailable* **then**
34: // Translada B para o próximo ponto de início, *nextStartPoint*
35: B.Translate($PolygonB\_RefPoint$ - *nextStartPoint*);
36: $NFPLoopStartRefPoint$ = *nextStartPoint*;
37: *loopCount*++;
38: **end if**
39: **else**
40: *bStartPointAvailable* = true; // Continua transladando pela aresta
41: **end if**
42: **end while**
43: $NFP_{A,B}$ = Complete($nfpEdges$); // Constroi o NFP do vetor de arestas
44: **return** $NFP_{A,B}$;
45: **End**

Algoritmo 2 FindNextStartPoint

Entrada: $PolygonA$, $PolygonB$, &*nextStartPoint*, &$PolygonB\_RefPoint$

1: Int $A\_EdgeCount$ = $PolygonA$.EdgeCount;
2: Int $B\_EdgeCount$ = $PolygonB$.EdgeCount;

```
3: Edge staticEdge;
4: Edge movingEdge;
5: begin
6: for (int i = 0; i < A_EdgeCount; i++) do
7:    if (PolygonA.IsEdgeMarked(i)) then
8:       continue;
9:    else
10:      staticEdge = PolygonA.GetEdge(i);
11:      for (int j = 0; j < B_EdgeCount; j++) do
12:         movingEdge = PolygonB.GetEdge(j);
13:         // Translada o polígono B de forma que o início de movingEdge esta no início de staticEdge
14:         PolygonB.Translate(movingEdge.Start - staticEdge.Start);
15:         Bool bFinishedEdge = false;
16:         Bool bIntersects = PolygonB.IntersectsWith(PolygonA);
17:         while (bIntersects and !bFinishedEdge) do
18:            // Translada sobre a aresta até que não haja interseções ou que o final de staticEdge
19:            // tenha sido alcançada
20:            Toucher currentToucher = MakeToucher(staticEdge.Start);
21:            Toucher trimmedToucher = Trim(currentToucher, PolygonA, PolygonB);
22:            PolygonB.Translate(trimmedToucher.Translation);
23:            bIntersects = PolygonB.IntersectsWith(PolygonA);
24:            if (bIntersects) then
25:               if (movingEdge.Start == staticEdge.End) then
26:                  bFinishedEdge = true;
27:               end if
28:            end if
29:         end while
30:         // Marca a aresta como percorrida
31:         staticEdge.Mark(true);
32:         if (!bIntersects) then
33:            // Altera o ponto de referência do polígono B para o próximo ponto de início e retorna
34:            // true
35:            nextStartPoint = movingEdge.Start;
36:            PolygonB_RefPoint = movingEdge.Start;
37:            return true;
38:         end if
39:      end for
40:   end if
41: end for
42: return false;
43: end
```

2.4.4 Resultados

Nós implementamos o algoritmo proposto por Burke *et al.* [8] para gerar No-Fit-Polygons. Nesta seção, ilustramos alguns No-Fit-Polygons obtidos para instâncias problemáticas e apresentamos o tempo gasto para gerar os No-Fit-Polygon de 18 instâncias da literatura.

Burke *et al.* [8] apontam como casos problemáticos o cálculo do No-Fit-Polygon de polígonos que possuem concavidades; de polígonos que possuem encaixes perfeitos, também chamados de "chave-e-fechadura"; de polígonos que possuem passagens exatas para o deslizamento de outros polígonos; e de polígonos que possuem buracos. A Figura 2.17 ilustra esses casos, onde um polígono fixo é representado pela cor azul, um polígono orbital pela cor vermelha e o No-Fit-Polygon pela cor preta.

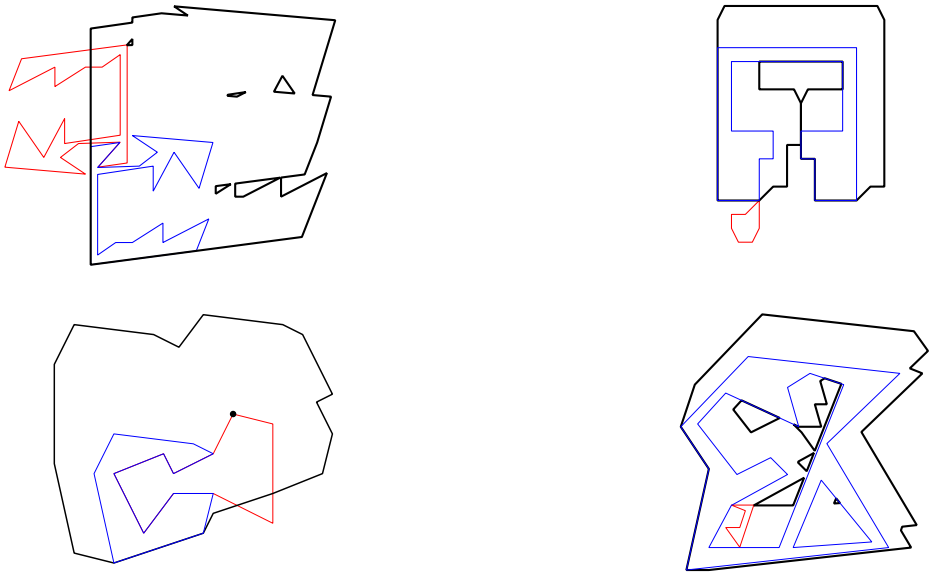


Figura 2.17: No-Fit-Polygons de Casos Problemáticos.

Na Tabela 2.2, apresentamos os resultados do nosso algoritmo para gerar No-Fit-Polygons de algumas instâcias obtidas no *site* da ESICUP[1] – *EURO Special Interest Group on Cutting and Packing*. Para cada instância apresentamos o tempo total para o cálculo dos NFPs (coluna G) e o número de NFPs por segundo (coluna H). O número total de itens (coluna E) é calculado por $E = B * D$ e o número de NFPs (coluna F) por $F = E^2$.

Todos os No-Fit-Polygons foram calculados em um computador Intel Core2 Quad de 2.40GHz e 4GBs de memória utilizando um processador. Burke *et al.* já havia calculado

[1]Disponível em `http://www.fe.up.pt/esicup`.

A	B	C	D	E	F	G	H
Instância	**Número de Itens**	**Restrição de Rotações**	**Rotações por Item**	**Número Total de Itens**	**Número de NFPs**	**Tempo (s)**	**NFPs/s**
ALBANO180	8	180°	2	16	256	2.04	125
ALBANO90	8	90°	4	32	1024	8.98	114
BLAZ1	7	180°	2	14	196	1.14	171
BLAZ2	4	90°	4	16	256	1.94	131
DAGLI	10	90°	4	40	1600	11.37	140
DIGHE1	16	90°	4	64	4096	10.06	407
DIGHE2	10	90°	4	40	1600	4.92	325
FU	12	90°	4	48	2304	5.38	428
JACKOBS1	25	90°	4	100	10000	76.28	131
JACKOBS2	25	90°	4	100	10000	66.87	149
MAO	9	90°	4	36	1296	27.11	47
MARQUES	8	90°	4	32	1024	9.07	112
SHAPES	4	90°	4	16	256	6.4	40
SHAPES0	4	0°	1	4	16	0.46	34
SHAPES1	4	180°	2	8	64	1.79	35
SHIRTS	8	180°	2	16	256	2.04	125
SWIM	10	180°	2	20	400	79.38	5
TROUSERS	17	180°	2	34	1156	4.53	255

Tabela 2.2: Tempo para os cálculos de No-Fit-Polygons.

No-Fit-Polygons para o mesmo conjunto de instâncias em um Pentium 4 de 2GHz com 256MBs de memória. Quando comparamos o tempo de execução da nossa implementação com o tempo de execução alcançado por Burke *et al.* [8], observamos que nossa implementação obteve resultados piores. No trabalho de Burke *et al.* [8] foi desenvolvida uma biblioteca específica para os cálculos trigonométricos, com todas as funções necessárias. Em nossa implementação utilizamos a biblioteca CGAL[2] (*Computational Geometry Algorithms Library*) devido à grande complexidade da implementação. Talvez isso tenha tornado nosso algoritmo mais lento.

[2]Disponível em `http://www.cgal.org`.

Capítulo 3

Rotina Básica para Empacotamento de Itens Irregulares

Neste capítulo, apresentamos uma rotina básica, denominada *Empacota*, para o empacotamento de um item irregular em um recipiente, dado que podem haver itens já empacotados no recipiente. É um procedimento básico que pode ser usado como sub-rotina em algoritmos de empacotamentos em geral. Utilizamos essa rotina em nossa heurística GRASP e em nossa estratégia híbrida para resolver o Problema da Mochila com itens irregulares.

Ao empacotar um novo item em um recipiente, devemos determinar todos os arranjos que o novo item pode assumir com os itens já empacotados, com o objetivo de encontrar todas as posições viáveis para seu empacotamento. A descrição de todos esses arranjos pode ser obtida por meio de No-Fit-Polygons. Algoritmos para o cálculo do No-Fit-Polygon foram descritos nas Seções 2.3 e 2.4.

Entre todas as posições viáveis para o empacotamento do novo item, devemos encontrar aquela que minimiza a área retangular (área do menor retângulo que cobre os itens) e a área convexa (área da menor envoltória convexa que cobre os itens) do novo item e dos itens já empacotados. Para isso, utilizamos o algoritmo da Busca Estendida proposto por Adamowicz e Albano [1].

A rotina proposta empacota itens por meio dos algoritmos de No-Fit-Polygons e do algoritmo da Busca Estendida. Definições importantes para o entendimento deste capítulo serão dadas na Seção 3.1. Detalhes acerca do uso de algoritmos para o cálculo do No-Fit-Polygon serão vistos na Seção 3.2. O Algoritmo da Busca Estendida será visto na Seção 3.3. A rotina *Empacota* será apresentada na Seção 3.4.

3.1 Conceitos

Nesta seção, definimos *Cluster*, *Envoltória Retangular* e *Envoltória Convexa*, que serão utilizados no decorrer desse Capítulo.

Definição 3.1.1 *Cluster é o posicionamento de dois polígonos de forma que eles se tocam e não se sobrepõem.*

Definição 3.1.2 *Envoltória Retangular é o retângulo de área mínima que cobre um conjunto de polígonos.*

Definição 3.1.3 *Envoltória Convexa é o polígono convexo de área mínima que contém todos os pontos de um conjunto de polígonos.*

3.2 Usando o No-Fit-Polygon

Nesta seção, descrevemos a maneira como utilizamos o algoritmo de Burke *et al.* [8] para calcular os No-Fit-Polygons necessários para resolver problemas de empacotamentos.

Como já foi mencionado, o No-Fit-Polygon pode ser utilizado para definir todos possíveis pontos em que um polígono pode ser posicionado em relação a outro polígono, de forma que eles sempre estejam em contato e nunca se sobreponham. Essas são características importantes em problemas de empacotamento, porque permitem a criação de leiautes viáveis e compactos.

Os No-Fit-Polygons são necessários à medida em que os itens são empacotados. O cálculo do No-Fit-Polygon é feito sempre entre dois polígonos. No entanto, durante o empacotamento, pode haver vários itens no recipiente, além do novo item. Consideramos duas abordagens para obter os pontos de posicionamento viáveis do novo item em relação aos itens já empacotados.

Na primeira abordagem, à medida em que os itens são empacotados, construímos o polígono que representa a união dos itens nas posições que foram colocados. Assim, quando um novo item for empacotado, calculamos o No-Fit-Polygon do novo item, i, com o polígono resultante da união dos itens já empacotados, $\cup j$, isto é, calculamos $NFP_{(\cup j,i)}$. Entretanto, à medida em que o número de itens no recipiente aumenta, a união dos itens gera polígonos cada vez mais complexos. Dessa maneira, o cálculo do No-Fit-Polygon torna-se cada vez mais difícil.

Na segunda abordagem, computamos anteriormente os No-Fit-Polygons entre todos os pares de itens em cada rotação possível. Assim, quando um novo item for empacotado, já conhecemos o No-Fit-Polygon, denotado por $NFP(i, j, \theta_i, \theta_j)$, do novo item i, com rotação θ_i, em relação a cada item já empacotado j, com rotação θ_j. Dados um conjunto

$\{j_1, \ldots, j_k\}$ de itens já empacotados e um novo item i a ser empacotado, computamos o NFP obtido pela união $UNFP = \bigcup_{q=1}^{k} NFP(i, j_q, \theta_i, \theta_{j_q})$. Por meio do NFP resultante, a posição do novo item pode ser encontrada. Se a operação de união utilizada for regularizada, isto é, considerar pontos de borda como parte de um polígono, então a união $UNFP$ pode não ser exatamente igual ao $NFP_{(\cup j,i)}$. De qualquer maneira, no entanto, garantidamente não teremos sobreposição de itens, já que $NFP_{(\cup j,i)} \subseteq UNFP$.

Utilizamos as duas abordagens em nossa implementação. Quando usamos a primeira abordagem, calculamos diretamente o No-Fit-Polygon $NFP_{(\cup j,i)}$, que contém apenas os pontos onde o novo item i intersepta algum item já empacotado j. Já quando usamos a segunda abordagem, a união $UNFP$ pode conter também alguns pontos (da borda de $NFP_{(\cup j,i)}$) onde o empacotamento de i não causaria interseção, o que pode levar a um empacotamento diferente em alguns casos. Entretanto, comparando as duas abordagens, percebemos que a segunda é significativamente menos custosa. Isso acontece porque, na primeira abordagem, devemos calcular No-Fit-Polygons cada vez mais complicados à medida em que os itens são empacotados, enquanto na segunda, realizamos apenas operações de união de No-Fit-Polygons pré-calculados, que são mais baratas que o cálculo de No-Fit-Polygons.

3.3 Busca Estendida

Nesta seção, apresentamos o algoritmo da busca estendida proposto por Adamowicz e Albano [1] e algumas modificações que fizemos para usá-lo em nossa implementação. Usamos a busca estendida para encontrar a posição de empacotamento de um item que minimize a área retangular e a área convexa, respeitando as dimensões do recipiente.

Considere dois polígonos A e B. Denominamos *cluster* o arranjo dos polígonos A e B, em alguma posição tal que A e B não se sobrepõem. A área retangular de um conjunto de polígonos é a área da envoltória retangular dos polígonos e, similarmente, a área convexa é a área da envoltória convexa. Um *cluster* ótimo é aquele que minimiza a área retangular dos polígonos e, se houver mais de um *cluster* com área retangular mínima, é aquele que também minimize a área convexa.

Para encontrar a melhor combinação entre os dois polígonos, devemos determinar todos os possíveis arranjos que eles podem assumir, evitando que eles se sobreponham e escolher, nesse arranjo, a posição que minimize a área retangular dos polígonos. A descrição de todos os arranjos que os itens podem assumir é obtida por meio do No-Fit-Polygon. Examinando o NFP_{AB}, é possível encontrar a posição ótima do polígono B em relação ao polígono A.

Uma forma de encontrar uma boa posição do polígono B é verificar os vértices do NFP_{AB} à procura do vértice que gere a menor envoltória retangular. Para cada vértice

do NFP_{AB}, transladamos o polígono B, utilizando seu ponto de referência, e calculamos a envoltória retangular dos polígonos A e B. O vértice do NFP_{AB} que gerar a envoltória retangular de área mínima é o melhor vértice para posicionar o polígono B.

A Figura 3.1 mostra um exemplo do uso do NFP_{AB} para encontrar o ponto de posicionamento de B de forma que a área da envoltória retangular de A e B seja mínima. A Figura 3.1(a) retrata um *cluster* com ponto de referência do polígono B no vértice 3 do NFP_{AB}. O ponto de posicionamento cuja área da envoltória retangular é mínima é obtido na Figura 3.1(b), com o ponto de referência do polígono B no vértice 7 do NFP_{AB}. No exemplo da Figura 3.1, é possível encontrar um *cluster* ótimo considerando apenas os vértices do NFP_{AB}. Entretanto, isso nem sempre é possível.

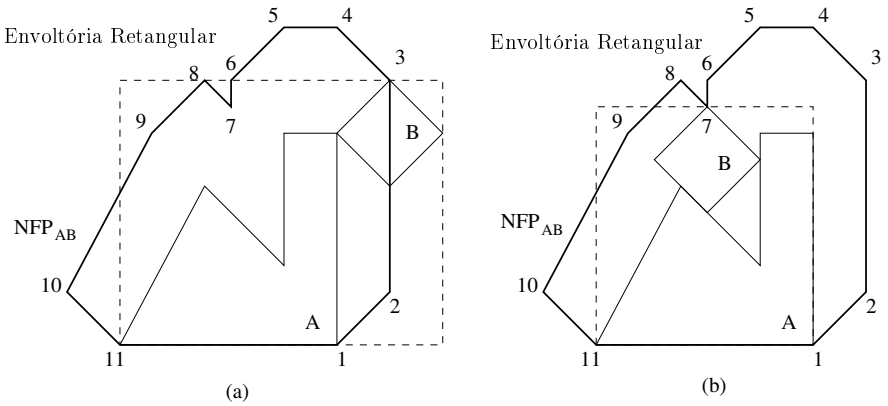


Figura 3.1: Busca de uma posição ótima para B olhando os vértices do NFP_{AB}: (a) *Cluster* com B no vértice 3 do NFP_{AB}. (b) *Cluster* ótimo com B no vértice 7 do NFP_{AB}.

Adamowicz e Albano [1] apresentaram um algoritmo que busca uma posição para o polígono B que minimiza a área da envoltória retangular. O algoritmo é usado em cada vértice do NFP_{AB} com o objetivo de determinar se é possível melhorar o *cluster* pelo movimento do polígono B ao longo das arestas conectadas ao vértice do NFP_{AB}. Este algoritmo é chamado de *busca estendida*.

A seguir, apresentamos algumas definições utilizadas na descrição do algoritmo de busca estendida.

Definição 3.3.1 *O span de um polígono ou conjunto de polígonos são os valores mínimos e máximos de x e y para todos os pontos contidos no polígono ou no conjunto de polígonos.*

Definição 3.3.2 *A região de controle de span correspondente a um vértice do NFP_{AB} (do polígono B relativo ao polígono A) é a região onde, colocando o ponto de referência de B, o span dos polígonos não aumenta.*

Definição 3.3.3 *Dado um vértice* (x_i, y_i) *de um polígono, o vetor de chegada é a aresta direcionada de* (x_{i-1}, y_{i-1}) *para o vértice* (x_i, y_i)*. O vetor de saída é a aresta direcionada do vértice* (x_i, y_i) *para o vértice* (x_{i+1}, y_{i+1})*.*

Para cada vértice do NFP_{AB}, analisamos os vetores de chegada e saída à procura do melhor *cluster*. Assumindo que o polígono B esteja em um vértice i do NFP_{AB}, podemos calcular a envoltória retangular que cobre os polígonos A e B, o *span* e também a *região de controle de span*. Calcula-se a interseção da aresta que chega ao vértice i do NFP_{AB} com a *região de controle de span*. Calcula-se também a interseção da aresta que sai do vértice i do NFP_{AB} com a *região de controle de span*. Se algum dos pontos de interseção gerar uma envoltória retangular de área menor, então este é o ponto onde B deve ser posicionado para que o *cluster* seja ótimo.

A Figura 3.2 mostra uma representação gráfica da aplicação da busca estendida. A Figura 3.2(a) mostra o *cluster* dos polígonos A e B, obtido pelo posicionamento do ponto de referência do polígono B no vértice 3 do NFP_{AB}. Mostra também os valores mínimos e máximos de x e y da envoltória retangular dos polígonos, isto é, o *span*. A Figura 3.2(b) mostra a *região de controle de span* correspondente ao vértice 3 do NFP_{AB}. O uso dessa região garante que os valores máximos de x e y não aumentem e que os valores mínimos de x e y não diminuam. Isso significa que o *span* dos polígonos A e B não aumentará se posicionarmos o polígono B em qualquer posição do NFP_{AB} pertencente à *região de controle de span*. A Figura 3.2(c) mostra os pontos da busca estendida, obtidos pelo cálculo das interseções entre os vetores (o que chega e o que sai do vértice 3 do NFP_{AB}) e a *região de controle de span*. Há dois pontos de interseção, um deles coincide com o vértice 3 do NFP_{AB}, t_1, e o outro está na aresta $3-4$, t_2. Ambos os pontos de interseção são examinados para determinar se uma melhora na área da envoltória retangular pode ser obtida. A área da envoltória retangular é menor no ponto t_2, logo, este é o ponto para o *cluster* ótimo dos polígonos A e B, como mostra a Figura 3.2(d).

O algoritmo da busca estendida também pode ser aplicado para múltiplos polígonos. Primeiro, obtém-se o *cluster* ótimo de dois polígonos. Dada esta posição ótima, pode-se obter um polígono composto, que é formado pela união dos dois polígonos nesta posição. Em seguida, obtém-se o *cluster* do polígono composto com outro polígono e forma-se outro polígono composto. O algoritmo pode ser repetido quantas vezes forem necessárias.

A busca estendida proposta por Adamowicz e Albano [1] verifica, para cada vértice do NFP_{AB}, qual a melhor posição do polígono B para que o *cluster* formado tenha a área da envoltória retangular mínima. Em nossa implementação, consideramos não apenas o objetivo de minimizar a área da envoltória retangular, mas também o objetivo de minimizar a área da envoltória convexa.

Há casos em que temos várias posições possíveis para posicionar o polígono B de forma que a área da envoltória retangular seja a mesma. Quando isso acontece, é interessante

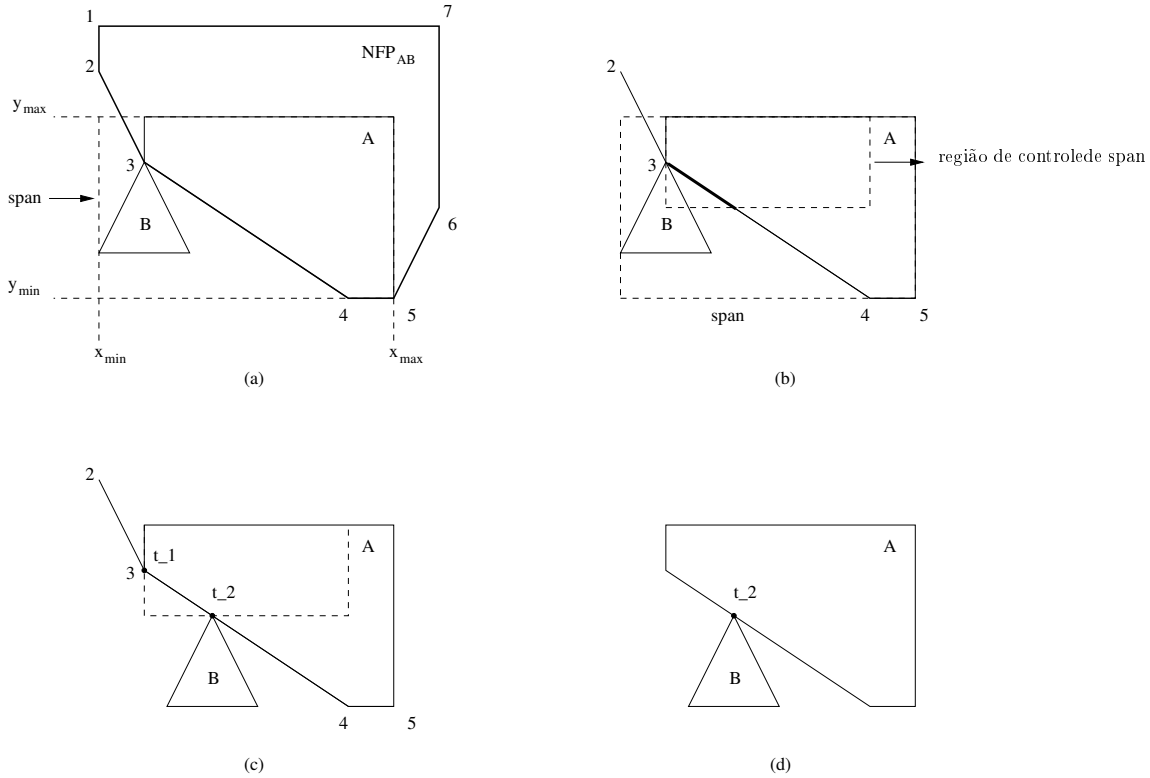


Figura 3.2: *Cluster* usando o No-Fit-Polygon e a Busca Extendida

escolher a posição que deixe os polígonos no leiaute o mais compacto possível. Passamos então a considerar também a área da envoltória convexa.

A Figura 3.3(a) mostra duas posições viáveis para o posicionamento do polígono B, no ponto t e no vértice 6 do NFP_{AB}, que minimizam a área da envoltória retangular do *cluster*. Isto é, a área da envoltória retangular é a mesma tanto para o ponto t quanto para o vértice 6. Em problemas de empacotamento, buscamos o leiaute mais compacto possível. Entretanto, apenas a minimização da área da envoltória retangular não garante que o vértice 6 do NFP_{AB} será escolhido a fim de se obter o leiaute mais compacto. As Figuras 3.3(b) e 3.3(c) mostram a envoltória convexa dos polígonos A e B quando B é posicionado no ponto t e no vértice 6 do NFP_{AB}, respectivamente. Se o interesse for minimizar a área da envoltória retangular e da envoltória convexa, o *cluster* ótimo é obtido posicionando o polígono B no vértice 6 do NFP_{AB}.

Além de minimizar as áreas das envoltórias retangular e convexa, limitamos o espaço de busca do algoritmo de busca estendida para as dimensões do recipiente. Assim, uma posição ótima é obtida quando a altura, H, e a largura, W, da envoltória retangular do polígonos A e B respeitam as dimensões do recipiente e quando a área das envoltórias retangular e convexa são mínimas nesta posição.

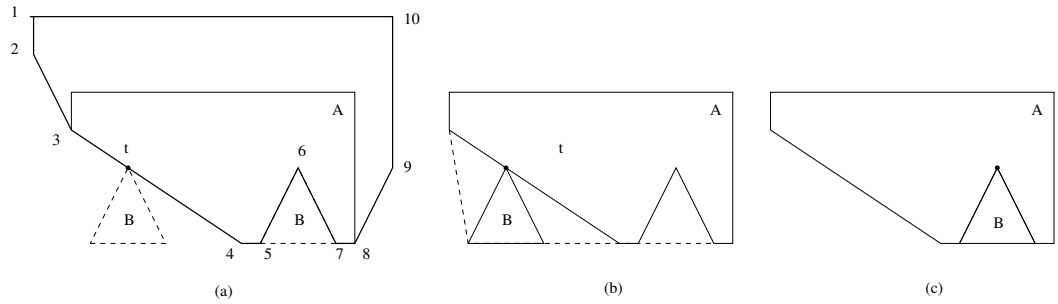


Figura 3.3: *Cluster* formado minimizando a envoltória retangular e a envoltória convexa

Outra consideração feita na implementação é o caso do NFP_{AB} que possui buracos. A área da envoltória retangular e da envoltória convexa é mínima quando conseguimos posicionar o polígono B em um dos buracos do NFP_{AB}. Se o NFP_{AB} possuir buracos, a busca estendida é feita primeiramente nos buracos. Se for possível posicionar o polígono B em um dos buracos, então um *cluster* ótimo foi encontrado. O interesse é posicionar o polígono B em um dos buracos do NFP_{AB} e só fazer a busca estendida na borda externa do NFP_{AB} se nenhuma posição foi encontrada em um dos buracos do NFP_{AB}.

3.4 Rotina para Empacotamento de Itens

O procedimento *Empacota* é uma rotina básica para o posicionamento de um item irregular em um recipiente em alguma rotação permitida.

No empacotamento de um item, devemos verificar a existência de itens no recipiente. O empacotamento é realizado com o objetivo de minimizar a área da envoltória retangular. Quando o recipiente está vazio, o empacotamento torna-se fácil: se a largura e a altura do item são menores que as do recipiente, podemos empacotá-lo. Adicionalmente, rotacionamos o item à procura da rotação que gere a menor área da envoltória retangular do item.

Já no caso em que há itens no recipiente, o empacotamento torna-se mais difícil, porque temos que encontrar uma posição que atenda ao objetivo e respeite as dimensões do recipiente, evitando sobreposições de itens. Nesse caso, encontramos posições viáveis utilizando o algoritmo para o cálculo de No-Fit-Polygons e o algoritmo de Busca Estendida.

Apresentamos o pseudocódigo para o empacotamento de um item no Algoritmo 3 e, em seguida, explicamos seu funcionamento.

Algoritmo 3 Empacota

Entrada: *item*, Θ, H, W
item é o item a ser empacotado

Θ é a lista de rotações
H é a altura do recipiente
W é a largura do recipiente

Saída: $flag$, x, y, θ
$flag$ é uma variável booleana que indica se é ou não possível empacotar $item$
(x, y) é a posição para o empacotamento de $item$
θ é a melhor rotação para $item$

1: **begin**
2: $flag = falso$;
3: $area = 0$;
4: **for** ($\theta_i \in \Theta$) **do**
5: $flag' = falso$;
6: $nfp = vazio$;
7: **for each** (item j no recipiente) **do**
8: $nfp = nfp \cup NFP(item, j, \theta_i, \theta_j)$;
9: **end for**
10: Faz a Busca Estendida no nfp à procura de ponto de posicionamento (x', y') para $item$ na rotação θ_i e atualiza $flag'$;
11: **if** (flag') **then**
12: $area'$ = área da envoltória retangular de $item$ rotacionado θ_i na posição (x', y') e dos itens já empacotados;
13: **if** ($area = 0 \parallel area' < area$) **then**
14: $\theta = \theta_i$;
15: $x = x'$;
16: $y = y'$;
17: $area = area'$;
18: $flag = verdadeiro$;
19: **end if**
20: **end if**
21: **end for**
22: **end**

O algoritmo tem como entrada o $item$ a ser empacotado, a lista $\Theta = \{\theta_1, \ldots, \theta_r\}$ com as r possíveis rotações que o item pode assumir, a altura, H, e a largura, W, do recipiente. Tem como saída a variável booleana $flag$, que indica se o item pôde ser empacotado no recipiente de dimensões H e W, as variáveis x e y, que definem o ponto de posicionamento do item no recipiente, e a melhor rotação θ para o item.

Inicialmente, indicamos que ainda não encontramos nenhuma posição em que $item$ pode ser empacotado, definindo $flag$ como falso (linha 2). Em seguida, testamos o empacotamento de $item$ usando todas as r possíveis rotações, à procura de uma posição de empacotamento viável que gere a menor área da envoltória retangular dos itens (linhas 4 a 20). Definimos $flag'$ como falso para indicar que na rotação θ_i o item não pode ser

empacotado.

Para encontrar a melhor posição de empacotamento, fazemos todos os possíveis empacotamentos para *item*. Para cada rotação possível, encontramos todos os pontos de posicionamento viáveis por meio do cálculo do No-Fit-Polygon de *item* em relação à todos os itens que já estão no recipiente e, em seguida, fazemos a Busca Estendida em busca do melhor ponto de empacotamento.

Na Seção 3.2, descrevemos duas abordagens para o cálculo do No-Fit-Polygon. Utilizamos no pseudocódigo a segunda abordagem. Nessa abordagem, computamos os No-Fit-Polygons de cada par de itens em cada rotação possível. O cálculo de todos os No-Fit-Polygons é feito anteriormente e armazenado, de forma que, a cada chamada de *Empacota*, simplesmente consultamos os No-Fit-Polygons de pares de itens já calculados. Nas linhas 6 a 9, calculamos o No-Fit-Polygon do novo item em relação aos itens que já estão no recipiente. O No-Fit-Polygon resultante é obtido pela união do No-Fit-Polygon de *item* na rotação θ_i com cada item já empacotado j na rotação θ_j. Mais precisamente, a cada passo, unimos o No-Fit-Polygon parcial *nfp* ao polígono armazenado $NFP(item, j, \theta_i, \theta_j)$.

O No-Fit-Polygon fornece todas as posições viáveis para um item, isto é, todas as posições que não causam sobreposição com os itens que já estão no recipiente. Na linha 10, utilizamos o algoritmo da Busca Estendida, descrito na Seção 3.3 para encontrar a melhor posição para *item* rotacionado pelo ângulo θ_i. Esse algoritmo busca o ponto de posicionamento que minimiza as áreas das envoltórias retangular e convexa e que respeita as dimensões do recipiente. Quando é possível empacotar o item no recipiente de dimensões H e W, o algoritmo da Busca Estendida atualiza $flag'$, indicando que é possível empacotar *item* na rotação θ_i, e retorna o ponto de posicionamento (x', y').

A linha 11 verifica se o empacotamento de *item* na rotação θ_i é possível. Caso seja possível, translada-se *item* para a posição (x', y') e calcula-se a área da envoltória retangular do item e dos itens já empacotados, $area'$ (linha 12). A linha 13, verifica se a área encontrada é melhor do que a melhor área conhecida. Se for o caso, nas linhas seguintes, atualizam-se os valores das variáveis θ, x, y e $flag$ com informações do melhor empacotamento encontrado.

A algoritmo termina quando todas as rotações permitidas para *item* tiverem sido testadas e devolve as variáveis $flag$, x, y e θ.

Capítulo 4

Heurística GRASP para o Problema da Mochila com Itens Irregulares

Neste capítulo, apresentamos nossa heurística GRASP para resolver o Problema da Mochila bidimensional com itens irregulares. Assumimos que os itens podem ser representados por polígonos e consideramos, como valor de um item, a área do polígono correspondente.

Na Seção 4.1, apresentamos alguns conceitos utilizados neste capítulo. Na Seção 4.2, fazemos uma descrição geral do GRASP. Na Seção 4.3, descrevemos o algoritmo guloso para gerar a solução inicial, na Seção 4.4, o algoritmo de busca local e, na Seção 4.5, a heurística GRASP.

4.1 Conceitos

Nesta seção, definimos *Ocupação* e *Ocupação Retangular*, que serão utilizados no decorrer desse capítulo.

Definição 4.1.1 *Ocupação é a taxa de utilização do recipiente: área dos polígonos do recipiente pela área do recipiente.*

Definição 4.1.2 *Ocupação Retangular é a taxa de utilização da envoltória retangular: área dos polígonos pela área da envoltória retangular destes polígonos.*

4.2 Descrição do GRASP

GRASP (Procedimento de Busca Aleatório e Adaptativo) é uma meta-heurística desenvolvida para tratar de problemas de otimização combinatória NP-Difíceis. A partir de

uma solução construída por um algoritmo guloso, aplica-se, repetidamente, um método de busca local para obter melhores soluções que as encontradas inicialmente [13, 18, 26].

A heurística é um processo iterativo, constituída em duas fases: a fase de construção e a fase de busca local. A solução resultante é a melhor solução encontrada em alguma de suas iterações.

Na fase de construção, um algoritmo guloso constrói a solução inicial adicionando um novo elemento à solução parcial por vez. Em cada iteração, um conjunto de elementos candidatos é formado por elementos que podem fazer parte da solução parcial. A seleção do próximo elemento é determinada pela avaliação de todos elementos candidatos por uma função de avaliação gulosa. A avaliação dos elementos por esta função gulosa leva à criação de uma lista de candidatos restrita (*Restricted Candidate List – RCL*) formada pelos elementos bem avaliados. O elemento adicionado à solução parcial é selecionado na *RCL*.

Quando um elemento é adicionado à solução parcial, o custo dos elementos candidatos devem ser reavaliados. Essa é uma característica adaptativa da heurística, porque os benefícios de cada elemento mudam com a seleção do elemento anterior. A seleção aleatória de um elemento na *RCL*, não necessariamente o melhor avaliado, é o que caracteriza a componente probabilística do GRASP. Isso permite que diferentes soluções sejam obtidas em diferentes iterações do GRASP.

Não há garantias que a solução inicial gerada pelo algoritmo guloso seja uma solução ótima local. A fase de busca local tenta melhorar a solução inicial de forma iterativa, substituindo sucessivamente a solução corrente por uma solução melhor na vizinhança da solução corrente. O método de busca local baseia-se na exploração de soluções vizinhas, buscando soluções melhores até que uma solução ótima local seja encontrada.

Considere um problema de otimização combinatória de maximização definido por um conjunto $E = \{e_1, \ldots, e_n\}$, um conjunto de soluções viáveis X formado por sequências de itens distintos de E e uma função objetivo a ser maximizada $f : X \to \mathbb{R}$. Se $S \in X$ é uma solução viável, então uma solução vizinha, $S' \in X$, é obtida por meio de permutações ou trocas de elementos da solução corrente, S. A cada iteração, a heurística busca melhorar a solução corrente, ou seja, encontrar uma solução $S' \in X$ tal que $f(S') > f(S)$.

Como uma solução vizinha, S', é construída por meio de permutações ou trocas dos elementos de S, as sequências S e S' diferem em poucos elementos. No entanto, S' pode ser inviável, porque as trocas em S podem produzir soluções que não estão em X. Quando isso acontece, descarta-se a solução S'.

Em cada iteração, é analisada, no máximo, uma quantidade fixa de vizinhos. A busca pelo melhor vizinho pode ser implementada usando as estratégias *best-improving* ou *first-improving*. Na estratégia *best-improving*, todos os vizinhos são avaliados e a busca local seleciona a melhor solução encontrada como solução corrente da próxima iteração. Na es-

tratégia *first-improving*, o primeiro vizinho que melhora a função de otimização é adotado como solução corrente e a busca local continua na próxima iteração.

4.3 Algoritmo Guloso

Algoritmos gulosos funcionam por meio de uma sequência de escolhas. Em cada passo de execução, o algoritmo faz a escolha que parece ser a melhor no momento. Isto é, faz uma escolha ótima para as condições locais na esperança de produzir uma solução ótima para a situação global. Algoritmos gulosos nem sempre produzem soluções ótimas, mas em alguns casos são eficientes e funcionam bem.

Utilizamos uma estratégia gulosa para encontrar uma solução inicial, possivelmente boa, para o Problema da Mochila. Nessa estratégia, várias abordagens podem ser usadas para posicionar cada item a ser empacotado. Bennell e Oliveira [5] descrevem abordagens relacionadas à geometria (Capítulo 2) e regras de posicionamento que determinam a posição em que um item será empacotado. Além disso, descrevem também formas para obter sequências de empacotamento, que determinam a ordem em que os itens são empacotados.

A regra de posicionamento mais popular e utilizada é a heurística *Bottom Left*. Nela, o item é movido horizontalmente para a esquerda enquanto não se sobrepõe com os itens que já foram empacotados. Em seguida, é movido verticalmente para baixo, até que possa ser movido novamente para a esquerda. Esse processo é repetido até que não seja mais possível mover o item para esquerda ou para baixo. A posição final do item é a posição viável, sem sobreposição, mais à esquerda e inferior possível no recipiente. Esta regra também pode ser aplicada em outras direções.

Oliveira *et al.* [25] apresentaram uma regra de posicionamento alternativa ao *Bottom Left*. Nessa regra, para cada item a ser empacotado, o No-Fit-Polygon correpondente é usado para determinar todas as posições viáveis de empacotamento. Para definir em qual posição do No-Fit-Polygon o item será posicionado, consideram-se as posições que minimizam a área da envoltória retangular de dois itens; ou que minimizam o comprimento da envoltória retangular de dois itens; ou, ainda, que maximizam a sobreposição entre a envoltória retangular de dois itens. Para definir qual posição do No-Fit-Polygon o item será posicionado, Oliveira *et al.* [25] utilizaram um algoritmo baseado na busca estendida de Adamowicz e Albano [1].

A sequência em que os itens são posicionados no recipiente influencia a qualidade da solução resultante. A seleção aleatória dos itens é a abordagem mais simples para definir uma sequência de posicionamento. Outras abordagens são as sequências pré-definidas e a seleção dinâmica. Nas abordagens com sequências pré-definidas de itens, busca-se empacotar itens considerados mais difíceis primeiro. Normalmente, são considerados itens de maior dificuldade os itens maiores, mais irregulares, ou que existem em maior quantidade.

Na abordagem de seleção dinâmica de itens, todos os tipos de itens podem ser avaliados para a definição da sequência de posicionamento. Os itens são empacotados usando uma regra de posicionamento, que gera um número de soluções parciais candidatas. As soluções parciais são avaliadas por meio de um critério e o item que gera a melhor solução parcial é selecionado para fazer parte da sequência.

É importante ressaltar que a rotação que os itens podem assumir também influencia na qualidade da solução final. Para cada item a ser empacotado, todas as suas rotações permitidas são calculadas e, para cada rotação, uma regra de posicionamento é utiliza, obtendo, assim, várias soluções parciais. Cada solução parcial é avaliada de acordo com algum critério e o item será empacotado na rotação que obtiver a melhor solução parcial.

O algoritmo guloso implementado utiliza a regra de posicionamento que combina o No-Fit-Polygon e o algoritmo de busca estendida proposto por Adamowicz e Albano [1]. Entretanto, o critério da busca estendida para escolha do ponto de posicionamento no No-Fit-polygon foi melhorado. Além de minimizar a área da envoltória retangular, procuramos minimizar a área da envoltória convexa, nos casos em que há empate da área da envoltória retangular. As sequências de posicionamentos dos itens são definidas de forma aleatória e os itens podem ser rotacionados. A seguir, apresentamos nossa estratégia para gerar nossa solução aleatória gulosa inicial.

4.3.1 Estratégia para Gerar Solução Aleatória Gulosa Inicial

Os itens são empacotados um por um no recipiente, construindo uma solução parcial que aumenta à medida em que é feito o empacotamento. Toda vez que um novo item é empacotado, o No-Fit-Polygon é usado para determinar os pontos de posicionamento viáveis no leiaute e, por meio do algoritmo da busca estendida, um desses pontos é escolhido. O algoritmo guloso verifica em cada passo se há itens viáveis e escolhe o melhor item, a melhor orientação e o melhor ponto de posicionamento.

Apresentamos no Algoritmo 4 o pseudocódigo do algoritmo guloso e, em seguida, descrevemos seu funcionamento.

Algoritmo 4 GeraSolucaoAleatoriaGulosa

Entrada: H, W, I, Θ
H é a altura do recipiente
W é a largura do recipiente
I é a lista de itens
Θ é a lista de rotações

Saída: S, I_1, I_2
S é solução aleatória gulosa inicial
I_1 é a lista com os itens que estão na solução

I_2 é a lista com os itens que não estão na solução

```
1: begin
2: S = vazio;
3: while (I != vazio) do
4:    I' = escolhe p% itens aleatórios de I;
5:    for (i ∈ I') do
6:       flag, x, y, θ = Empacota(i, Θ, H, W);
7:       if (flag) then
8:          Calcula o custo de incluir i em S;
9:       else
10:         I' = I' – i;
11:         I = I – i;
12:         I2 = I2 + i;
13:      end if
14:   end for
15:   if (I'! = vazio) then
16:      Escolhe i ∈ I' de menor custo e obtém a rotação θ e a posição (x, y) correspondentes;
17:      I = I – i;
18:      I1 = I1 + i;
19:      Atualiza S com i rotacionado θ, transladado para (x, y);
20:   end if
21: end while
22: end
```

O algoritmo recebe como entrada a altura, H, e a largura, W, do recipiente, a lista de itens, I, e a lista de rotações, Θ. Como saída, devolve a solução, S, e as listas I_1 e I_2. Essas duas listas são construídas no decorrer do algoritmo: I_1 possui todos os itens que estão em S e I_2 possui os itens que não puderam ser empacotados no recipiente. É importante guardar os itens que não estão na solução porque eles serão usados na busca local.

O algoritmo guloso proposto constrói a solução por meio de amostragens. Em cada iteração (linhas 3 a 21), o algoritmo seleciona uma amostra aleatória de itens e calcula a contribuição de cada item para a solução parcial. O item com melhor contribuição é selecionado e adicionado à solução parcial. Esse tipo de construção é denominado Construção por Amostragem Gulosa [26].

Inicialmente, a solução parcial S é vazia (linha 2). Em cada iteração do laço da linha 3, uma amostra aleatória com $p\%$ dos itens de I é gerada (linha 4). Essa amostra constitui uma *Restricted Candidate List*, representada por I'. Se há itens viáveis em I', o melhor deles é selecionado e adicionado à solução parcial S. Para selecionar o melhor item i de I' , empacota-se cada um deles em cada uma das r rotações permitidas em Θ.

Considere um item $i \in I$ e as rotações permitidas $\Theta = \{\theta_1, \ldots, \theta_r\}$. Cada rotação θ_j

produz um empacotamento diferente para o item i. Assim, empacotamos o item i nas r possíveis rotações e adotamos a rotação θ_j, que gera o melhor empacotamento, como a melhor rotação do item. A melhor rotação θ_j de i é a que produz o empacotamento com menor área da envoltória retangular dos itens.

Para empacotar um item i, chamamos a rotina *Empacota*, descrita na Seção 3.4, na linha 6. A rotina retorna uma variável booleana, *flag*, que indica se foi possível empacotar o item i no recipiente de dimensões H e W. Quando for possível, também é devolvida a posição do item no recipiente (x, y), e a melhor rotação do item, θ.

Se o item i pôde ser empacotado (linha 7), então calculamos o custo de inserir i à solução parcial S. Esse custo depende da ocupação retangular e da área da envoltória retangular do empacotamento obtido de i e S. Quando o empacotamento não puder ser feito, removemos i de I e de I' o adicionamos à lista, I_2.

No final do laço das linhas 5 a 14, I' contém apenas os itens que podem ser adicionados à solução parcial S. Para cada um desses itens, já calculamos o custo, a rotação e a posição no recipiente correspondentes a seu empacotamento. Adicionamos à solução parcial o item i de I' que possui o menor custo, isto é, que possui a maior ocupação retangular e, em caso de empate, que possui a menor área da envoltória retangular. Em seguida, adicionamos i à lista I_1 e à solução parcial S, aplicando a rotação θ e transladando-o para a posição (x, y) do recipiente.

O algoritmo termina quando não há mais itens em I, retornando a solução S e as listas I_1 e I_2.

4.4 Busca Local

Na fase da busca local, tentamos melhorar uma solução, que inicialmente é gerada pelo algoritmo guloso da Seção 4.3, procurando e avaliando soluções vizinhas daquela conhecida atualmente.

Uma solução é uma tupla $(i_1, \ldots, i_{m'})$, onde cada item é empacotado em ordem usando a função *Empacota* vista anteriormente (Seção 3.4). Vizinhos são gerados por meio de alterações na tupla, inserindo, trocando ou permutando itens.

Apresentamos o pseudocódigo da Busca Local no Algoritmo 5.

Algoritmo 5 BuscaLocal

Entrada: H, W, S, I_1, I_2
H é a altura do recipiente
W é a largura do recipiente
S é solução aleatória gulosa inicial
I_1 é a lista com os itens que estão na solução

I_2 é a lista com os itens que não estão na solução

Saída: S

S é solução inicial atualizada após a Busca Local

```
1: begin
2: while (há melhorias em S) do
3:     for (i = 1, ..., x) do
4:         Seleciona um tipo t de vizinho;
5:         Calcula um vizinho de S do tipo t usando I1 e I2;
6:     end for
7:     if (há melhorias em um dos x vizinhos de S) then
8:         Atualiza S;
9:     end if
10: end while
11: end
```

O algoritmo recebe como entrada a altura e a largura do recipiente, H e W, a solução inicial gulosa, S, a lista I_1 com os itens que fazem parte da solução e a lista I_2 com os itens que não pertencem à solução. A saída é a solução S, possivelmente melhorada, após a busca local.

A busca local ocorre enquanto há melhorias na solução S (linhas 2 a 10). Em cada iteração, o algoritmo gera x vizinhos da solução S usando as listas I_1 e I_2 (linhas 3 a 6). Os vizinhos são obtidos por meio de trocas e inserções nessas listas. Consideramos três tipos de vizinhos:

Tipo 1: Um vizinho é obtido trocando dois itens de I_1 de posição, o objetivo é que a nova ordem de itens produza um leiaute com algum espaço livre para inserção de um novo item de I_2 no final de I_1. Mesmo que um novo item não possa ser empacotado, a nova solução será melhor que a anterior se a área retângular do empacotamento for menor.

Tipo 2: Um vizinho é obtido trocando dois itens de I_1 e I_2. As trocas só ocorrem se a área do item $i \in I_2$ for maior que a área do item $j \in I_1$. Quando a troca de i e j é feita e é possível empacotar todos os itens de I_1, a nova solução será melhor que a anterior, porque a ocupação do leiaute será maior.

Tipo 3: Um vizinho é obtido inserindo um item de I_2 em I_1. A nova solução será melhor que a anterior se for possível empacotar todos os itens de I_1, porque o número de itens no recipiente aumenta.

Antes de gerar um vizinho, o algoritmo seleciona um dos tipos de vizinhos listados acima (linha 4). A seleção ocorre de forma aleatória, sendo que, vizinhos do tipo 1 têm a maior probabilidade de serem selecionados, seguidos dos tipos 2 e 3.

É importante destacar que todas as trocas e inserções em I_1 ocorrem apenas nos P_q% últimos itens de I_1. Assim, as alterações ocorrerão no final de I_1 e o processo de empacotamento continua do ponto em que essas alterações ocorreram. Dessa maneira, quanto menor for o valor de P_q, mais rápido o custo de um vizinho será calculado.

Na linha 7, utilizamos a estratégia *best-improving* para selecionar o vizinho: todos os vizinhos são avaliados. Ressaltamos que encontramos um empacotamento ótimo, S', quando for possível empacotar todos os itens de I_1 no recipiente de dimensões H x W e I_2 estiver vazio, caso em que podemos parar a busca local.

Quando avaliamos um vizinho, a lista I_1 possui uma nova configuração de itens e não sabemos se o empacotamento correspondente a I_1 é viável. Assim, primeiro verificamos se é possível empacotar todos os itens de I_1, na sequência, para gerar uma solução candidata S'. Em seguida, se tivermos obtido uma solução candidata S' viável, verificamos se S' é um empacotamento melhor do que S, isto é, se a ocupação de S' é maior do que a de S e, quando a ocupação de S' e S forem iguais, se a área da envoltória retangular de S' é menor do que a de S.

O algoritmo termina quando nenhum vizinho gerado melhora a solução S.

4.5 GRASP

O Algoritmo 6 apresenta o pseudocódigo do GRASP para o Problema da Mochila, composto pela fase de construção da solução inicial e pela fase de busca local.

Algoritmo 6 GRASP

Entrada: H, W, I, Θ
H é a altura do recipiente
W é a largura do recipiente
I é a lista de itens
Θ é a lista de rotações permitidas

Saída: S^*
S^* é solução para o Problema da Mochila

1: **begin**
2: $S^* = vazio$;
3: **for** $(i = 1, \ldots, maxIteracao)$ **do**
4: $S, I_1, I_2 =$ **GeraSolucaoAleatoriaGulosa**(H, W, I, Θ);
5: $S' =$ **BuscaLocal**(H, W, S, I_1, I_2);
6: **if** (custo de S' é melhor que S^*) **then**
7: atualiza S^*;
8: **end if**
9: **end for**

10: **end**

O algoritmo recebe como entrada a altura e a largura do recipiente, H e W, a lista de itens, I, e a lista das possíveis rotações, Θ. As iterações do GRASP ocorrem nas linhas 3 a 9 e o critério de parada é um número máximo de iterações. A linha 4 corresponde à fase de construção do GRASP, que gera a solução inicial gulosa. A linha 5 corresponde à fase de busca local, onde tentamos melhorar a solução inicial. A melhor solução, S^*, é inicialmente vazia e, a cada busca local realizada, é atualizada se a melhor solução encontrada durante a busca for melhor que S^*.

Na linha 6, avaliamos a solução candidata S'. Estamos interessados em maximizar o valor dos itens empacotados no recipiente. Como esse valor está relacionado à ocupação dos itens no recipiente, quanto maior for a área do recipiente ocupada pelos itens, melhor será o empacotamento. Assim, se a ocupação dos itens em S' é maior que a ocupação dos itens em S^*, a solução candidata S' é melhor que S^*. Quando a ocupação dos itens em S' e S^* são iguais, escolhemos a solução mais compacta: se a área da envoltória retangular dos itens de S' é menor que a área da envoltória retangular dos itens de S^*, então S' é mais compacta que S^*. Se a solução candidata for melhor que a solução corrente, atualizamos S^*.

O algoritmo guloso retorna a solução S obtida ao empacotar os itens de I no recipiente de dimensões H e W. Além disso, retorna as listas I_1 e I_2. Essas listas são construídas à medida em que os itens são empacotados: I_1 contém os itens que foram empacotados e I_2 os itens que não couberam no recipiente e que, portanto não fazem parte da solução. Na fase de busca local as listas, I_1 e I_2 são utilizadas para encontrar soluções vizinhas, como descrito na Seção 4.4.

Capítulo 5

Estratégia Híbrida para o Problema da Mochila com Itens Irregulares

Neste capítulo, apresentamos uma estratégia híbrida para resolver o Problema da Mochila com itens irregulares. A estratégia divide-se em passos com empacotamento de itens irregulares e empacotamento de itens regulares. Representamos itens por polígonos e consideramos, como valor de um item, a área do polígono correspondente.

Como descrito acima, são poucos os trabalhos que tratam de problemas de posicionamento de itens irregulares bidimensionais, em especial os problemas de maximização [30]. Alguns trabalhos que tratam do Problema da Mochila com itens irregulares, por exemplo, são os de Anand *et al.* [2], de Martins e Tsuzuki [21, 22, 23, 24] e de Tay *et al.* [28]. Esses trabalhos consideram a versão restrita do Problema da Mochila, em que cada item pode ser empacotado apenas uma vez e, em geral, utilizam técnicas heurísticas, como Simulated Annealing e AG, para encontrar uma solução boa.

Para problemas de posicionamento de itens regulares bidimensionais, por outro lado, há uma vasta gama de referências disponíveis, principalmente para os problemas de minimização (*Bin Packing* e *Strip Packing*) [30]. Para o Problema da Mochila Retangular, além da versão restrita do problema, é tratada também sua versão irrestrita, em que cada item pode ser empacotado várias vezes. São tratadas também outras variações do problema, como cortes guilhotinados e múltiplos estágios. Tanto algoritmos aproximados, quanto algoritmos exatos são utilizados.

Neste trabalho, propomos uma nova estratégia para resolver o Problema da Mochila irrestrito com itens irregulares baseada na geração de itens regulares a partir de itens irregulares. Consideramos o caso em que o valor de um item corresponde à sua área. Primeiro, utilizamos técnicas como No-Fit-Polygon e a Busca Estendida para empacotar itens irregulares em retângulos pequenos com alta taxa de ocupação. Em seguida, utilizamos algoritmos para itens regulares para empacotar os retângulos gerados, obtendo como

resultado um empacotamento dos itens originais.

Algoritmos exatos para o problema do *Rectangular Knapsack*, o Problema da Mochila com itens retangulares, foram propostos por Cintra *et al.* [10]. Eles buscam maximizar o valor dos objetos empacotados e permitem o empacotamento de várias cópias de um mesmo tipo de item. Foram tratadas as versões do problema com cortes guilhotinados, com o algoritmo DP (*Dynamic Program*), e a versão com corte guilhotinado com estágios, com o algoritmo SDP (*k-Staged Dynamic Program*). Os algoritmos de Cintra *et al.* [10] são baseados na recorrência de Beasley [3] e utilizam o conceito de pontos de discretização de Herz [17].

De um modo geral, em DP, consideramos $V(W, H)$ o valor da solução ótima para um recipiente de largura W e altura H. Como somente é necessário analisar um número finito de pontos de discretização de largura e altura, é possível utilizar uma matriz para representar V, o que leva a um algoritmo de programação dinâmica. O algoritmo SDP é análogo, mas considera estágios de cortes.

Na estratégia proposta para resolver o Problema da Mochila com itens irregulares, utilizamos uma sub-rotina para gerar retângulos com boa taxa de ocupação. Utilizamos os algoritmos exatos DP e SDP [10] para o empacotamento desses retângulos. Assim, assumindo que a sub-rotina devolva retângulos com boa ocupação e utilizando os algoritmos exatos para empacotamento de retângulos, espera-se obter boas soluções. Na versão do Problema da Mochila apresentada, permitem-se várias cópias de um mesmo tipo de item irregular.

Para gerar retângulos a partir de itens irregulares, utilizamos o algoritmo descrito na Seção 5.1. Na seção 5.2, apresentamos o algoritmo para empacotar os retângulos gerados, utilizando como sub-rotina os algoritmos DP e SDP.

5.1 Estratégia para Gerar Retângulos

Os itens são empacotados um por um, construindo retângulos parciais, que aumentam de tamanho à medida em que os itens são empacotados. O tamanho do retângulo é limitado por uma altura e uma largura máximas definidas. A cada novo item empacotado, um No-Fit-Polygon é usado para determinar os pontos de posicionamento viáveis do item no retângulo e, por meio do algoritmo da busca estendida, um desses pontos é escolhido. Quando o novo item é empacotado, se a ocupação dos itens no retângulo atingir um bom valor, um retângulo é salvo. Assim, no decorrer do empacotamento, o algoritmo pode gerar vários retângulos de diferentes tamanhos e com diferentes conjuntos de itens empacotados.

Apresentamos o pseudocódigo para gerar vários retângulos de itens irregulares no Algoritmo 7 e, em seguida, descrevemos seu funcionamento.

Algoritmo 7 GeraRetangulos

Entrada: H, W, I, Θ, B
H é a altura máxima do retângulo
W é a largura máxima do retângulo
I é a lista de itens
Θ é a lista de rotações
B é a taxa de ocupação retangular

Saída: *Retangulos*
Retangulos é a lista de retângulos

```
1: begin
2: while (número de iterações≤ maxIteracoes && número de retângulos≤ maxRetangulos) do
3:    rtg = vazio;
4:    I' = escolha p% dos itens aleatórios de I;
5:    for each (i ∈ I') do
6:       flag, x, y, θ = Empacota(i, Θ, H, W);
7:       if (flag) then
8:          Empacote i na posição (x, y) com a rotação θ em rtg
9:          ocupacao = área dos itens empacotados/área da envoltória retangular;
10:         if (ocupacao ≥ B) then
11:            Salve o retângulo rtg com as dimensões da envoltória retangular em Retangulos;
12:         end if
13:      end if
14:   end for
15: end while
16: end
```

O algoritmo recebe como entrada a altura e a largura máximas permitidas para um retângulo, H e W, a lista de itens irregulares a serem empacotados, I, a lista de possíveis rotações dos itens, Θ, e um valor positivo menor ou igual a 1, B. Como saída, é retornada uma lista de retângulos, *Retangulos*. Cada retângulo em *Retangulos* é um empacotamento de itens irregulares com ocupação retangular de pelo menos B.

O algoritmo tenta gerar retângulos até que um número máximo de iterações, $maxIteracoes$, seja alcançado, ou até que um número desejado de retângulos, $maxRetangulos$, seja obtido. Em cada iteração, criamos um retângulo, rtg, inicialmente vazio (linha 3) e obtemos uma lista, I', com $p\%$ dos itens de I (linha 4). O objetivo é empacotar os itens de I' à procura de retângulos que possuam uma boa ocupação.

Os itens de I' são empacotados na ordem em que aparecem em I', nas melhores rotações para o empacotamento. Mais especificamente, suponha que um item $i \in I'$ possa ser rotacionado em vários ângulos possíveis, contidos no conjunto Θ, então um empacotamento diferente é obtido para cada ângulo $\theta \in \Theta$. A rotação θ que gerar o

melhor empacotamento é adotada como a melhor rotação do item. Consideramos como o melhor empacotamento aquele com a menor área da envoltória retangular dos itens.

Na linha 6, a sub-rotina *Empacota*, descrita na Seção 3.4, é usada para empacotar o item i. A função retorna uma variável booleana, *flag*, simbolizando se foi possível empacotar o item i em algum retângulo de dimensões máximas H e W. Se for possível realizar esse empacotamento, então também é devolvida a posição do item no retângulo, (x, y), e a melhor rotação do item, θ.

Quando é possível empacotar o item i (linha 7), ele é rotacionado no ângulo θ e colocado na posição (x, y), passando a integrar o empacotamento do retângulo rtg (linha 8). Em seguida, calculamos a nova ocupação retangular de rtg (linha 9). Quando o item não puder ser empacotado, ele é descartado e passamos para o próximo item.

A ocupação retangular é definida como o somatório das áreas dos itens no retângulo dividido pela área da envoltória retangular desses itens. Na linha 10, se a taxa de ocupação retangular mínima desejada, B, é atingida, então encontramos um retângulo de boa ocupação, que é salvo na lista de retângulos, *Retangulos*. Ressaltamos que, embora o retângulo rtg possua uma ocupação boa nesse momento, continuamos empacotando os itens de I' em rtg, a fim de obter retângulos com maior número de itens e que também possuam uma boa ocupação retangular. Quando obtemos um número desejado de retângulos, ou o número máximo de iterações é atingido, retornamos a lista de retângulos, *Retangulos*.

O algoritmo descrito acima gera retângulos que representam empacotamentos com uma ocupação retangular mínima. No entanto, não há garantia de que sempre é possível obter algum empacotamento de um subconjunto de itens irregulares com a ocupação mínima desejada, nem de que todos os itens serão empacotados. Por um lado, se o valor de B for muito alto, é possível que poucos ou nenhum item seja empacotado. Por outro lado, se o valor de B for muito pequeno, os itens podem ser empacotados em vários retângulos de baixa ocupação, mesmo que fosse possível empacotá-los em retângulos melhores.

No intuito de gerar retângulos de ocupação tão boa quanto possível para cada conjunto de itens, utilizamos a estratégia de geração de retângulos acima como sub-rotina de um outro algoritmo. Inicialmente, tentamos empacotar todos os itens em retângulos com uma taxa de ocupação retangular alta. Em seguida, nós diminuímos a taxa de ocupação desejada de um valor pequeno e tentamos gerar retângulos para os itens ainda não empacotados em nenhum retângulo obtido. Nós repetimos esse processo sucessivamente.

Apresentamos o pseudocódigo para empacotar todos itens irregulares em retângulos no Algoritmo 8 e, em seguida, descrevemos seu funcionamento.

Algoritmo 8 EmpacotamentoEmRetangulos

Entrada: H, W, I, Θ

H é a altura máxima do retângulo
W é a largura máxima do retângulo
I é a lista de itens
Θ é a lista de rotações

Saída: *Retangulos*
Retangulos é a lista de retângulos

1: **begin**
2: $B = B_0$;
3: **while** ($B > 0$ && há itens não empacotados) **do**
4: I' = Itens de I não empacotados em nenhum retangulo;
5: $Retangulos'$ = **GeraRetangulos**(H, W, I', Θ, B);
6: Salve os retângulos de $Retangulos'$ em *Retangulos*
7: $B = B$ - ε;
8: **end while**
9: **end**

O algoritmo recebe como entrada a altura e a largura máximas permitidas para um retângulo, H e W, a lista de itens irregulares a serem empacotados e a lista com as rotações que eles podem assumir, Θ. Como saída, é retornada uma lista de retângulos, *Retangulos*.

Nós iniciamos a taxa de ocupação retangular desejada, B, com um valor dado, B_0 (linha 2). O algoritmo gera retângulos até que todos itens sejam empacotados, ou que a taxa de ocupação retangular seja nula (linha 3). A cada iteração, nós obtemos a lista I' de itens de I ainda não empacotados em nenhum retângulo (linha 4) e tentamos gerar retângulos com taxa de ocupação retangular de pelo menos B. Para isso, utilizamos a sub-rotina *GeraRetangulos*, que retorna uma lista de retângulos $Retangulos'$, que é unida à lista de retângulos de retorno *Retangulos* (linhas 5 e 6).

Na linha 7, a taxa de ocupação retangular desejada é decrementada em um valor dado ε. O valor de ε é escolhido suficientemente pequeno, a fim de empacotar os itens de I' na próxima iteração, mas não gerar retângulos com ocupação retangular menor do que é necessário. Ressaltamos que, quando a taxa de ocupação retangular, B, atinge valores próximos de 0, possivelmente todos os itens já terão sido empacotados. Quando isso ocorrer, ou a taxa de ocupação retangular desejada se anular, a lista de retângulos *Retangulos* é retornada.

5.2 Estratégia para Empacotar Retângulos

Nesta seção, apresentamos o algoritmo pra resolver o Problema da Mochila com itens irregulares. Uma vez que empacotamos itens irregulares em retângulos, podemos empacotar retângulos usando os algoritmo de Cintra *et al.* [10] e obter uma solução.

Apresentamos o pseudocódigo para resolver o Problema da Mochila no Algoritmo 9 e, em seguida, descrevemos seu funcionamento.

Algoritmo 9 ResolveMochila

Entrada: H, W, I, Θ, k
H é a altura do recipiente
W é a largura do recipiente
I é a lista de itens
Θ é a lista de rotações
k é o número de estágios

Saída: S_I
S_I é a solução para o Problema da Mochila com itens irregulares

1: **begin**
2: $Retangulos$ = **EmpacotamentoEmRetangulos**(H, W, I, Θ);
3: Cria uma instância para o problema Retangular Packing a partir de $Retangulos$;
4: S_R = **MochilaRetangular**$(instancia, k)$;
5: Obtemos do empacotamento de retângulos, S_R, o empacotamento de itens irregulares, S_I;
6: **end**

O algoritmo recebe como entrada a altura e a largura do recipiente, H e W, a lista de itens irregulares a serem empacotados, I, a lista de possíveis rotações para os itens, Θ, e um número inteiro k, que indica o número de estágios para o algoritmo de Cintra *et al.* [10]. Como saída, é retornado o empacotamento dos itens irregulares, S_I.

Inicialmente, a rotina *EmpacotamentoEmRetangulos* é chamada para gerar retângulos de dimensões que respeitem os valores máximos, W e H, a partir da lista de itens I e da lista de rotações Θ. Os retângulos são gerados de forma que tenham a melhor ocupação retangular possível. Essa rotina é descrita na Seção 5.1.

Em seguida, cria-se uma instância para o problema *Rectangular Packing*. A instância é definida por uma tupla (W, H, R, v) onde W é a largura do recipiente, H é a altura do recipiente, R é um conjunto de itens retangulares obtidos de Retangulos e v é uma função que associa um valor a cada item de R. Para cada retângulo r de *Retangulos*, criamos um item retangular r' da mesma largura e altura de r e adicionamos r' a R. O valor de r', $v(r')$, é definido como o somatório das áreas dos itens empacotados no retângulo r.

Na linha 4, a rotina *MochilaRetangular* é chamada para empacotar os retângulos recém gerados. Quando é utilizado o algoritmo SDP, k indica o número máximo de estágios de corte. Se for utilizado o algoritmo DP, então deve-se definir $k = 0$, para indicar que não utilizamos estágios de corte. Se as rotações de itens permitidas são pré-definidas em Θ, então não consideramos rotações em retângulos, porque isso alteraria a inclinação de itens irregulares neles empacotados, o que possivelmente levaria a um item rotacionado em um ângulo não definido em Θ. No entanto, os itens podem ser rotacionados quando

são empacotados em retângulos.

O algoritmo *MochilaRetangular* devolve o empacotamento dos retângulos, salvo em S_R. Para recuperar os itens irregulares empacotados em cada retângulo, são utilizados identificadores de retângulos. Assim, podemos descobrir quais itens originais devem fazer parte da solução e, bastando transladar cada item ao interior do retângulo correspondente em sua posição, obtendo o empacotamento de itens irregulares resultante, R_I.

Capítulo 6

Resultados

Neste capítulo, apresentamos os resultados experimentais do Algoritmo GRASP e da Estratégia Híbrida para o Problema da Mochila com itens irregulares. O GRASP e o algoritmo para gerar retângulos foram implementado em linguagem C++ e utilizando a biblioteca geométrica CGAL[1] (*Computational Geometry Algorithms Library*). Todos os testes foram feitos em um computador Intel Core2 Quad de 2.40GHz e 4GB de memória utilizando um processador. Na Seção 6.1, descrevemos as instâncias de teste, na Seção 6.2, configuramos os parâmetros do GRASP e, nas Seções 6.3 e 6.4, apresentamos os resultados obtidos pelo GRASP e pela estratégia Híbrida, respectivamente.

6.1 Instâncias

Não há instâncias públicas disponíveis para o problema da mochila com itens irregulares. Assim, nos testes realizados, consideramos conjuntos de dados encontradas na literatura que são tipicamente utilizados para o problema de *Strip Packing*[2]. Para esse problema, uma instância é composta por um conjunto de itens irregulares, representados por polígonos, pelas possíveis rotações que os itens podem assumir e por uma dimensão fixa do recipiente. O objetivo do problema é empacotar o conjunto de itens minimizando a dimensão variável do recipiente. A Tabela 6.1 apresenta instâncias para o *Strip Packing*.

Estamos interessados no Problema da Mochila para o qual o recipiente é retangular e todas suas dimensões são fixas. Como não há instâncias disponíveis para esse problema na literatura, criamos novas, utilizando conjuntos de dados utilizados em *Strip Packing*, de modo que a largura do recipiente seja a largura da faixa e a altura corresponda ao menor comprimento de faixa obtido por meio de vários algoritmos para *Strip Packing*.

[1]Disponível em `http://www.cgal.org/`.
[2]Disponíveis em `http://www.fe.up.pt/esicup`.

Instância	Número de Itens	Rotações Possíveis	Largura do Recipiente
FU	12	0, 90, 180	38
JACKOBS1	25	0, 90, 180	40
JACKOBS2	25	0, 90, 180	70
SHAPES0	43	0	40
SHAPES1	43	0, 180	40
SHAPES2	28	0, 180	15
DIGHE1	16	0	100
DIGHE2	10	0	100
ALBANO	24	0, 180	4900
DAGLI	30	0, 180	60
MAO	20	0, 90, 180	2550
MARQUES	24	0, 90, 180	104
SHIRTS	99	0, 180	40
SWIM	48	0, 180	5752
TROUSERS	64	0, 180	79

Tabela 6.1: Instâncias do *Strip Packing*.

Dividimos as instâncias criadas em dois grupos. No primeiro, as alturas dos recipientes são os comprimentos da faixa encontrados por algoritmos conhecidos, não necessariamente os melhores, citados por Gomes e Oliveira [16]. O segundo grupo corresponde aos melhores comprimentos de faixa encontrados por Gomes e Oliveira [16]. As Tabelas 6.2 e 6.3 apresentam as instâncias criadas, onde a coluna *Altura do Recipiente* corresponde ao melhor resultado encontrado pelo algoritmo da coluna *Algoritmo*. A Tabela 6.2 apresenta o Grupo 1 e a Tabela 6.3, o Grupo 2.

As instâncias do Grupo 2 também foram utilizadas por Martins [21] no Problema da Mochila com itens irregulares.

6.2 Configuração de parâmetros

O algoritmo GRASP possui vários parâmetros a serem configurados, como o número de iterações, número de vizinhos, percentual de itens escolhidos em listas, etc. Assim, cada configuração de parâmetros pode ser entendida como a configuração de um algoritmo diferente. Portanto, antes de obter os resultados definitivos, testamos nosso algoritmo com diversas configurações de parâmetros, a fim de encontrarmos bons empacotamentos.

Utilizamos as instâncias do Grupo 1 descritas acima para testar e definir diferentes valores para os parâmetros do algoritmo. Testamos um parâmetro por vez: para cada parâmetro, variamos os valores correspondentes e mantivemos os valores de todos os

Instância	**Altura do Recipiente**	**Algoritmo**
FU	34	Hybrid
JACKOBS1	13	Caga
JACKOBS2	28.2	HoopperSA
SHAPES0	63	Jostling
SHAPES1	59	2-Exchange
SHAPES2	27.3	2-Exchange
DIGHE1	138.13	NestLib
DIGHE2	134.05	HoopperGA
ALBANO	10122.63	HoopperSA
DAGLI	65.6	NestLib
MAO	2058.6	HoopperGA
MARQUES	83.6	HoopperNE
SHIRTS	63.13	2-Exchange
SWIM	6568	NestLib
TROUSERS	245.75	2-Exchange

Tabela 6.2: Grupo 1: Instâncias do Problema da Mochila. Fonte: [16]

Instância	**Altura do Recipiente**	**Algoritmo**
FU	31.33	GLSHA
JACKOBS1	12	SAHA
JACKOBS2	24.97	SAHA
SHAPES0	60	SAHA
SHAPES1	56	SAHA
SHAPES2	25.84	SAHA
DIGHE1	100	SAHA
DIGHE2	100	GLSHA
ALBANO	9957.41	SAHA
DAGLI	58.2	SAHA
MAO	1785.73	SAHA
MARQUES	78.48	SAHA
SHIRTS	62.21	GLSHA
SWIM	5948.37	SAHA
TROUSERS	242.11	SAHA

Tabela 6.3: Grupo 2: Instâncias do Problema da Mochila. Fonte: [16]

outros parâmetros fixos. Para cada valor testado, executamos o GRASP para o conjunto de instâncias e calculamos a média da ocupação, isto é, a soma da área de todos os itens empacotados sobre a área do recipiente. A ocupação é um bom indicador para a qualidade do empacotamento porque, no Problema da Mochila considerado, o valor de um item corresponde a sua área e queremos maximizar o valor total dos itens empacotados. Calculamos também o tempo médio gasto pelo algoritmo para as instâncias. Para a escolha do melhor valor de um parâmetro, preferimos aquele que produziu uma boa média de ocupação para o conjunto de instâncias em uma média de tempo aceitável.

Como mencionado no Capítulo 4, o GRASP é composto pelas fases de construção e de busca local. Na fase de construção, temos o algoritmo guloso, dependente apenas do parâmetro, p, que define o percentual de itens avaliados em cada iteração do algoritmo. Na busca local, o algoritmo é dependente do número de vizinhos avaliados, x; das probabilidades dos tipos de vizinhos 1, 2 e 3 ocorrerem, q_1, q_2 e q_3, respectivamente; e do percentual dos itens da tupla que participam do cálculo dos vizinhos, p_q. Por fim, o algoritmo geral do GRASP depende apenas do número de iterações, $maxIteracao$.

Ao testar o parâmetro p, executamos apenas o algoritmo guloso, já que esse parâmetro indica o percentual de itens avaliados nesse algoritmo. Variamos p de 0,05 a 0,4 e, para cada valor, executamos o algoritmo para o conjunto de instâncias mencionado acima. O tempo de execução cresceu linearmente e, embora não exista um comportamento claro, tendo em vista a aleatoriedade do algoritmo guloso, observamos que a ocupação aumentou até que o valor de p atingisse 0,1. Para valores maiores, a média da ocupação passou a diminuir. Desse modo, definimos $p = 0,1$.

Para o teste do parâmetro x, procuramos gerar soluções iniciais iguais ou pelo menos parecidas. Para isso, fixamos $p = 1$ a fim de gerarmos soluções iniciais possivelmente iguais em cada iteração do algoritmo guloso. Fixamos também todos os outros parâmetros e variamos x de 10 até 40. Ao aumentar o valor de x, as médias de ocupação tendem a melhorar e o tempo de execução aumenta rapidamente. Espera-se que a média da ocupação aumente até um certo número de vizinhos, quando aumentar x não melhoraria a solução, no caso de um máximo local ter sido encontrado. No entanto, não podemos escolher valores muito altos para x, pois, em cada melhora obtida, o algoritmo produzirá mais x vizinhos, o que pode aumentar muito o tempo do algoritmo, tornando-o inviável. Baseando-se em nossos testes, observamos que a ocupação tende a se estabilizar a partir de $x = 25$, que foi o valor adotado para x.

Na fase da busca local, ao gerar um vizinho, o algoritmo seleciona um tipo de vizinho para alterar a solução corrente de acordo com os parâmetros q_1, q_2 e q_3. Ao testar esses parâmetros, percebemos que vizinhos do tipo 1, que trocam itens da tupla de posição, devem ocorrer com maior frequência, porque eles compactam o leiaute aumentando a possibilidade de que novos itens sejam inseridos na tupla. Vizinhos do tipo 2, que trocam

itens de duas tuplas, devem ocorrer com menor frequência, pois é incomum que um novo item possa ser empacotado, já que ele é necessariamente maior que o anterior. Vizinhos do tipo 3, que inserem um item na tupla, devem ocorrer com menor frequência ainda, pois observamos que a solução inicial gerada pelo algoritmo guloso já é boa, o que diminui as chances do novo item ser empacotado. Assim, o ideal é que ocorram mais vizinhos do tipo 1 para que o leiaute seja compactado e menos vizinhos dos tipos 2 e 3. No teste, variamos os parâmetros q_1, q_2 e q_3 e mantivemos os outros parâmetros fixos. Encontramos as melhores médias de ocupação e de tempo com $q_1 = 70\%$, $q_2 = 20\%$ e $q_3 = 10\%$.

Outro parâmetro relacionado à busca local é o percentual de itens que participam do cálculo dos vizinhos, p_q. Variamos p_q de 10% até 100%. Apenas o sufixo da sequência de empacotamento com esse percentual de itens será alterado. Dessa maneira, na avaliação de um vizinho, não é necessário empacotar os itens anteriores ao sufixo e, consequentemente, o tempo de execução cresce à medida em que se aumenta p_q. Aumentando p_q, aumenta-se também a possibilidade de gerar vizinhos, possivelmente melhores. Apesar disso, não observamos um padrão de melhora da ocupação com a variação de p_q. Baseando-se nisso, adotamos $p_q = 60\%$ para não restringir muito os itens que participam do cálculo dos vizinhos e para manter o tempo de execução relativamente baixo.

Para o procedimento principal do GRASP, testamos o parâmetro correspondente ao número de iterações, $maxIteracao$, variando de 15 a 40 iterações. Em cada iteração, o algoritmo guloso gera uma solução inicial boa e a busca local melhora essa solução, até encontrar um máximo local. Nos testes, observamos que não há necessidade de muitas iterações, uma vez que a busca local já converge para uma solução com boa média de ocupação e que muitas iterações podem elevar muito o tempo médio de execução. Desse modo, definimos $maxIteracao = 15$.

Ressaltamos que, durante os testes realizados para a definição de parâmetros, existiram grandes dificuldades para a análise do comportamento do algoritmo GRASP com diferentes valores de parâmetros, o que é explicado pela natureza aleatória do algoritmo. Embora pudemos identificar tendências com a variação de certos parâmetros, como o aumento da ocupação com o aumento do valor de x, isso não foi possível para todos os parâmetros. Por exemplo, não é claro se é vantajoso aumentar o valor de p_q, já que, com esse aumento, ora obtemos ocupações melhores, ora obtemos piores, aleatoriamente.

Os parâmetros adotados a partir da realização dos testes e da discussão realizada acima estão resumidos na Tabela 6.4.

6.3 Resultados do GRASP

Nesta seção, apresentamos os resultados obtidos pelo GRASP. Primeiramente, testamos algumas instâncias do Grupo 1 com as duas implementações do GRASP correspondentes

Parâmetro	Valor
p	0.1
x	25
q_1	0.7
q_2	0.2
q_3	0.1
p_q	0.6
$maxIteracoes$	15

Tabela 6.4: Parâmetros do GRASP.

às maneiras de cálculo do No-Fit-Polygon, usando o NFP e o UNFP descritos na Seção 3.2, e comparamos os resultados e os tempos das abordagens. Em seguida, usando a implementação mais rápida (a que utiliza UNFP), testamos todas instâncias dos Grupo 1 e 2. Apresentamos os resultados e mostramos os empacotamentos encontrados que foram ótimos. Finalmente, comparamos rapidamente a abordagem GRASP com trabalhos relacionados.

A Tabela 6.5 apresenta os resultados obtidos pelo GRASP para as duas implementações consideradas. Listamos o número de itens empacotados, a ocupação e o tempo de execução do algoritmo para cada instância para os algoritmos usando o NFP e o UNFP. O tempo de execução do algoritmo que utiliza o UNFP já inclui o tempo necessário para o cálculo prévio dos NFPs entre pares de itens. Um traço na tabela indica que não usamos essa instância, ou que o tempo de execução do algoritmo que calcula o NFP diretamente é inviável para essa instância.

Analisando os resultados obtidos pelo GRASP usando o NFP e o UNFP, podemos observar que os empacotamentos encontrados têm praticamente o mesmo número de itens empacotados e os valores de ocupação obtidos são muito próximos. Isso pode ser observado por meio do gráfico da Figura 6.1. As barras pretas representam a ocupação alcançada quando usamos o UNFP e as barras cinzas a ocupação quando usamos o NFP. A pequena diferença de ocupação é explicada pela aleatoriedade do algoritmo.

Quando comparamos o tempo de execução do algoritmo usando as duas implementações, podemos ver que os tempos são bastante discrepantes. Ilustramos essa diferença no gráfico da Figura 6.2, cujas as barras pretas e cinzas representam o tempo de execução do algoritmo usando o UNFP e o NFP, respectivamente. Observamos que, enquanto encontramos uma solução em tempo viável para todas as instâncias usando o UNFP, para a implementação que calcula o NFP diretamente, isso nem sempre foi possível. Isso acontece porque, para instâncias relativamente grandes (como SWIM), o cálculo do NFP envolve operações cada vez mais complexas, tornando o procedimento inviável. Destacamos, por exemplo, a instância ALBANO, para a qual foram necessárias quase oito horas

Instância		UNFP			NFP		
		Número de Itens	**Ocupação**	**Tempo Total (s)**	**Número de Itens**	**Ocupação**	**Tempo (s)**
A	FU	12	0.8382	22.05	12	0.8382	380.41
B	JACKOBS1	25	0.7538	67.79	25	0.7538	43.50
C	JACKOBS2	25	0.6844	619.51	25	0.6844	932.61
D	SHAPES0	41	0.6016	1603.78	-	-	-
E	SHAPES1	41	0.6424	4094.16	-	-	-
F	SHAPES2	26	0.7289	1066.02	26	0.7375	18887.39
G	DIGHE1	16	0.7240	10.80	15	0.6799	997.72
H	DIGHE2	10	0.7460	0.22	10	0.7460	18.30
I	ALBANO	23	0.8038	975.99	22	0.7901	29273.26
J	DAGLI	29	0.7586	1132.56	-	-	-
L	MAO	20	0.7160	223.33	20	0.7160	8650.28
M	MARQUES	24	0.8274	275.27	24	0.8274	5961.96
N	SHIRTS	96	0.7702	14525.62	-	-	-
O	SWIM	46	0.6427	40869.64	-	-	-
P	TROUSERS	62	0.7866	5844.81	-	-	-

Tabela 6.5: Resultados do GRASP usando o NFP e o UNFP

para completar a execução do algoritmo que usa o NFP, enquanto o algoritmo levou cerca de dezesseis minutos usando o UNFP. Ressaltamos que a implementação com o NFP foi mais rápida para a instância JACKOBS1 (Grupo 1), porque, nesse caso, a solução ótima foi encontrada na primeira iteração.

Em todos os nossos resultados consideramos a implementação que usa o UNFP. As Tabelas 6.6 e 6.7 apresentam os resultados de 10 execuções do algoritmo GRASP para as instâncias de teste dos Grupo 1 e 2, respectivamente. A diferença entre esses grupos está apenas no tamanho dos recipientes, que têm dimensões descritas nas Tabelas 6.2 e 6.3, respectivamente. As colunas das tabelas incluem para cada instância, o melhor número de itens empacotados, a média de número de itens empacotados, a melhor ocupação, a média de ocupação, o tempo médio de execução do algoritmo GRASP, excluído o tempo para o cálculo dos NFPs, e o tempo de pré-processamento para o cálculo de NFPs entre pares de itens.

Para o Grupo 1, nosso algoritmo alcançou resultados ótimos para as instâncias FU, JACKOBS1, JACKOBS2, DIGHE1, DIGHE2, DAGLI, MAO e MARQUES. Para as instâncias SHAPES0, SHAPES1, SHAPES2, ALBANO e SWIM, embora não tenham sido encontrados empacotamentos ótimos, o algoritmo deixou de empacotar um único item. A Figura 6.3 apresenta os empacotamentos ótimos obtidos pelo GRASP para o Grupo 1.

Nas instâncias do Grupo 2, o algoritmo GRASP alcançou resultado ótimo para as instâncias JACKOBS1 e DIGHE2. Os empacotamentos obtidos estão apresentados na

Instância	**Melhor Nº Itens**	**Média Nº Itens**	**Melhor Ocupação**	**Média Ocupação**	**Média Tempo (s)**	**Tempo NFPs (s)**
FU	12	11.80	0.8382	0.8252	128.58	0.26
JACKOBS1	25	25	0.7538	0.7538	71.29	59.49
JACKOBS2	25	25	0.6844	0.6844	148.12	53.80
SHAPES0	42	40.90	0.6222	0.6016	1497.90	51.32
SHAPES1	42	40.90	0.6644	0.6419	3228.65	202.56
SHAPES2	27	26.20	0.7643	0.7383	1061.84	10.48
DIGHE1	16	15.10	0.7240	0.6784	111.43	0.12
DIGHE2	10	10	0.7460	0.7460	0.68	0.15
ALBANO	23	22.80	0.7992	0.7775	1226.11	14.40
DAGLI	30	29.50	0.7731	0.7609	851.57	26.16
MAO	20	20	0.7160	0.7160	161.71	102.91
MARQUES	24	23.60	0.8274	0.8039	1326.75	57.50
SHIRTS	96	96	0.7788	0.7667	14657.89	208.49
SWIM	47	46.50	0.6539	0.6418	50499.54	1088.21
TROUSERS	62	61.20	0.7994	0.7723	6098.13	48.22

Tabela 6.6: Resultados do GRASP para o Grupo 1.

Instância	**Melhor Nº Itens**	**Média Nº Itens**	**Melhor Ocupação**	**Média Ocupação**	**Média Tempo (s)**	**Tempo NFPs (s)**
FU	11	10.60	0.8391	0.7916	186.34	0.26
JACKOBS1	25	24.90	0.8167	0.8125	582.54	59.49
JACKOBS2	24	23.10	0.7180	0.7030	1384.05	53.80
SHAPES0	41	39.70	0.6200	0.6108	1506.22	51.32
SHAPES1	40	39.30	0.6643	0.6359	2874.38	202.56
SHAPES2	26	25.10	0.7572	0.7405	1036.33	10.48
DIGHE1	14	11.30	0.7461	0.6957	65.41	0.12
DIGHE2	10	10	1.0000	1.0000	1.82	0.15
ALBANO	23	22.70	0.8069	0.7625	1167.33	14.40
DAGLI	28	26.70	0.7863	0.7679	1036.63	26.16
MAO	19	19	0.7564	0.7534	1984.25	102.91
MARQUES	23	23	0.8177	0.8112	1806.03	57.50
SHIRTS	96	95.90	0.7903	0.7703	13913.73	208.49
SWIM	45	43.80	0.6619	0.6456	36500.16	1088.21
TROUSERS	61	61	0.7868	0.7738	6283.52	48.22

Tabela 6.7: Resultados do GRASP para o Grupo 2.

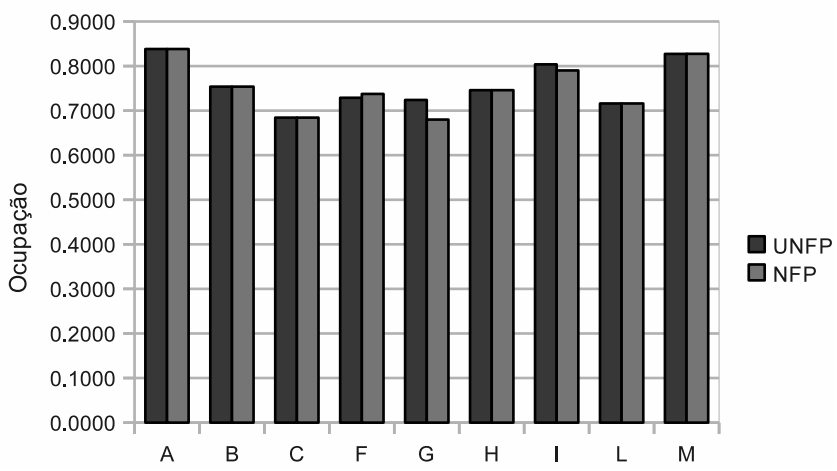


Figura 6.1: Comparação de ocupação.

Figura 6.4. Como esperado, o algoritmo teve maior dificuldade para tratar esse grupo de instâncias, já que as dimensões do recipiente são menores que as do Grupo 1, correspondentes aos melhores valores de comprimento de fita conhecidos para o problema de *Strip Packing*. No entanto, o algoritmo conseguiu manter uma boa média de ocupação, com valores próximos do ótimo. Para as instancias FU, JACKOBS2, ALBANO, MAO e MARQUES, apenas um item não foi empacotado.

Martins [21] também propôs um algoritmo para resolver o Problema da Mochila com itens irregulares, que é baseado em Simulated Annealing. Em seus testes, considerou instâncias adaptadas do problema *Strip Packing*. Ele apresentou resultados para as instâncias correspondentes ao Grupo 2 e obteve empacotamentos ótimos para as instancias ALBANO, JACKOBS1, JACKOBS2, FU, SHAPES0 e DAGLI. Apresentou também gráficos que representam a temperatura e o número de iterações em que o algoritmo converge para o ótimo. No entanto, não foi apresentado o tempo de execução do algoritmo para nenhuma instância, o que não nos permite comparar o desempenho com o algoritmo GRASP. Ressaltamos que melhores empacotamentos podem ser encontrados com nossos algoritmos se utilizarmos parâmetros do GRASP menos conservadores, aumentando o tempo de execução.

6.4 Resultados da Estratégia Híbrida

Nesta seção, apresentamos os resultados obtidos pela estratégia híbrida para resolver o Problema da Mochila com itens irregulares. Os empacotamentos obtidos podem ser muito

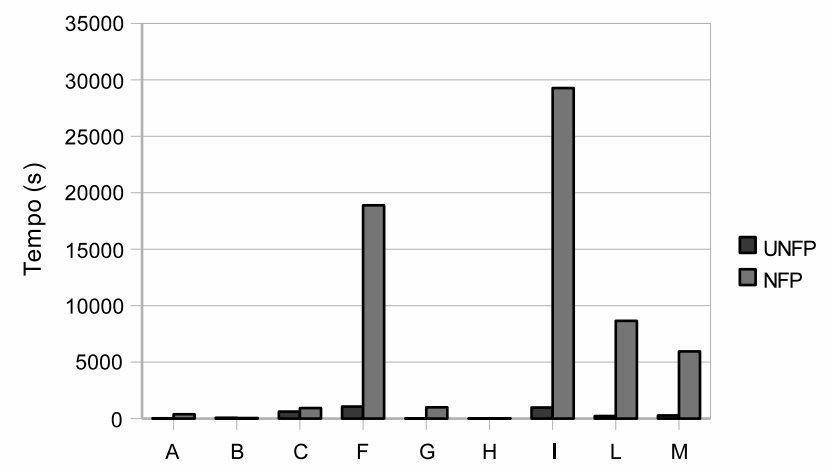


Figura 6.2: Comparação de tempo.

diferentes daqueles obtidos pelo GRASP, pois, neste caso, tratamos da versão irrestrita do problema.

Como mencionado no Capítulo 5, a estratégia híbrida utiliza uma sub-rotina para gerar retângulos a partir de itens irregulares e os algoritmos de empacotamento de retângulo de Cintra *et al.* [10]. O algoritmo para a geração de retângulos depende dos parâmetros: número máximo de iterações, $maxIteracoes$; número máximo de retângulos, $maxRetangulos$; taxa de itens em listas, p; taxa de ocupação retangular inicial, B_0; e valor de decremento da ocupação retangular, ε. Ao contrário do que ocorre no GRASP, os parâmetros desse algoritmo têm menor influência nos resultados. Observamos que isso acontece porque, normalmente, apenas os retângulos de maior ocupação são considerados no empacotamento resultante. Escolhemos, assim, $maxIteracoes = 15$, $p = 0,8$, $B_0 = 0,9$ e $\varepsilon = 0,1$ e definimos $maxRetangulos$ como duas vezes o número de itens da lista de itens.

Para esses valores, o algoritmo começa à procura de retângulos cuja ocupação retangular seja de pelo menos 0,9 e, à medida em que gera retângulos, diminui B de 0,1. Utilizamos $B_0 = 0,9$ em todas as instâncias testadas. No entanto, para instâncias com formas muito irregulares, pode ser interessante definir um valor inicial de B menor, porque, do contrário, o algoritmo poderia procurar retângulos com alta taxa de ocupação sem êxito, o que aumentaria o tempo de execução desnecessariamente.

Nos testes, utilizamos o algoritmo sem estágios de corte, DP, de Cintra *et al.* [10]. Consideramos o conjunto de instâncias do Grupo 1, listadas na Tabela 6.2. Para simplificar a implementação, consideramos recipientes de dimensões inteiras, descartando a parte fracionária de W e H no Grupo 1. Ressaltamos, no entanto, que recipientes com outras

Instância	Melhor Ocupação	Média da Ocupação	Média de Tempo (s)	Tempo NFPs (s)
FU	0.9892	0.9892	5.01	0.26
JACKOBS1	1.0000	0.9870	49.60	59.49
JACKOBS2	1.0000	0.9851	66.08	53.80
SHAPES0	0.6190	0.5873	134.43	51.32
SHAPES1	0.6949	0.6893	248.15	202.56
SHAPES2	0.9284	0.9183	49.37	10.48
DIGHE1	0.7631	0.6909	7.62	0.12
DIGHE2	0.7791	0.7657	3.14	0.15
ALBANO	0.9653	0.9653	37.93	14.40
DAGLI	0.9256	0.9196	50.99	26.16
MAO	0.9812	0.9644	71.53	102.91
MARQUES	0.9606	0.9515	43.76	57.50
SHIRTS	1.0000	1.0000	508.92	208.49
SWIM	0.7590	0.7272	2206.42	1088.21
TROUSERS	1.0000	0.9986	193.54	48.22

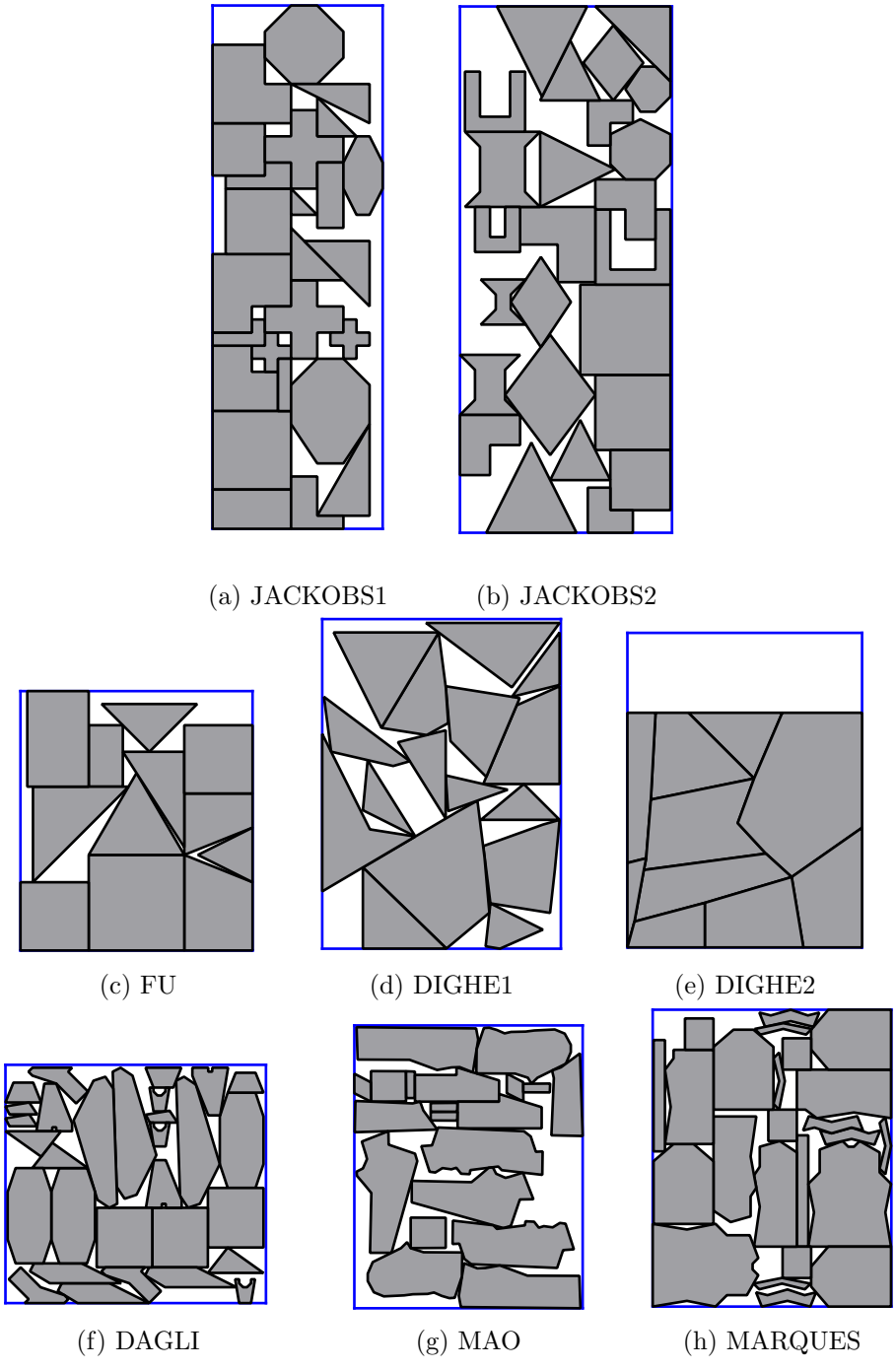
Tabela 6.8: Resultados da Estratégia Híbrida.

dimensões poderiam ser considerados neste caso irrestrito. Consideramos os valores H e W também para limitar o tamanho dos retângulos gerados. Para cada instância, executamos o algoritmo 10 vezes e calculamos a média do tempo de execução e a média da ocupação dos empacotamentos. A Tabela 6.8 apresenta os resultados obtidos.

Analisando os resultados, percebemos que as instâncias que possuem itens retangulares ou itens não muito irregulares produzem bons retângulos e o empacotamento resultante apresenta média de ocupação acima de 90%. São exemplos desses casos as instâncias FU, JACKOBS1, JACKOBS2, SHAPES2, ALBANO, DAGLI, MAO, MARQUES, SHIRTS e TROUSERS. A Figura 6.5 apresenta os empacotamentos de algumas dessas instâncias.

As instâncias SHAPES0, SHAPES1, DIGHE1, DIGHE2 e SWIM são instâncias que possuem apenas itens muito irregulares. O algoritmo gera retângulos com a melhor ocupação possível e empacota os melhores retângulos. No entanto, a ocupação média obtida para essas instâncias são bem menores do que as obtidas para as outras. A Figura 6.6 apresenta os empacotamentos de algumas dessas instâncias.

O tempo total do algoritmo para uma instância é dado pelo tempo gasto para gerar os retângulos e empacotá-los somado ao tempo para o cálculo dos No-Fit-Polygons. O tempo do algoritmo foi razoável para todas as instâncias testadas. Como mencionado acima, instâncias com itens mais irregulares acabam gastando mais tempo. Esse é o caso da instância SWIM, que dentre todas as instâncias foi a que gastou mais tempo.

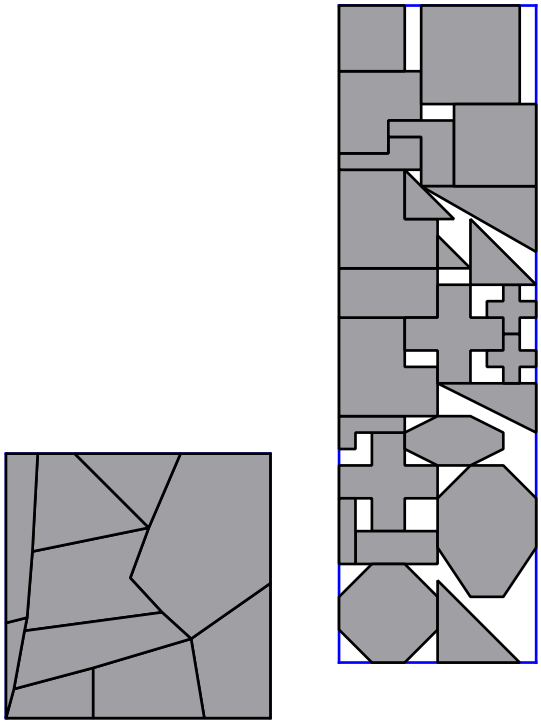


(a) JACKOBS1 (b) JACKOBS2

(c) FU (d) DIGHE1 (e) DIGHE2

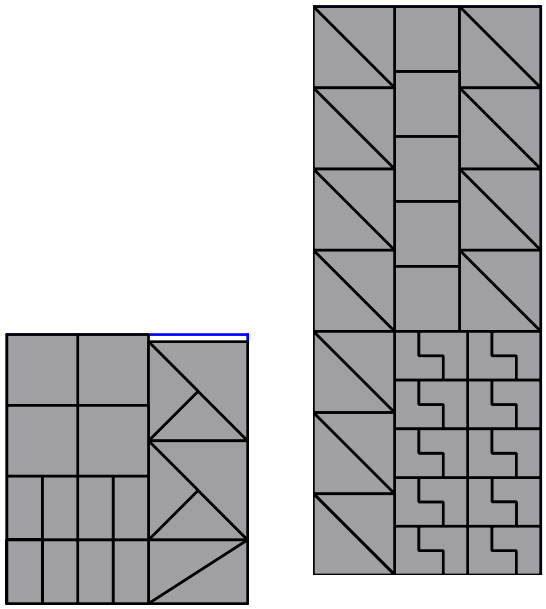
(f) DAGLI (g) MAO (h) MARQUES

Figura 6.3: Empacotamentos ótimos do GRASP para o Grupo 1.



(a) DIGHE2 (b) JACKOBS1

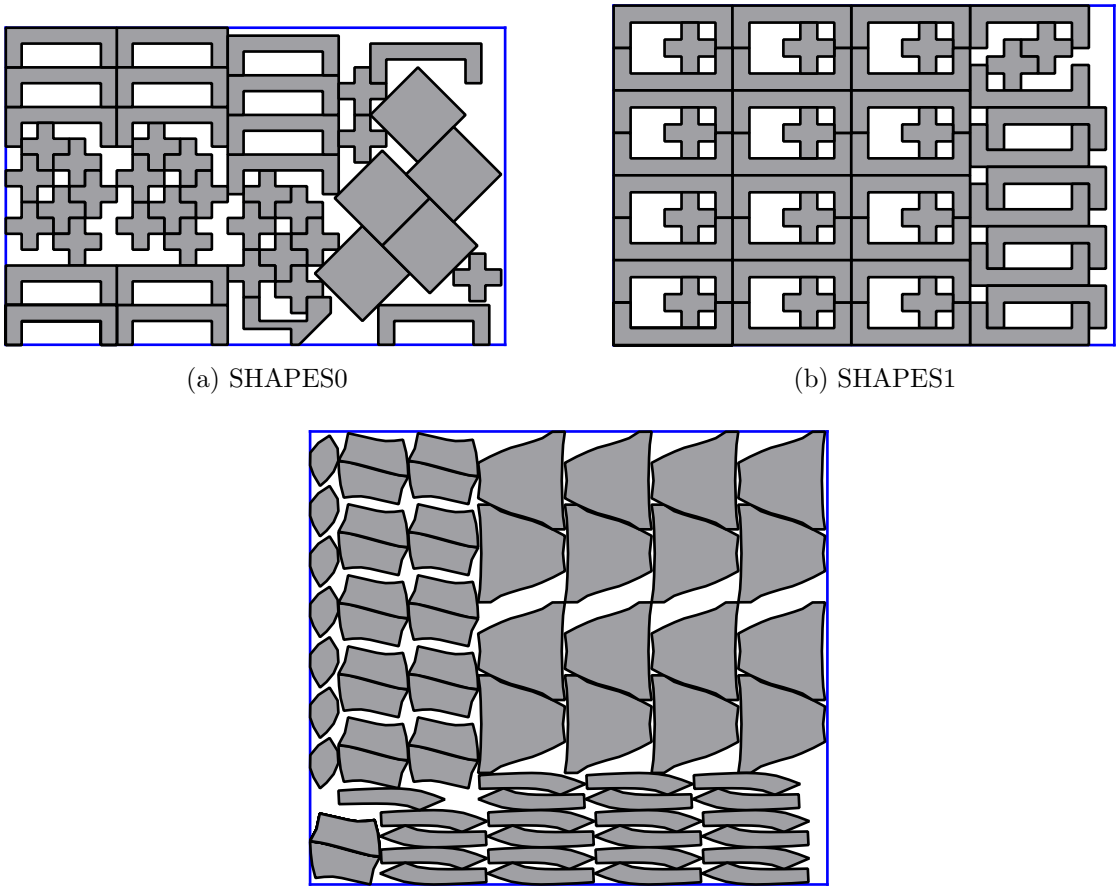
Figura 6.4: Empacotamentos ótimos do GRASP para o Grupo 2.



(a) FU (b) JACKOBS2

(c) SHAPES2

Figura 6.5: Empacotamentos da estratégia Híbrida para Instâncias que possuem retângulos.



(a) SHAPES0

(b) SHAPES1

(c) SWIM

Figura 6.6: Empacotamentos da estratégia Híbrida para Instâncias que não possuem retângulos.

Capítulo 7

Conclusões

Neste trabalho, abordamos o Problema da Mochila com itens irregulares em suas versões restrita e irrestrita, considerando como valor de um item a sua área. O objetivo, portanto, é encontrar um leiaute sem sobreposição de itens irregulares que maximize a área ocupada de um recipiente de dimensões fixas. Como esse problema é NP-difícil, não esperamos encontrar um algoritmo eficiente que o resolva de maneira exata, a menos que P = NP. Dessa maneira, consideramos heurísticas para obter soluções, se não ótimas, de qualidade razoável. Propusemos uma heurística GRASP para a versão restrita do problema e uma estratégia híbrida para a versão irrestrita.

Existem diversas abordagens para obter o posicionamento de itens irregulares de maneira que eles não se sobreponham. A maioria dos algoritmos de empacotamento utilizam o conceito de No-Fit-Polygons para essa resolução. Implementamos o algoritmo proposto por Burke *et al.* [8] e o utilizamos para determinar os pontos de posicionamentos viáveis dos itens nos recipientes em nossos algoritmos. O uso No-Fit-Polygons tem se mostrado eficiente porque permite identificar todos pontos de empacotamento viáveis de um item.

A borda de um No-Fit-Polygon descreve todos os pontos de posicionamentos de um item que não causam sobreposições. Embora qualquer um desses pontos possa ser escolhido, devemos preferir aquele que respeite as dimensões do recipiente e que gere o melhor empacotamento. Como heurística para gerar o melhor empacotamento, escolhemos o ponto que minimiza a área retangular do leiaute. Para isso, utilizamos o algoritmo da busca estendida de Adamowicz e Albano [1]. Adaptamos o algoritmo de busca estendida para que fossem consideradas as dimensões do recipiente e, também, para que o empacotamento obtido fosse mais compacto, minimizando também a área convexa.

Baseando-se na heurística de posicionamento definida acima por meio do algoritmo do No-Fit-Polygon e da busca estendida, definimos uma rotina genérica de empacotamento de itens irregulares, denominada *Empacota*. Dado um empacotamento válido de um subconjunto de items, a rotina adiciona um item retornando um empacotamento viável dos itens

antigos e do novo item. Essa rotina pode ser utilizada para algoritmos de empacotamento em geral e foi usada no algoritmo GRASP e na estratégia híbrida propostos.

Como não há conjuntos de dados para teste para o problema da mochila, testamos nossos algoritmo utilizando instâncias adaptadas do problema *Strip Packing*. Para essas instâncias, fixamos as dimensões do recipiente. Definimos dois grupos de instâncias com graus de dificuldade diferentes, de forma que os recipientes do Grupo 1 possuam dimensões maiores que os recipientes do Grupo 2. As dimensões escolhidas foram valores encontrados para o *Strip Packing* por algoritmos conhecidos. Dessa maneira, os valores ótimos das instâncias para o problema restrito da mochila correspondiam ao empacotamento de todos os seus itens.

O algoritmo GRASP proposto depende de vários parâmetros que devem ser escolhidos. Além disso, duas abordagens para o calculo de No-Fit-Polygons, NFP e UNFP, foram implementadas. Para definir a melhor configuração dos parâmetros e escolher uma abordagem para obter No-Fit-Polygons, realizamos testes utilizando as instâncias do Grupo 1. Por meio dos testes, definimos valores de parâmetros com o objetivo de obter empacotamentos bons em tempo razoável e concluímos que a qualidade da solução é a mesma para as duas abordagens NFP e UNFP. Como o tempo gasto utilizando a abordagem de NFP foi muito maior, continuamos nossos testes utilizando UNFP.

Para as instâncias do Grupo 1, o algoritmo GRASP obteve empacotamentos ótimos para as instâncias FU, JACKOBS1, JACKOBS2, DIGHE1, DIGHE2, DAGLI, MAO e MARQUES. Já para as instâncias do Grupo 2, com recipientes de dimensões menores, conseguiu empacotamento ótimo para JACKOBS1 e DIGHE2. Mesmo para as instâncias em que o algoritmo não obteve resultados ótimos, os empacotamentos obtidos tiveram boa taxa de ocupação, com valores relativamente próximos do ótimo. O tempo de execução do algoritmo foi razoável, com valores bastante similares para os dois grupos as instâncias.

Como as dimensões dos recipientes permitiam que todos os itens fossem empacotados, a princípio, esperava-se obter empacotamentos ótimos para todas as instâncias. Ressaltamos que os parâmetros foram escolhidos de tal forma que o tempo de execução não fosse tão alto. Acreditamos que com diferentes parâmetros, como número de iterações maior ou mais vizinhos, por exemplo, podemos obter resultados melhores. Nesse caso, no entanto, o tempo de execução aumentaria muito, o que não atenderia a nosso objetivo de obter boas soluções em tempo razoável.

A estratégia híbrida para o caso irrestrito do Problema da Mochila foi testada com instâncias do Grupo 1. Embora, para esse caso, não conheçamos os valores ótimos para as instâncias testadas, obtivemos empacotamentos bons para a maioria das instâncias, com taxa de ocupação acima de 90%. Ressaltamos, no entanto, que as taxas de ocupação obtidas dependem das características dos itens contidos em cada instância. Para instâncias que possuem itens retangulares, por exemplo, a taxa de ocupação tende a ser bem maior

do que aquelas para instâncias com itens muito irregulares. A estratégia híbrida mostrou-se eficiente para Problema da Mochila Irrestrito, obtendo altas taxas de ocupação em tempos de execução relativamente baixos.

Trabalhos Futuros: Como trabalhos futuros, há duas possibilidades para melhorar os resultados de nossa heurística GRASP. Na primeira, podemos considerar alternativas para geração de vizinhos e soluções iniciais. Ao gerar vizinhos, poderíamos adicionar outras heurísticas, visando melhorar a qualidade da solução em uma iteração, como tentar empacotar apenas novos itens que cabem no espaço livre do recipiente. A outra possibilidade é varrer um espaço de soluções maior, de forma rápida, para que mais empacotamentos possam ser testados. Logo, a fim de melhorar o desempenho, deveríamos otimizar a implementação, implementando uma biblioteca geométrica eficiente e especializada para as operações mais utilizadas no algoritmo, como união e interseção de polígonos.

Variações do Problema da Mochila também podem ser consideradas. Por exemplo, embora consideramos o valor de um item como a sua área, poderíamos considerar a versão do problema em que os itens possuem valores especificados na entrada. Outra variação do problema envolve demanda de cada tipo de item. Neste trabalho, itens de um mesmo tipo, isto é, com a mesma forma, foram tratados separadamente. Em uma implementação que considerasse demanda, poderíamos evitar a replicação de itens.

Também podemos usar o nosso algoritmo para o problema irrestrito como uma subrotina em um algoritmo de geração de colunas para o problema *Bin Packing*.

Referências Bibliográficas

[1] M. Adamowicz and A. Albano. Nesting two-dimensional shapes in rectangular modules. *Computer-Aided Design*, 8:27–33, 1976.

[2] S. Anand, C. McCord, R. Sharma, and T. Balachander. An integrated machine vision based system for solving the nonconvex cutting stock problem using genetic algorithms. *Journal of Manufacturing Systems*, 18:396–415, 1999.

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

[4] J. A. Bennell and J. F. Oliveira. The geometry of nesting problems: A tutorial. *Journal of the Operational Research Society*, 184:397–415, 2008.

[5] J. A. Bennell and J. F. Oliveira. A tutorial in irregular shape packing problems. *Journal of the Operational Research Society*, 60:93–105, 2009.

[6] J. O. Berkey and P. Y. Wang. Two-dimensional finite bin-packing algorithms. *Journal of the Operational Research Society*, 38:423–429, 1987.

[7] E. K. Burke, R. Hellier, G. Kendall, and G Whitwell. A new bottom-left-fill heuristic algorithm for the two-dimensional irregular packing problem. *Journal of the Operational Research Society*, 54:587–601, 2006.

[8] E. K. Burke, R. Hellier, G. Kendall, and G. Whitwell. Complete and robust no-fit polygon generation for the irregular stock cutting problem. *Journal of the Operational Research Society*, 179:27–49, 2007.

[9] F. R. K. Chung, M. R. Garey, and D. S. Johnson. On packing two-dimensional bins. *SIAM Journal on Algebraic and Discrete Methods*, 3:66–76, 1982.

[10] G. F. Cintra, F. K. Miyazawa, Y. Wakabayashi, and E. C. Xavier. Algorithms for two-dimensional cutting stock and strip packing problems using dynamic programming and column generation. *European Journal of Operational Research*, 191:61–85, 2008.

[11] R. Cuninghame-Green. Geometry, shoemaking and the milk tray problem. *New Scientist*, 123:50–53, 1989.

[12] H. Dyckhoff. A typology of cutting and packing problems. *Journal of the Operational Research Society*, 44:145–159, 1990.

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

[14] J. B. G. Frenk and G. Galambos. Hybrid next-fit algorithm for the two-dimensional rectangle bin-packing problem. *Computing*, 39:201–217, 1987.

[15] A. M. Gomes and J. F. Oliveira. A 2-exchange heuristic for nesting problems. *European Journal of Operational Research*, 141:359–370, 2002.

[16] A. M. Gomes and J. F. Oliveira. Solving irregular strip packing problems by hybridising simulated annealing and linear programming. *Journal of the Operational Research Society*, 171:811–829, 2006.

[17] J. C. Herz. Recursive computational procedure for two-dimensional stock cutting. *IBM J. Res. Dev.*, 16:462–469, 1972.

[18] M. Laguna, T. A. Feo, and H. C. Elrod. A greedy randomized adaptive search procedure for the 2-partition problem. *Journal of the Operational Research Society*, 42:677–687, 1994.

[19] A. Lodi, S. Martello, and D. Vigo. Heuristic and metaheuristic approaches for a class of two-dimensional bin packing problems. *INFORMS Journal on Computing*, 11:345–357, 1999.

[20] A. Mahadevan. *Optimization in computer-aided pattern packing (marking, envelopes)*. PhD thesis, North Carolina State University, 1984.

[21] T. C. Martins. *Estudo do Recozimento Simulado e do Polígono de Obstrução Aplicados ao Problema de Empacotamento Rotacional de Polígonos Irregulares Não-Convexos em Recipientes Fechados*. PhD thesis, Universidade de São Paulo, 2007.

[22] T. C. Martins and M. S. G. Tsuzuki. Solving irregular rotational knapsack problems. In *ISDA '07: Proceedings of the Seventh International Conference on Intelligent Systems Design and Applications*, pages 711–716, 2007.

[23] T. C. Martins and M. S. G. Tsuzuki. Rotational placement of irregular polygons over containers with fixed dimensions using simulated annealing and no-fit polygons. *Journal of the Brazilian Society of Mechanical Sciences and Engineering*, 30:205–212, 2008.

[24] T. C. Martins and M. S. G. Tsuzuki. Simulated annealing applied to the irregular rotational placement of shapes over containers with fixed dimensions. *Expert Systems with Applications*, 37:1955–1972, 2010.

[25] J. F. Oliveira, A. M. Gomes, and J. S. Ferreira. Topos – a new constructive algorithm for nesting problems. *OR Spectrum*, 22:263–284, 2000.

[26] Mauricio G. C. Resende and Ricardo M. A. Silva. Grasp: Greedy randomized adaptive search procedures. Technical report, AT & T Labs Research, 2009.

[27] M. Sipser. The history and status of the P versus NP question. In *STOC '92: Proceedings of the twenty-fourth annual ACM symposium on Theory of computing*, pages 603–618, 1992.

[28] F. E. H. Tay, T. Y. Chong, and F. C. Lee. Pattern nesting on irregular-shaped stock using genetic algorithms. *Engineering Applications of Artificial Intelligence*, 15:551–558, 2002.

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

[30] G. Wascher, H. Hausner, and H. Schumann. An improved typology of cutting and packing problems. *Journal of the Operational Research Society*, 183:1109–1130, 2007.

[31] L. A. Wolsey. *Integer Programming*. Wiley, 1998.

[32] L. A. Wolsey and G. L. Nemhauser. *Integer and Combimatorial Optimization*. John Wiley and Sons, 1988.