UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIA DA COMPUTAÇÃO – BACHARELADO

DISTRIBUIÇÃO OTIMIZADA DE POLÍGONOS EM UM PLANO BIDIMENSIONAL

DENISE BRANDT

BLUMENAU
2011

2011/1-12

DENISE BRANDT

DISTRIBUIÇÃO OTIMIZADA DE POLÍGONOS EM UM PLANO BIDIMENSIONAL

Trabalho de Conclusão de Curso submetido à Universidade Regional de Blumenau para a obtenção dos créditos na disciplina Trabalho de Conclusão de Curso II do curso de Ciência da Computação — Bacharelado.

Prof. Dalton Solano dos Reis , M.Sc. - Orientador

BLUMENAU
2011

2011/1-12

DISTRIBUIÇÃO OTIMIZADA DE POLÍGONOS EM UM PLANO BIDIMENSIONAL

Por

DENISE BRANDT

Trabalho aprovado para obtenção dos créditos na disciplina de Trabalho de Conclusão de Curso II, pela banca examinadora formada por:

Presidente:	Prof. Dalton Solano dos Reis, M.Sc. – Orientador, FURB
Membro:	Prof. Paulo César Rodacki Gomes – FURB
Membro:	Prof. Antonio Carlos Tavares – FURB

Blumenau, 28 de junho de 2011

Dedico este trabalho a todos aqueles que me incentivavam e apoiaram.

AGRADECIMENTOS

À minha família, pelo apoio incondicional dado ao longo de todo o curso.

Aos meus amigos, pelas palavras de incentivo.

Ao professor Aurélio Faustino Hoppe, pela ajuda inicial.

Ao meu namorado João Paulo Gonçalves, que sempre esteve presente.

Ao meu orientador, Dalton Solano dos Reis, pela confiança, ajuda e apoio dedicados ao longo do tempo.

“Só sei que nada sei.”

Sócrates

RESUMO

Este trabalho desenvolve uma solução para o problema de corte e empacotamento. Problema que ocorre em indústrias de manufatura onde moldes de peças são cortados a fim de produzir o produto final. Um algoritmo de encaixe destes moldes é de grande valia devido à economia de matéria-prima e a realocação de mão-de-obra. Este trabalho apresenta os algoritmos *hill climbing* e *tabu search* para a geração de combinações dos polígonos sobre a área disponível. Também é desenvolvido o algoritmo *no-fit polygon* para a detecção de colisão entre os polígonos e o algoritmo *bottom-left fill* para o encaixe dos mesmos. O polígonos são encaixados de forma otimizada e sem sobreposição. Eles devem ter orientação anti-horária.

Palavras-chave: Problema de corte bidimensional. Polígono. Otimização.

ABSTRACT

This paper develops a solution to the cutting and packing problem. Problem that occurs in manufacturing industries where mold pieces are cut to produce the final product. An algorithm to fit these molds has a great value due to the economy of raw materials and the reallocation of manpower. This paper presents the hill climbing and tabu search algorithms for generating combinations of these polygons on the available area. It's also developed the no-fit polygon algorithm for polygons overlapping and the bottom-left fill algorithm for polygons packing. The polygons are packed in an optimal way and with no overlap. They must have anti-clockwise orientation.

Key-words: Two-dimensional cutting problem. Polygon. Optimization.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

Muitas indústrias deparam-se com o desafio de encontrar soluções para o problema de cortar grandes objetos para produzir pedaços menores específicos. O problema de corte consiste em posicionar objetos de formatos diferentes em uma área com dimensões conhecidas a fim de economizar matéria-prima.

Este tipo de problema possui uma enorme gama de aplicação para pesquisa operacional na atualidade. Uma característica marcante é a existência de um grande impacto econômico, pois qualquer decisão tomada significa custos fixos elevados (GOLDBARG; LUNA, 2000, p. 478).

A utilização de ferramentas que realizam a geração de planos de corte é de interesse de indústrias de diversos ramos, como a de móveis, vidros, metais, confecções, espumas, papelões e outras (MARIANO et al., 2009).

Um dos mais importantes aspectos de qualquer rotina de empacotamento automático é o uso de geometria. Isso não é importante apenas por uma perspectiva funcional, mas pode também afetar a qualidade da solução (WHITWELL, 2004, p. 131). Um recurso utilizado para o problema de corte e empacotamento é o conceito de polígono sem encaixe, o qual pode ser usado entre pares de formas para lidar com problemas geométricos. O conceito de polígono sem encaixe pode simplificar significantemente o teste de intersecção entre as duas formas. Este conceito define o caminho que uma forma toma quando está orbitando ao redor de outra forma fixa. Na geração deste caminho, as duas formas sempre se encostam. O polígono gerado a partir desta operação reflete o estado de intersecção entre os dois polígonos (WHITWELL, 2004, p. 129). Pode-se restringir a rotação das formas com o intuito de diminuir a quantidade de rotações, tornando o processo de construção da solução mais rápido.

Diante do exposto, este trabalho desenvolve uma ferramenta que organiza polígonos em um espaço bidimensional, buscando ocupar a menor área possível. Os polígonos podem ser rotacionados de acordo com a parametrização passada para cada problema. São usados algoritmos geométricos para calcular a disposição dos polígonos de forma otimizada. Além disso, são usados algoritmos combinatoriais que buscam uma sequência de polígonos que geram um bom resultado.

1.1 OBJETIVOS DO TRABALHO

Este trabalho tem como objetivo desenvolver uma ferramenta capaz de posicionar polígonos no espaço bidimensional a fim de ocupar a menor área possível.

Os objetivos específicos do trabalho são:

a) ter como resultado uma solução que não necessariamente precisa ser a solução ótima;
b) permitir visualizar o resultado do processo de encaixe;
c) solucionar o encaixe dos polígonos em um tempo aceitável.

1.2 ESTRUTURA DO TRABALHO

Este trabalho está dividido em quatro capítulos, sendo que o segundo capítulo apresenta a fundamentação teórica com conceitos de metaheurística, geometria computacional, otimização combinatorial, técnica de empacotamento de polígonos e detecção de colisão. Além disso, o capítulo também expõe detalhes de trabalhos correlatos.

O terceiro capítulo trata do desenvolvimento do algoritmo de empacotamento, iniciando com os requisitos e a especificação da aplicação. Fazem parte desta especificação os diagramas de casos de uso, de classes, de atividades e de sequência. Ainda no terceiro capítulo, são comentados os resultados e problemas encontrados durante a implementação da ferramenta.

Por fim, no quarto capítulo são apresentadas as conclusões finais sobre o trabalho e sugestões para extensões.

2 FUNDAMENTAÇÃO TEÓRICA

Para o entendimento deste trabalho são apresentados o problema do corte e empacotamento e a metaheurística. Logo em seguida são explicados o conceito de geometria computacional. Na sequência é explicada a construção do *no-fit polygon* e sua utilização na detecção de colisão de polígonos. Em seguida é detalhado o funcionamento do algoritmo de empacotamento *bottom-left fill* e a utilização do *no-fit polygon* em seu processamento. É apresentado também o conceito de busca local e, na sequência, é explanado o funcionamento do *hill climbing* e do *tabu search*, apresentando suas características e pontos em comum.

Por fim, em trabalhos correlatos, é mencionada uma ferramenta existente no mercado e trabalhos com funcionalidades semelhantes às da ferramenta implementada.

2.1 PROBLEMA DE CORTE E EMPACOTAMENTO

Problemas de empacotamento ocorrem em muitas situações diferentes no cotidiano. Um indivíduo é submetido a várias situações onde as habilidades de empacotamento são requeridas, tais como colocar roupas em uma mala ou colocar alimentos em um congelador. Os seres humanos parecem ser capazes de resolver o problema de empacotamento relativamente bem usando a intuição e a percepção espacial. No entanto, em um ambiente industrial, onde há uma infinidade de casos semelhantes de problemas de empacotamento, geralmente é inviável ou financeiramente ineficaz resolver tais problemas manualmente. Diferentemente de humanos, computadores não têm intuição ou noção espacial e, portanto, estratégias algorítmicas precisam ser desenvolvidas para a geração de empacotamentos (WHITWELL, 2004, p. 9). No cenário industrial, isso tem benefícios financeiros especiais quando os recursos têm elevado custo unitário. Relaxamentos de problemas de corte também se prestam à comunidade acadêmica por se mostrarem simples e bem-formados, mas continuam NP-Completos e, assim, acredita-se que não pode ser resolvido com um algoritmo de tempo polinomial (WHITWELL, 2004, p. 10). Apesar disso, na maioria das aplicações industriais os objetivos são semelhantes. Eles consistem em produzir empacotamentos de boa qualidade com o objetivo de maximizar a utilização de material e, portanto, minimizar o desperdício (WHITWELL, 2004, p. 17).

Apesar do problema de empacotamento de retângulos ter recebido um foco considerável na pesquisa acadêmica desde a formulação do corte e empacotamento nos anos 50, a variante irregular do problema não recebeu muito interesse até os últimos 25 anos. Isso foi inevitável devido à geometria irregular que precisava ser modelada, e às operações geométricas que exigiam recursos computacionais elevados necessários para realizar manipulações sobre as formas. Entretanto, como o problema da irregularidade ocorre em diversas indústrias importantes, isso acabou sendo considerado interesse acadêmico e industrial. Devido à evolução da tecnologia computacional, a área de pesquisa avançou rapidamente e aumentou o número de novas estratégias sendo desenvolvidas e disseminadas entre a comunidade acadêmica e indústria (WHITWELL, 2004, p. 33).

Segundo Whitwell (2004, p. 57), a indústria têxtil é uma das áreas que pesquisa sobre o empacotamento automático. O corte de roupas a partir de rolos de materiais oferece muitas dificuldades e nuances que precisam ser lidadas pela automatização. Alguns problemas podem incluir padrões de rolos de materiais, com isso, limitações rotacionais devido à orientação que a roupa deve ser cortada. São comuns formas altamente irregulares, o que requer cálculos geométricos complexos, e áreas defeituosas podem estar presentes no rolo de material. Apesar da importância, não houve um interesse substancial na pesquisa deste problema têxtil até os últimos 15 anos. Isso pode ser explicado pela modelagem complexa que é requerida pela grande irregularidade das formas.

O empacotamento de retângulos consiste no posicionamento das formas em uma folha de matéria-prima retangular maior, de maneira que as formas não se sobreponham e que a altura total necessária seja minimizada (WHITWELL, 2004, p. 65).

O empacotamento de formas irregulares segue uma definição similar ao empacotamento de formas retangulares. As formas precisam ser posicionadas em uma área onde a largura total de matéria-prima deva ser minimizada. A principal diferença entre os dois problemas é que o teste de intersecção entre formas irregulares é consideravelmente mais complicado que o caso com apenas formas retangulares (WHITWELL, 2004, p. 127).

2.2 METAHEURÍSTICA

Metaheurísticas são métodos de solução que orquestram uma interação entre procedimentos de melhoramento local e estratégias de alto nível com o objetivo de criar um

processo capaz de escapar de uma solução local ótima e realizar uma pesquisa robusta em toda a área da solução. Ao longo do tempo, estes métodos passaram a incluir procedimentos que empregam estratégias de superar a armadilha do melhor caso local em áreas de solução complexas, especialmente os procedimentos que utilizam uma ou mais estruturas de vizinhanças como um meio de definir movimentos admissíveis entre uma solução e outra, ou para construir ou destruir soluções em processos construtivos e destrutivos (GLOVER; KOCHENBERGER, 2010, p. ix).

Um número de ferramentas e mecanismos que emergiram da criação dos métodos metaheurísticos provaram ser notavelmente eficazes, tanto que a metaheurística foi transferida para o centro das atenções nos últimos anos, como a linha preferencial de ataque para resolver muitos tipos de problemas complexos, em particular os de natureza combinatorial. Enquanto metaheurísticas não são capazes de garantir que a solução encontrada seja o melhor caso, procedimentos exatos têm provado incapacidade de encontrar soluções cuja qualidade está perto àquelas obtidas por meio de metaheurísticas — particularmente para problemas do mundo real, que normalmente atingem altos níveis de complexidade. Além disso, algumas das aplicações mais bem sucedidas que utilizam métodos exatos, incorporaram estratégias metaheurísticas (GLOVER; KOCHENBERGER, 2010, p. x).

Pesquisa é um dos assuntos centrais em sistemas de resolução de problemas. Isto acontece quando o sistema, por falta de conhecimento, fica diante da necessidade de escolha de um número de alternativas, onde cada escolha leva a necessidade de fazer novas escolhas, até o problema ser resolvido. Um exemplo inclui determinar a melhor maneira de cortar um material para fazer uma peça de roupa com o menor desperdício possível (THORNTON; BOULAY, 1998, p. 1).

Onde o número de possibilidades é pequeno, o programa pode ser capaz de fazer exaustivas análises dentre todas e então escolher a melhor. Geralmente métodos exaustivos não são viáveis, e uma decisão deve ser feita em cada ponto de decisão para examinar apenas um número limitado de alternativas mais promissoras. Embora seja fácil desenvolver um programa para resolver o problema que seja bom em manter os caminhos já percorridos e escolher os que ainda devem ser explorados, é difícil criar um programa com uma heurística que possa atravessar essa confusão de possibilidades para concentrar sua análise principal no pequeno número de escolhas críticas (THORNTON; BOULAY, 1998, p. 1).

O grau de pesquisa necessário na solução de problemas pode ser enormemente reduzido por métodos especiais e por relaxamento de algumas restrições no problema, por exemplo, encontrar uma solução aceitável ao invés da solução ótima (THORNTON;

BOULAY, 1998, p. 2).

2.3 GEOMETRIA COMPUTACIONAL

A geometria computacional é o sub-campo da teoria dos algoritmos e envolve o desenvolvimento e análise de algoritmos eficientes para problemas envolvendo o processamento de entradas e saídas geométricas. Um dos objetivos da geometria computacional é prover ferramentas de geometria básicas, para que cada área do conhecimento possa construir seus programas. Houve um grande avanço com relação a isso, mas ainda continua distante de alcançar seus objetivos (MOUNT, 2002, p. 2-3). De acordo com Chen (1996, p. 2), um grande número de áreas de aplicação, como padrão de reconhecimento, gráficos de computador, processamento de imagem, pesquisa de operações, estatísticas, entre outros, têm sido a incubadora desta disciplina desde que elas proveram problemas geométricos inerentes. Um grande número de problemas industriais envolvem facilidades de localização, corte e problemas de otimização geométrica.

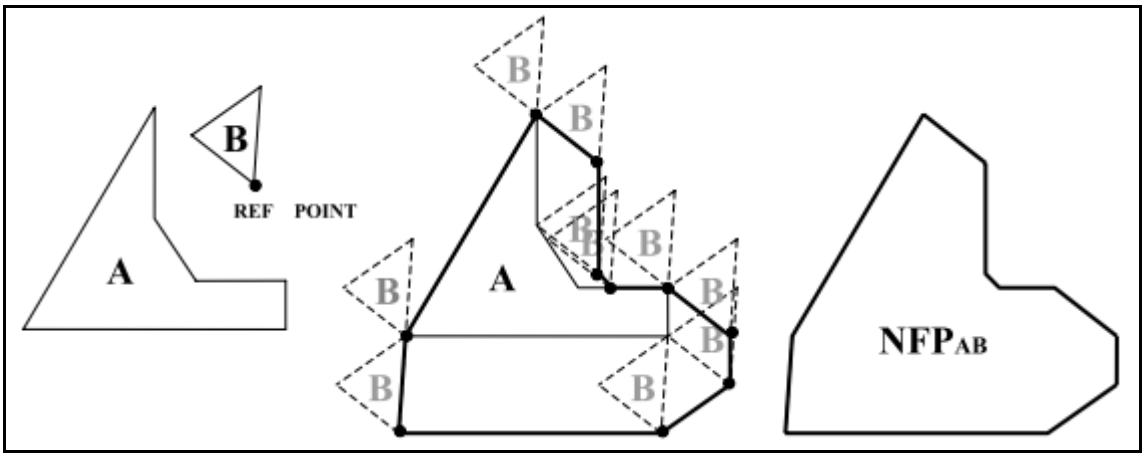
Grande parte do trabalho tem sido tentar fazer os resultados teóricos da geometria computacional acessíveis para profissionais. Isso tem sido feito simplificando algoritmos existentes, lidando com degenerescências geométricas, e produzindo bibliotecas de procedimentos geométricos (MOUNT, 2002, p. 3).

2.4 NO-FIT POLYGON

O *no-fit polygon* é uma construção que pode ser usada entre pares de formas para obter-se uma manipulação geométrica rápida e eficiente em problemas de corte e empacotamento bidimensionais. De acordo com Whitwell (2004, p. 136), anteriormente, o *no-fit polygon* não era amplamente aplicado devido à percepção de que era difícil implementá-lo por causa da inexistência de soluções genéricas, sem que houvesse necessidade de lidar caso a caso.

Dados dois polígonos, A e B, o *no-fit polygon* pode ser encontrado traçando uma

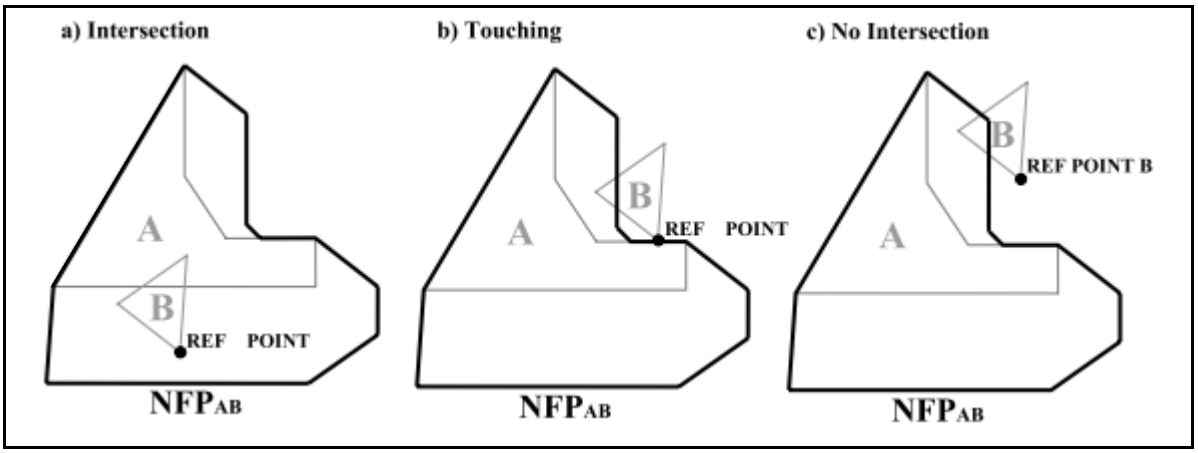
forma ao redor do outro polígono. Um dos polígonos permanece fixo em sua posição e o outro desliza ao redor das arestas do polígono fixo, sempre certificando-se que os polígonos se tocam mas nunca se sobrepõem. Para criar o *no-fit polygon*, deve ser escolhido um ponto de referência de B que será traçado conforme B vai se movimentando ao redor de A. O ponto de referência pode ser arbitrário e seguirá os movimentos do polígono móvel. É importante armazenar a posição relativa do ponto de referência, pois ele é usado no teste de intersecção conforme exibido na Figura 1 (WHITWELL, 2004, p. 135).



Fonte: Whitwell (2004, p. 135).

Figura 1 - O *no-fit polygon* de duas formas A e B

Para testar se o polígono B sobrepõe o polígono A, é usado o NFPAB e o ponto de referência de B. Se o ponto de referência do polígono B está posicionado dentro do polígono NFPAB, significa que ele sobrepõe o polígono A. Se o ponto de referência está sobre o limite do NFPAB, significa que o polígono B apenas toca o polígono A. Finalmente, se o ponto de referência está fora do NFPAB, significa que os polígonos A e B não estão sobrepostos nem se tocam. Essas três possibilidades são exibidas na Figura 2 (WHITWELL, 2004, p. 135).



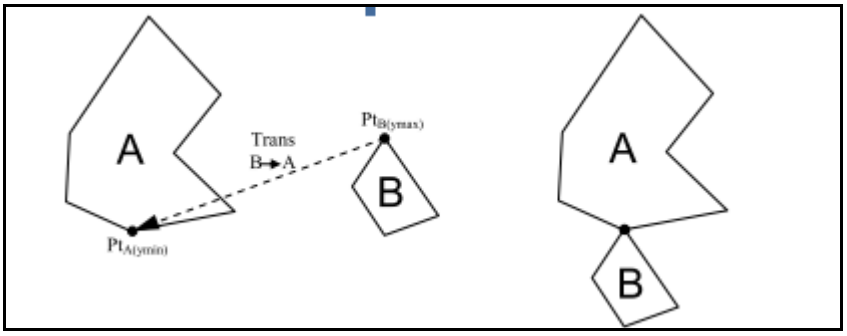
Fonte: Whitwell (2004, p. 136).

Figura 2 - Usando o *no-fit polygon* para testar a intersecção entre os polígonos A e B

2.4.1 Algoritmo de construção

O algoritmo consiste em um polígono deslizar ao redor de outro para criar um caminho externo formando o *no-fit polygon* dos dois polígonos.

Conforme descrito por Whitwell (2004, p. 181), deve-se assumir que existem dois polígonos com orientação anti-horária, A e B, que estão em algum lugar em um espaço bidimensional. A primeira operação a ser executada é transladar o polígono B a fim de que ele toque, mas não interseccione, o polígono A. O polígono B é transladado de forma que a maior coordenada y fique posicionada na menor coordenada y do polígono A, conforme é exibido na Figura 3.



Fonte: Whitwell (2004, p. 181) .

Figura 3 - Translação inicial do polígono móvel com o intuito de tocar o polígono fixo

Usando estes dois vértices como alinhamento, garante que A e B não estarão sobrepostos, mas apenas tocando-se. O resultado da translação que resulta em B tocando A é mostrado na fórmula do Quadro 1.

$$\text{Trans } B \rightarrow A = Pt_{A(ymin)} - Pt_{B(ymax)}$$

Fonte: Whitwell (2004, p. 181) .

Quadro 1 – Fórmula da translação para B tocar A

A partir deste momento, o processo de orbitação pode começar a gerar o caminho externo do *no-fit polygon* em sentido anti-horário (WHITWELL, 2004, p. 182).

2.4.2 Orbitando / Deslizando

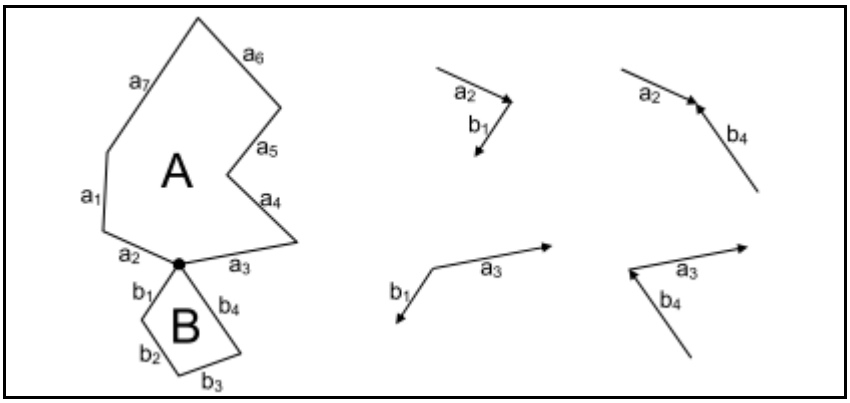
O principal objetivo da orbitação é detectar os movimentos corretos que B deve ter para percorrer ao redor de A a fim de retornar a posição original. Esse é um procedimento iterativo onde cada passo cria uma aresta do *no-fit polygon* (WHITWELL, 2004, p. 182).

Este processo pode ser quebrado em sub-partes que são descritas nesta seção uma de

cada vez. Primeiramente é descrita a detecção de arestas que se tocam. A seguir a criação de potenciais vetores de translação. Na sequência é encontrada uma translação viável. Logo após é explicado o ajuste da translação viável e por fim é aplicado o vetor de translação.

2.4.2.1 Detecção de arestas que se tocam

Segundo Whitwell (2004, p. 183), a capacidade de detectar corretamente as arestas que se tocam e intesectam é de suma importância para o sucesso do algoritmo. Isso é feito testando cada aresta do polígono A contra cada aresta o polígono B. Cada par de arestas que se tocam (uma do polígono A e uma do polígono B) é armazenado com relação a posição dos vértices em comum. A Figura 4 mostra os pares resultantes de arestas que se tocam.

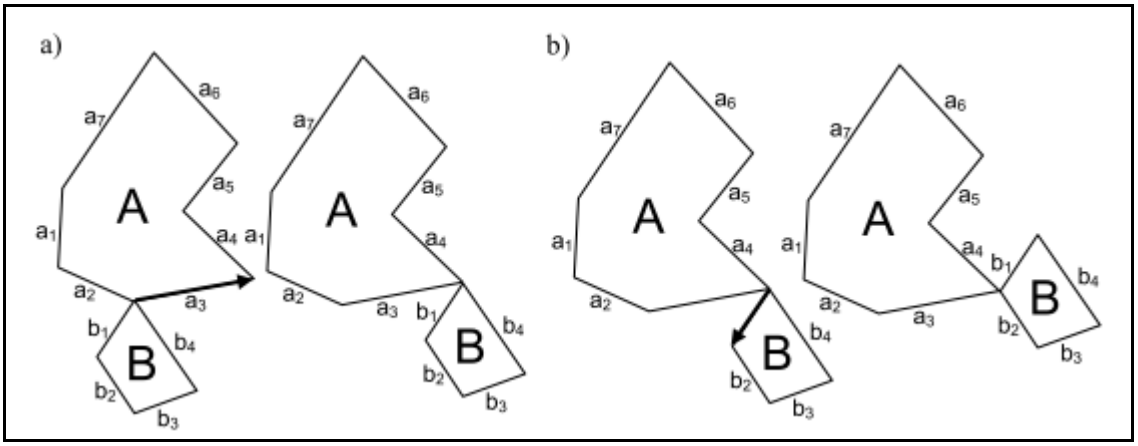


Fonte: Whitwell (2004, p. 183).

Figura 4 - Identificação dos pares de aresta que se tocam

2.4.2.2 Criação de potenciais vetores de translação

O vetor com o qual o polígono B deve ser transladado para orbitar o polígono A deve ser derivado do polígono A ou do B dependendo da situação. A Figura 5 mostra um exemplo de cada caso (WHITWELL, 2004, p. 183).

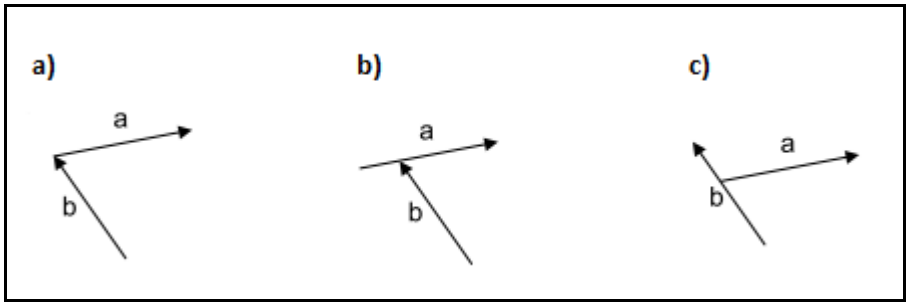


Fonte: Whitwell (2004, p. 184).

Figura 5 - Vetor de translação: a) derivado da aresta a3, b) derivado da aresta b1

A Figura 5a mostra a translação derivada da aresta fixa, onde o polígono B será movido até o fim da aresta gerando uma nova situação. Na Figura 5b, o melhor movimento a ser feito é com base a aresta polígono móvel.

Segundo Whitwell (2004, p. 184), um conjunto de potenciais vetores de translação pode ser criado usando os pares de arestas que se tocam. Existem três possibilidades. A primeira é quando ambas as arestas se tocam em um vértice conforme mostra a Figura 6a. A segunda é quando um vértice da aresta móvel toca o meio da aresta fixa conforme mostra a Figura 6b. E a última possibilidade é quando um vértice da aresta fixa toca o meio da aresta móvel, exibido na Figura 6c.



Fonte: Whitwell (2004, p. 184).

Figura 6 - Tipos de arestas que se tocam

Cada par de arestas que se tocam produz um potencial vetor de translação. É necessário identificar corretamente se ele é derivado da aresta fixa ou móvel. Isso pode ser identificado seguindo uma série de regras baseada nos vértices em comum e testando se a aresta móvel está à esquerda ou à direita da aresta fixa. O Quadro 2 mostra as diferentes possibilidades e a aresta da qual um potencial vetor é derivado em cada circunstância (WHITWELL, 2004, p. 184).

Caso	Vértices das arestas que se tocam		Posição relativa da aresta móvel/fixa	Origem do vetor de translação
	Fixo	Móvel		
1	Início	Início	Esquerda	Aresta móvel
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 móvel
7	Fim	Fim		-
8			Paralelo	Ambas as arestas

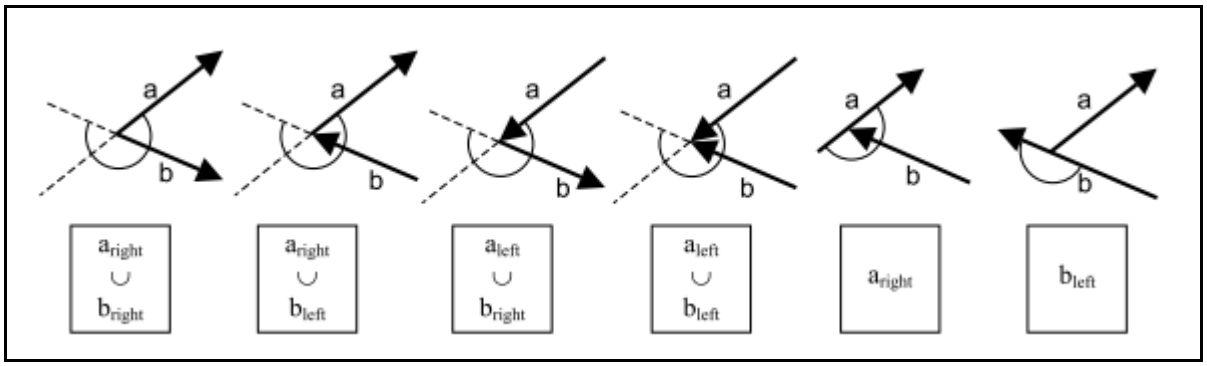
Fonte: Whitwell (2004, p. 185).

Quadro 2 – Identificando o vetor de translação

2.4.2.3 Encontrando uma translação viável

Conforme descrito por Whitwell (2004, p. 186), uma vez que os potenciais vetores de translação foram produzidos, o próximo estágio é selecionar um vetor de translação que não resulte em uma intersecção imediata. O processo consiste em pegar cada um dos potenciais vetores de translação e colocá-lo na posição em que as arestas se tocam. Isso deve ser feito para cada par de arestas. O relacionamento existente entre as arestas fixa e móvel pode ser definido com base na união das regiões da esquerda/direita, o que indica se um vetor de translação em particular será adequado para aquelas arestas (WHITWELL, 2004, p. 187). A .

Figura 7 mostra alguns exemplos de pares de arestas que se tocam e o intervalo angular das translações viáveis.



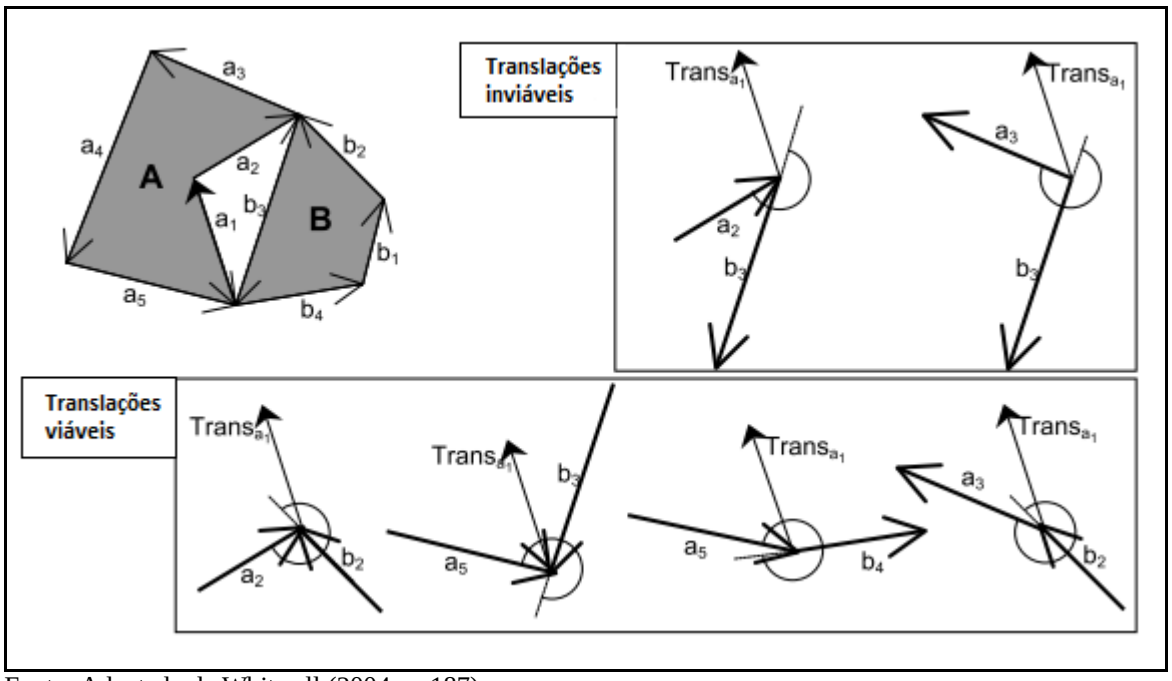
Fonte: Whitwell (2004, p. 187).

Figura 7 – Identificando o intervalo angular viável das translações (indicado pelo arco)

Existem outras possibilidades que podem ocorrer e que não estão sendo mostradas. Isso foi omitido pois os exemplos mostrados na Figura 7 provêm informação suficiente para derivar as combinações omitidas (WHITWELL, 2004, p. 187).

A Figura 8 utiliza o vetor de translação `a1` para demonstrar como este vetor é

eliminado (os pares de arestas envolvendo a aresta `a1` foram omitidos por brevidade, mas estes são viáveis pois o vetor de translação é também `a1`). Uma vez que uma potencial translação falha em algum destes testes, significa que a translação é inviável e pode ser eliminada sem mais testes.

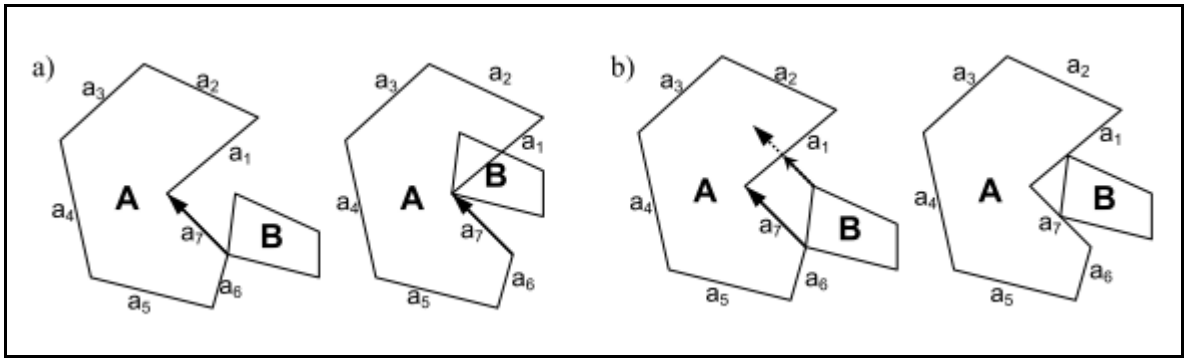


Fonte: Adaptado de Whitwell (2004, p. 187).

Figura 8 – Eliminação do potencial vetor de translação `a1`

2.4.2.4 Ajustando a translação viável

O último passo antes que o polígono B possa ser transladado é ajustar o vetor de translação. Isso é importante porque pode haver outras arestas que podem interferir na translação do polígono móvel. A Figura 9a provê um exemplo onde a aplicação do vetor de translação por completo resulta na intersecção das duas formas. A fim de prevenir que o polígono móvel entre no polígono fixo, a translação viável a7 deve ser ajustada de acordo com a aresta a1, conforme demonstrado na Figura 9b (WHITWELL, 2004, p. 189).



Fonte: Whitwell (2004, p. 189).

Figura 9 – Um vetor de translação que exige ajuste para evitar intersecção

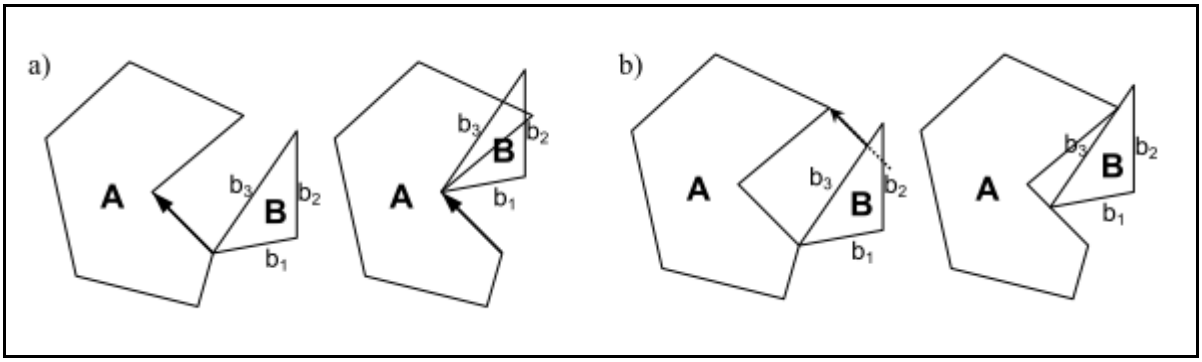
No intuito de encontrar a translação que não causa intersecção, o vetor de translação é projetado em cada um dos vértices do polígono B e é testada a intersecção com todas as arestas do polígono A (WHITWELL, 2004, p. 189). O Quadro 3 exibe a fórmula de ajuste da translação usada quando a intersecção é detectada a partir da aresta do polígono fixo.

Nova translação = InstersecçãoPt – $\text{Translação}_{\text{StartPt}}$

Fonte: Whitwell (2004, p. 189) .

Quadro 3– Fórmula de ajuste da translação

Segundo Whitwell (2004, p. 190), o vetor de translação deve ser projetado de volta a partir de cada um dos vértices do polígono A, para então ser testado se há intersecção com alguma das arestas do polígono B. Isso é retratado na Figura 10, onde é usado o mesmo exemplo, porém utilizando um polígono móvel diferente.



Fonte: Whitwell (2004, p. 190).

Figura 10 – Ajustando com projeções do polígono A

O Quadro 4 exibe a fórmula de ajuste da translação usada quando a intersecção é detectada a partir da aresta do polígono móvel.

Nova translação = $\text{Translação}_{\text{EndPt}}$ – InstersecçãoPt

Fonte: Whitwell (2004, p. 190) .

Quadro 4– Fórmula de ajuste da translação

2.4.2.5 Aplicando o vetor de translação

O último passo é realizar a translação do polígono B e adicionar o ponto de referência do mesmo ao final do *no-fit polygon* parcialmente criado. Isso fará com que o polígono mova-se para o próximo ponto de decisão e o processo possa ser reiniciado do ponto de detecção das arestas que se tocam (seção 2.4.2.1). A única verificação adicionada a este processo é a execução de um teste para detectar se o ponto de referência do polígono B retornou ao ponto inicial (WHITWELL, 2004, p. 191).

2.5 ALGORITMO BOTTOM-LEFT FILL

De acordo com Whitwell (2004, p. 149), existem várias formas diferentes de produzir soluções para o problema de corte e empacotamento bidimensional. Em geral, as soluções que atingiram os melhores resultados conhecidos usaram a técnica baseada no *no-fit polygon* para gerar potenciais posicionamentos dos polígonos e/ou testar sobreposições. O *bottom-left fill* é uma estratégia para empacotamento que utiliza uma técnica de resolução de sobreposição de formas. Esta técnica lida eficientemente com a natureza irregular das formas com a intenção de produzir soluções de alta qualidade em um curto período de tempo.

O algoritmo *bottom-lef fill* utiliza o tamanho de uma folha, a sequência de formas e as rotações permitidas. O algoritmo inicia posicionando a primeira forma no canto esquerdo inferior da folha na orientação mais eficiente. A posição mais eficiente é a rotação que produz a menor altura. As formas subseqüentes iniciam no canto esquerdo inferior da folha. Uma localização válida para o posicionamento é encontrada testando as intersecções. Se a forma não está sobreposta à outra, ela pode ser adicionada à folha. Quando uma forma está em uma posição que sobrepõe outra já adicionada, a técnica de resolução é usada para resolver a sobreposição. Se a resolução da sobreposição resulta na forma saindo do limite superior da folha, então é retornado para a parte inferior da folha e incrementado o eixo x em um certo valor (chamado de resolução). O processo continua como citado anteriormente com o teste de sobreposição e resolução das intersecções até que a forma possa ser posicionada. O processo fica completo quando todas as formas foram posicionadas na folha e a solução pode ser retornada para o usuário. As formas são sempre empacotadas na ordem em que são passadas

no parâmetro de entrada (WHITWELL, 2004, p. 167-168).

2.5.1 Técnica de resolução de intersecção

Após o desenvolvimento do *no-fit polygon*, o teste de intersecção entre pares de formas pode ser feito apenas testando se um ponto está dentro ou fora de um polígono. A sobreposição pode ser resolvida traçando uma linha vertical infinita a partir do ponto do teste e detectando a primeira intersecção com o *no-fit polygon* (WHITWELL, 2004, p. 224). A Figura 11 demonstra o cálculo da distância do ponto de referência até a aresta e a resolução da sobreposição de dois polígonos.

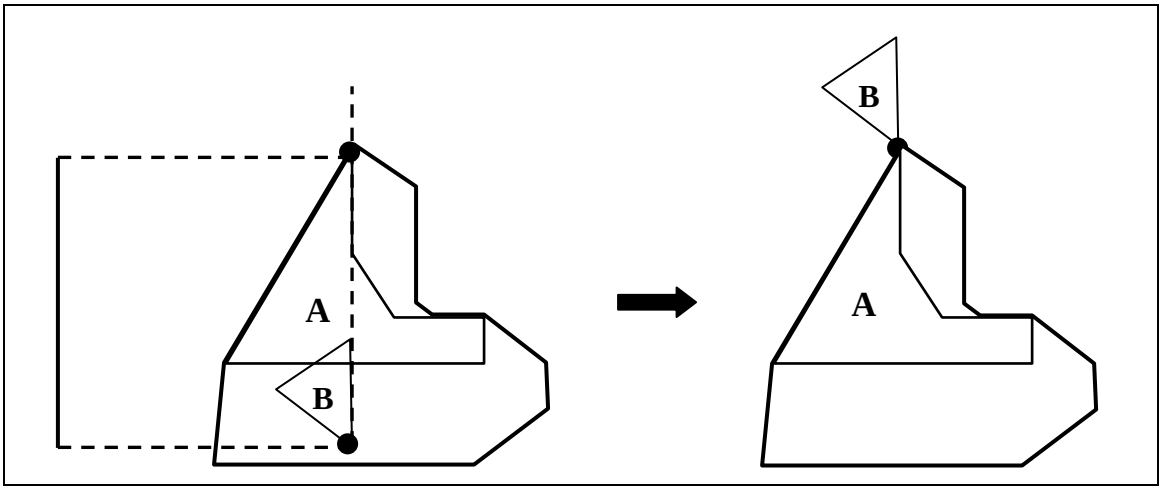


Figura 11 – Cálculo da distância do ponto de referência até a aresta

2.6 BUSCA LOCAL

Busca local é uma metaheurística que pode ser grosseiramente resumida como sendo um procedimento de busca iterativa que, inicia com uma solução viável e a melhora progressivamente aplicando uma série de modificações locais (ou movimentações). A cada iteração, a busca se move para uma nova solução viável melhorada que difere ligeiramente da solução corrente. A busca termina quando é encontrado um resultado local ótimo com relação aos resultados encontrados na busca. Isso é uma importante limitação do método, pois a menos que haja sorte extrema, o resultado local ótimo é geralmente uma solução medíocre (GENDREAU; POTVIN, 2010, p. 43).

Algoritmos de busca local não são sistemáticos, mas têm duas grandes vantagens: eles usam pouquíssima memória (normalmente apenas uma quantidade constante) e são frequentemente capazes de encontrar soluções em espaços de pesquisa grandes ou infinitos. Estas vantagens vêm, é claro, com um grande tempo de processamento. A busca local pode ser considerada como um caso especial de busca global que incorpora populações maiores. Se os requisitos previamente mencionados foram atendidos, a busca global, por sua vez, pode ser considerada como um caso especial de algoritmo de otimização global (WEISE, 2002, p.290-291).

Considerando o problema de empacotamento, é usual aplicar alguma ordenação às formas de um dado problema antes de realizar o empacotamento, geralmente decrescendo o tamanho ou área. Embora isso geralmente resulte em soluções de qualidade razoável, melhorias podem ser encontradas se um mecanismo de busca local é aplicado para gerar entradas reordenadas (WHITWELL, 2004, p. 169).

2.6.1 Algoritmo *hill climbing*

O algoritmo *hill climbing* provavelmente oferece uma das estratégias mais simples para problemas de otimização. Esse processo envolve manter uma solução corrente e gerar um resultado de uma vizinhança, o qual é então avaliado. Se o resultado é melhor que o corrente, o resultado corrente é sobrescrito pela nova vizinhança e a busca continua gerando uma solução da nova solução corrente. Se a vizinhança é de pior qualidade que a solução corrente, ela é descartada e o processo continua com a solução original. No final da busca a melhor solução encontrada é retornada (a solução corrente sempre será a melhor solução encontrada no caso do *hill climbing*) (WHITWELL, 2004, p. 49).

Conforme descrito por Whitwell (2004, p. 49), a principal desvantagem do *hill climbing* é que pode haver áreas da busca que resultem apenas em um ótimo resultado local encontrado e também não há meios de recuo de aceitação de resultados piores. Uma solução para evitar este problema é reiniciar a busca randomicamente se não tiver ocorrido uma melhora na qualidade da solução em um dado número de iterações.

Um algoritmo *hill climbing* padrão aplica operadores na solução corrente com objetivo de encontrar uma vizinhança de melhor qualidade. Se uma vizinhança melhor é encontrada, ela é adotada como a solução corrente e a busca continua. Se a vizinhança encontrada não é um melhor que a solução atual, ela é descartada e a busca continua com outras vizinhanças. A

melhor solução é retornada no final da busca (WHITWELL, 2004, p. 169).

2.6.2 Algoritmo *tabu search*

O algoritmo *tabu search* estende a idéia existente por trás do *hill climbing*. Diferentemente do *hill climbing*, o *tabu search* avalia um subconjunto de soluções geradas por vizinhanças a cada iteração da busca e caminha para a solução com o resultado de melhor valor. A solução corrente é sempre sobrescrita com a melhor vizinhança mesmo que a nova solução tenha um pior resultado. Além disso, este algoritmo mantém uma lista de tamanho fixo que contém um histórico das soluções vistas anteriormente. Essa lista de tabu é usada para evitar que sejam revisitadas áreas que já foram examinadas na história recente da busca. Se isso não fosse feito, potencialmente a busca poderia ficar presa alternando entre duas soluções (ou um ciclo de mais de duas movimentações) com os melhores resultados locais (WHITWELL, 2004, p. 50).

De acordo com Whitwell (2004, p. 169), o mecanismo de *tabu search* gera um número determinado de vizinhos e move-se para a melhor solução neste subconjunto de vizinhanças. Esta solução é então usada para gerar o próximo conjunto de vizinhos, considerando que uma lista de um dado tamanho não será revisitada.

2.6.3 Operadores

Os operadores usados em ambas as técnicas de busca, *hill climbing* e *tabu search*, são `1-Opt`, `2-Opt`, `3-Opt`, `4-Opt` e `N-Opt`. O operador `1-Opt` troca uma forma randomicamente selecionada de lugar com a primeira forma a sua direita. O operador `2-Opt` troca duas formas randomicamente selecionadas de lugar uma com a outra. Isso é estendido até o `4-Opt` onde quatro formas randomicamente selecionadas são trocadas de lugar umas com as outras. O operador `N-Opt` seleciona um número randômico de formas, susceptível a uma solução radicalmente diferente, e assim diversificando a busca. O operador é escolhido por meio de um número randômico que é então comparado com uma escala, que define qual operador deve ser usado. Cada operador tem uma chance diferente de seleção, partindo do `1-Opt`, o qual tem a maior chance de ser selecionado, até `N-Opt`, o qual tem uma chance bem menor de

ser selecionado. Isso porque os operadores menos radicais permitem a concentração na busca e os operadores mais radicais, o N-Opt, por exemplo, permitem que a busca tente encontrar a solução ótima (WHITWELL, 2004, p. 169-170).

O pseudocódigo exibido no Quadro 5 mostra como *hill climbing* e *tabu search* interagem com o *bottom-left fill*.

```
INPUT: Problem shapes, Quantities and allowable rotations, Sheet size
       current.ordering =Sort ordering(decreasing area, decreasing lenght)

Begin

Current.evaluation = Bottom-left-fill(current.ordering);
best = current;

while(!Stopping criteria) //either max number of iterations or time based
{
   opt = Select operator (1,2,3,4,n Opt);

   if(TABU)
   {
      for(i = 0; i < neighbourhood size; i++)
      {
         neighbour[i].ordering = Generate NotTabu Neighbour(current.
ordering, opt);
         neighbour[i].evaluation =Bottom-Left-Fill(neighbour[i].ordering);
      }

   }
   else if (HILL CLIMBING)
   {
      neighbour.ordering = Generate Neighbour(current.ordering, opt);
      neighbour.evaluation = Bottom-Left-Fill(neighbour.ordering);
      if (neighbour.evaluation <= current.evaluation){current =
neighbour;}
   }

   if (current.evaluation < best.evaluation) { best = current; }
}
return best;

End
```

Fonte: Withwell (2004, p.170).

Quadro 5 – Utilização do algoritmo *bottom-left fill*

2.7 TRABALHOS CORRELATOS

Existem vários documentos que descrevem o problema de corte e empacotamento e suas variantes. Ele é solucionado de diversas formas e está presente em várias áreas da

indústria do mundo todo. Foram selecionados três trabalhos que envolvem o encaixe de polígonos. Sendo eles: um estudo sobre o problema de corte e empacotamento descrito por Whitwell (2004), uma ferramenta para visualização gráfica de corte e empacotamento (MARIANO et al., 2009) e um sistema comercial de encaixe de moldes (AUDACES, 2010).

2.7.1 Estudo sobre nova heurística e metaheurística para o problema de corte e empacotamento

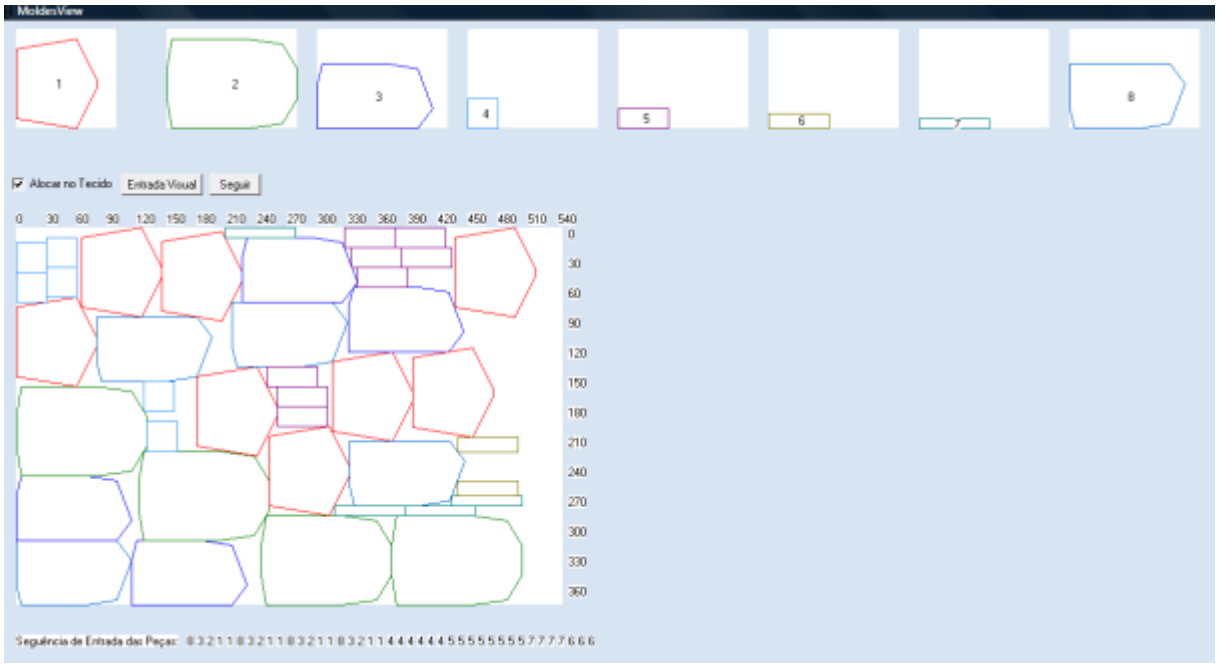
Segundo Whitwell (2004, p. 14), o objetivo de sua tese foi desenvolver e investigar novas soluções para problemas de corte de estoque bidimensionais. Três soluções para o empacotamento de retângulos e duas para o empacotamento de objetos irregulares foram desenvolvidas e apresentadas na tese, a qual produziu as melhores soluções conhecidas de todas disponíveis na literatura de corte e empacotamento dos últimos 20 anos.

O que foi produzido é capaz de empacotar os polígonos de forma rápida e precisa. Isto faz com que os algoritmos usados sejam fortes candidatos para uso em indústrias na vida real. Outra grande contribuição desta tese foi o desenvolvimento do algoritmo *no-fit polygon* que manipula arcos e buracos em polígonos.

2.7.2 Ferramenta para visualização gráfica de soluções em problemas de corte e empacotamento

MoldesView é uma ferramenta desenvolvida com o intuito de tratar o problema de corte e empacotamento de polígonos. Uma das possibilidades de uso da mesma é na área de vestuário. “Nesta ferramenta são tratadas soluções para o problema de corte e empacotamento para dimensões unidimensional e bidimensional. Durante seu desenvolvimento, levou-se em consideração teorias relacionadas à Programação Inteira e princípios de desenvolvimento em Delphi[...]” (MARIANO et al., 2009).

A ferramenta ainda permite visualizar em uma interface gráfica o resultado do encaixe dos moldes, o percentual de perda, possibilita ao usuário informar as dimensões do tecido, entre outras características. A Figura 12 exibe uma imagem da ferramenta.



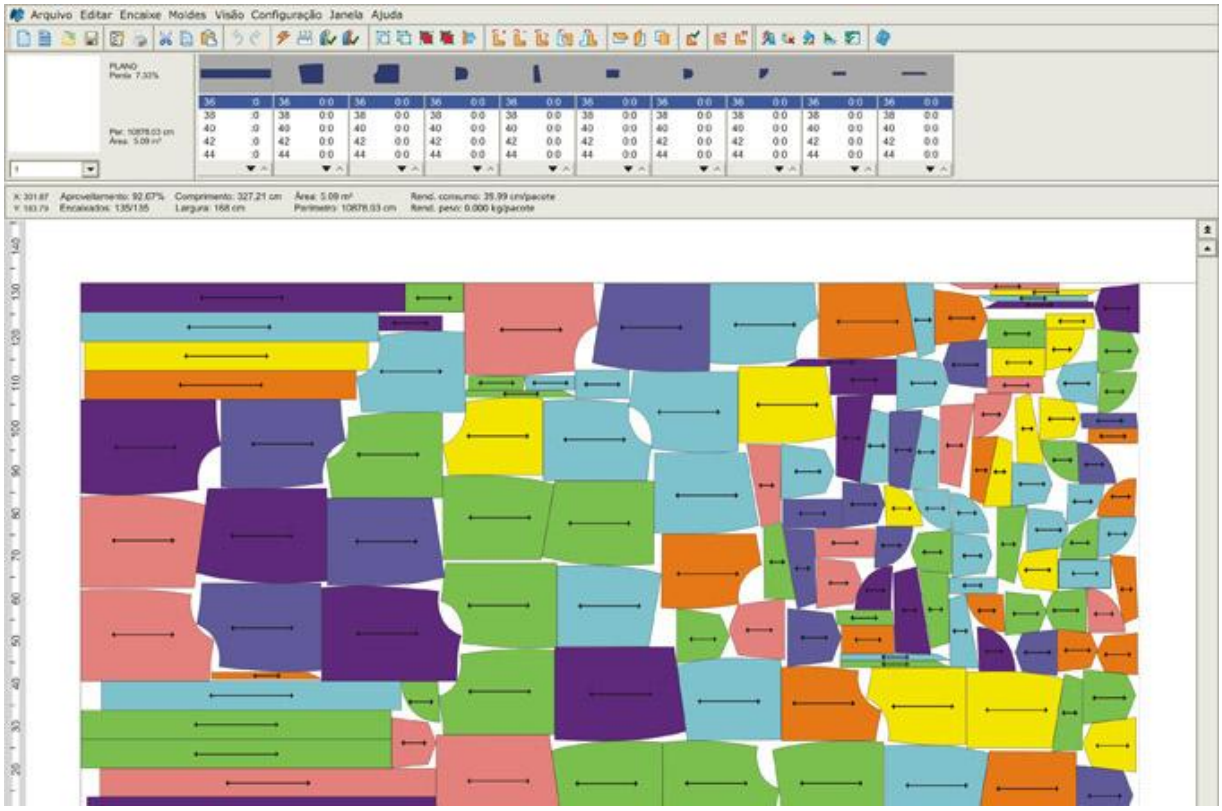
Fonte: Mariano et al. (2009).

Figura 12 – Ferramenta: MoldesView

2.7.3 Sistema comercial de encaixe de moldes

Ferramenta desenvolvida pela Audaces, empresa de Florianópolis, tem o objetivo de diminuir o consumo de matéria-prima, fazendo o encaixe de moldes com perfeição por meio de técnicas de inteligência artificial. Tem como objetivo também o melhor aproveitamento do tecido, sempre respeitando o sentido do fio e sem sobreposição de moldes. Além disso, esta ferramenta possibilita configurar o tempo de encaixe, permite usar encaixe manual junto ao especialista e diminui o custo operacional, otimizando a mão-de-obra (AUDACES, 2010). A visualização do encaixe dos moldes é atualizada durante a execução do processo de encaixe e é permitido parar o processo sem que o mesmo tenha terminado. Também é permitido informar quantos moldes devem ser usados e, caso existam falhas no tecido, pode-se delimitar a área que será inutilizada.

A Figura 13 mostra a tela da ferramenta, onde são exibidos os moldes após o processo de encaixe, a barra de ferramentas e informações referentes ao resultado do encaixe.



Fonte: Audaces (2010).

Figura 13 – Ferramenta: Vestuário Encaixe Especialista

3 DESENVOLVIMENTO

Neste capítulo são abordadas as etapas do desenvolvimento da ferramenta. São apresentados os principais requisitos, a especificação, a implementação e, por fim, são listados os resultados e discussão.

3.1 REQUISITOS PRINCIPAIS DO PROBLEMA A SER TRABALHADO

O protótipo do sistema gerador de encaixe de polígonos deverá:

a) disponibilizar uma interface visual para exibir o resultado do processo de encaixe (Requisito Funcional - RF);
b) realizar o encaixe de polígonos em uma área de largura informada pelo usuário a fim de utilizar menos espaço (RF);
c) exibir o comprimento da matéria-prima utilizada (RF);
d) exibir o tempo utilizado para efetuar o encaixe dos polígonos (RF);
e) não sobrepor os polígonos quando efetuar o encaixe (RF);
f) executar em um tempo menor que 30 minutos para um problema de dificuldade média (5 peças) (Requisito Não Funcional – RNF);
g) ser desenvolvido utilizando a linguagem de programação Java (RNF);
h) ser desenvolvido utilizando o ambiente Eclipse para a programação (RNF).

3.2 ESPECIFICAÇÃO

A especificação do presente trabalho foi desenvolvida utilizando os diagramas de casos de uso, de classes e de sequência da Unified Modeling Language (UML). Para a geração deste diagramas foi utilizada a ferramenta Enterprise Architect 7.1.833.

3.2.1 Diagrama de casos de uso

O trabalho desenvolvido possui duas formas de interação. Uma por meio da interface gráfica, onde um usuário sem conhecimentos de programação pode fazer uso da ferramenta. A outra é utilizando as classes desenvolvidas, onde quem interage é um desenvolvedor, conhecedor do desenvolvimento de programas. A Figura 14 exibe o relacionamento entre o usuário, a interface gráfica e a carga do arquivo de polígonos.

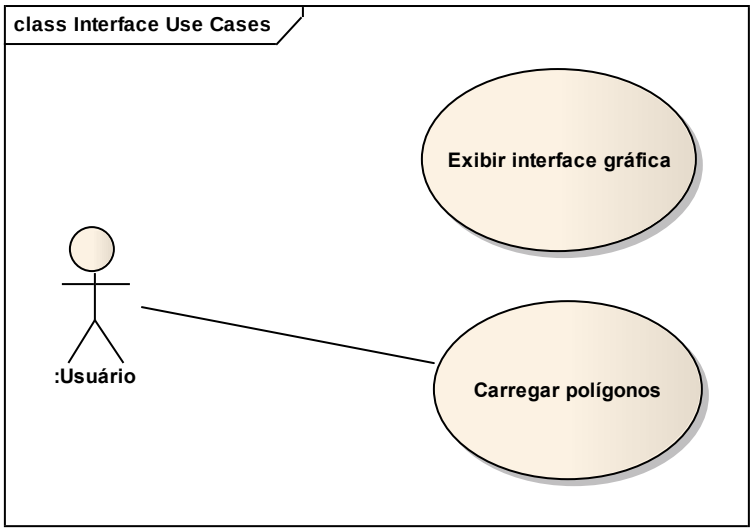


Figura 14 - Diagrama de caso de uso onde o ator é o usuário

O caso de uso `Exibir interface gráfica` descreve como o usuário interage com a interface gráfica e define os recursos gráficos que são exibidos. No Quadro 6 estão descritos alguns dos cenários possíveis.

Exibir interface gráfica: permite o usuário interagir com o programa por meio de uma tela	
Cenário principal	1. O usuário em carrega um arquivo de polígonos; 2. O usuário seleciona a busca *hill climbing*; 3. O usuário seleciona o critério de parada `Loop`; 4. O usuário digita a quantidade de iterações; 5. O usuário seleciona 1 rotação; 6. O usuário ordena a execução do empacotamento.
Fluxo alternativo 01	No passo 2, o usuário seleciona a busca *tabu search*.
Fluxo alternativo 02	No passo 3, o usuário seleciona o critério de parada `Tempo`. No passo 4, o usuário digita o tempo de execução em milissegundos.
Fluxo alternativo 03	No passo 5, o usuário seleciona 2 rotações.
Pós-condição	O sistema exibe os polígonos encaixados, o tempo da execução e a altura do empacotamento.

Quadro 6 – Caso de uso `Exibir interface gráfica`

O caso de uso `Carregar polígonos` (Quadro 7) descreve como é realizada a carga dos polígonos para o sistema por meio de arquivo eXtensible Markup Language (XML). Este

arquivo deve conter os pontos dos polígonos respeitando o sentido anti-horário, a largura e a altura da área onde o empacotamento deve ser feito.

Carregar polígonos: permite realizar a carga dos polígonos por meio de um arquivo XML	
Pré-condição	Deve existir um arquivo XML com as informações dos polígonos, altura e largura da área de empacotamento.
Cenário principal	1. O usuário seleciona um arquivo XML. 2. A ferramenta instancia a classe `XMLReader`; 3. A ferramenta chama o método `readXml` passando uma `String` contendo o caminho do arquivo.
Pós-condição	O sistema realiza a carga dos polígonos e os retorna no método `readXml`. A largura da área de empacotamento pode ser obtido por meio do método `getBorderX` e a altura pelo `getBorderY`. Os polígonos são exibidos na tela.

Quadro 7 – Caso de uso `Carregar polígonos`

A

Figura 15 demonstra o relacionamento do desenvolvedor com os casos de uso de geração do *no-fit polygon* e execução do *bottom-left fill*, *hill climbing* e *tabu search*.

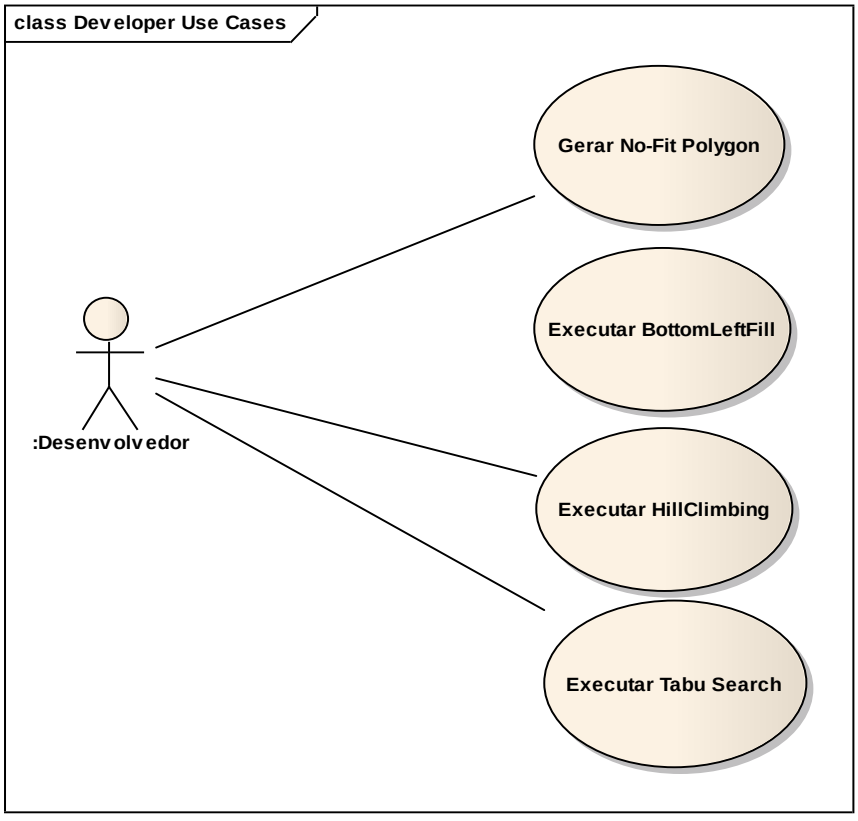


Figura 15- Diagrama de casos de uso onde o ator é o desenvolvedor

O caso de uso `Gerar No-Fit Polygon` é responsável pela geração de um novo polígono, que é resultado do percorrimento de um polígono ao redor de outro, sem que haja sobreposição, mas que sempre estão se tocando. O objetivo deste polígono resultante é a detecção de colisão entre esses dois polígonos que o geraram. No Quadro 8 está exposto o cenário deste caso de uso.

Gerar No-Fit Polygon: responsável pela geração do *no-fit polygon*	
Pré-condição	Os dois polígonos usados na geração do *no-fit polygon* devem ter sido previamente criados.
Cenário principal	1. O desenvolvedor instancia a classe `NoFitPolygon`; 2. O desenvolvedor chama o método `calculateNoFitPolygon` passando dois polígonos.
Pós-condição	O sistema retorna o polígono *no-fit polygon* gerado para os dois polígonos passados como parâmetro.

Quadro 8 – Caso de uso `Gerar No-Fit Polygon`

O caso de uso `Executar Bottom Left Fill` (Quadro 9) define o empacotamento dos polígonos considerando o número de rotações e a largura máxima definida. O objetivo é posicionar cada polígono o mais abaixo e à esquerda possível, rotacionando cada polígono de forma que o mesmo fique posicionado na menor altura possível.

Executar Bottom Left Fill: realiza o empacotamento dos polígonos	
Pré-condição	Os polígonos devem ter sido previamente criados.
Cenário principal	1. O desenvolvedor instancia a classe `BottomLeftFillAlgorithm`; 2. O desenvolvedor chama o método `doPacking` passando os polígonos a serem empacotados, o número de rotações e a largura da área de encaixe.
Pós-condição	O sistema retorna uma instância da classe `PackingResult` contendo os polígonos em suas novas posições e a altura total ocupada pelo empacotamento.

Quadro 9 – Caso de uso `Executar Bottom Left Fill`

O caso de uso `Executar Hill Climbing` (Quadro 10) descreve um método de otimização combinatorial que tem por objetivo alterar a ordenação de um conjunto de polígonos e verificar se essa nova ordenação gera um resultado melhor que os anteriormente gerados. Caso não seja gerado um resultado melhor, a combinação é descartada e o processo continua com a melhor ordenação encontrada até o momento. Este processo é executado até que uma condição de parada seja atingida, que pode ser de tempo ou quantidade de iterações. Ao final, o melhor resultado encontrado é o resultado final da execução.

Executar Hill Climbing: descreve uma estratégia de otimização combinatorial	
Pré-condição	Os polígonos devem ter sido carregados previamente.
Cenário principal	1. O desenvolvedor instancia a classe `HillClimbingAlgorithm`; 2. O desenvolvedor chama o método `doPacking` passando os polígonos, o número de rotações, a largura da área de empacotamento, o critério de parada e o valor a ser atingido para parar a execução.
Pós-condição	O sistema retorna uma instância da classe `PackingResult` contendo os polígonos em suas novas posições e a altura total ocupada pelo empacotamento.

Quadro 10 – Caso de uso `Executar Hill Climbing`

O caso de uso `Executar Tabu Search` (Quadro 11) define a utilização de um método de otimização combinatorial que estende a idéia do *hill climbing*, mas diferenciando-se no que diz respeito à geração de subconjuntos de combinações de polígonos. Dentre estes subconjuntos é obtido o melhor resultado que servirá de entrada para reiniciar o ciclo. O mesmo é finalizado ao atender a condição de parada e é retornada a melhor combinação de todas avaliadas.

Executar Tabu Search: descreve uma estratégia de otimização combinatorial	
Pré-condição	Os polígonos devem ter sido carregados previamente.
Cenário principal	1. O desenvolvedor instancia a classe `TabuSearch`; 2. O desenvolvedor chama o método `doPacking` passando os polígonos, o número de rotações, a largura da área de empacotamento, o critério de parada e o valor a ser atingido para parar a execução.
Pós-condição	O sistema retorna uma instância da classe `PackingResult` contendo os polígonos em suas novas posições e a altura total ocupada pelo empacotamento.

Quadro 11 – Caso de uso `Executar Tabu Search`

3.2.2 Diagrama de classes

A Figura 16 apresenta a especificação das classes utilizadas na implementação dos algoritmos de otimização combinatorial e de empacotamento dos polígonos. A Figura 17 mostra as classes desenvolvidas para implementar o *no-fit polygon* e a dependência da classe de empacotamento com relação a mesma. Algumas classes auxiliares foram omitidas para melhor visualização dos diagramas. As classes omitidas são de controle interno das rotinas desenvolvidas, representação gráfica dos objetos e auxiliares em cálculos matemáticos.

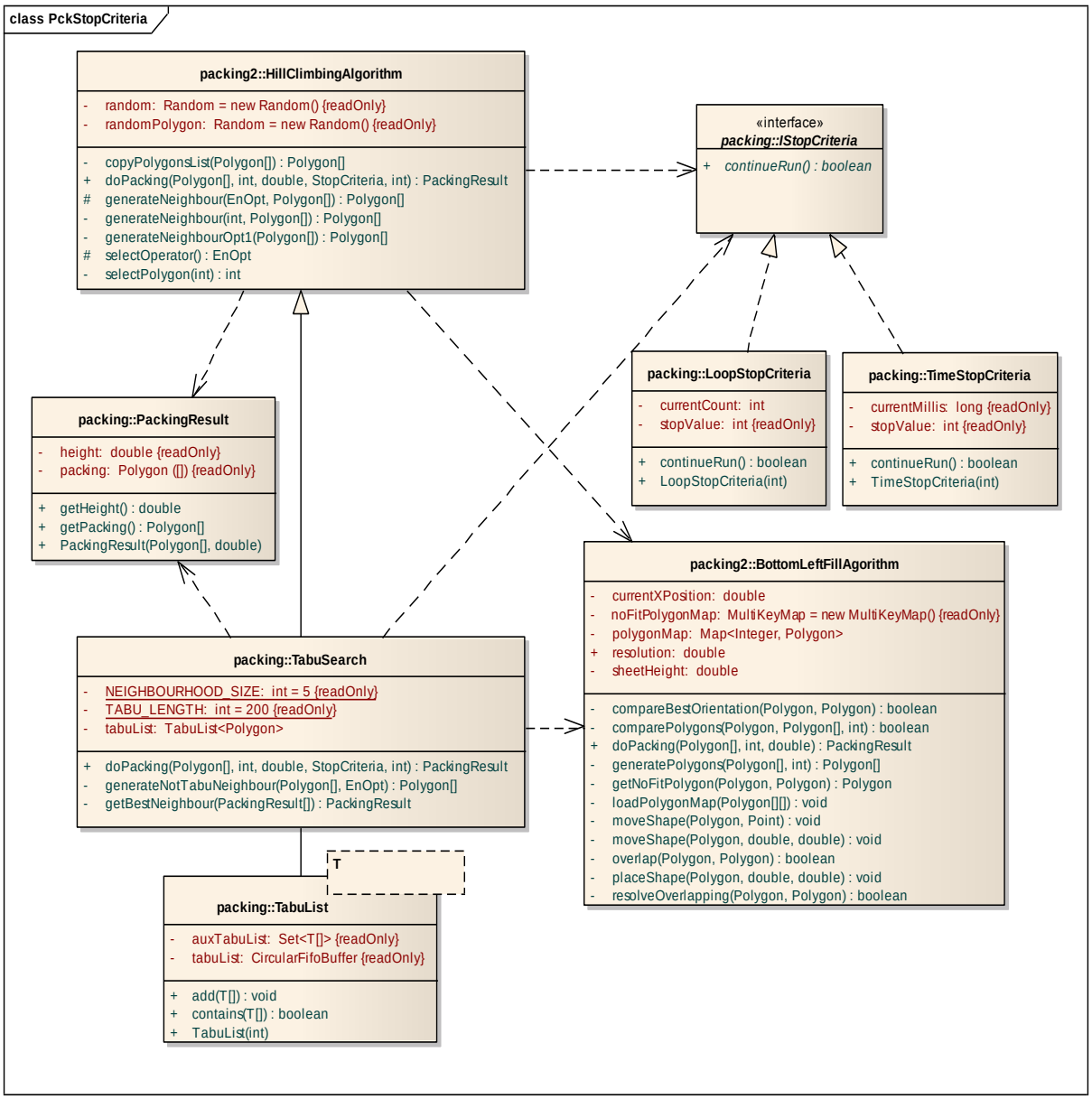


Figura 16 – Diagrama de classes usadas para busca local

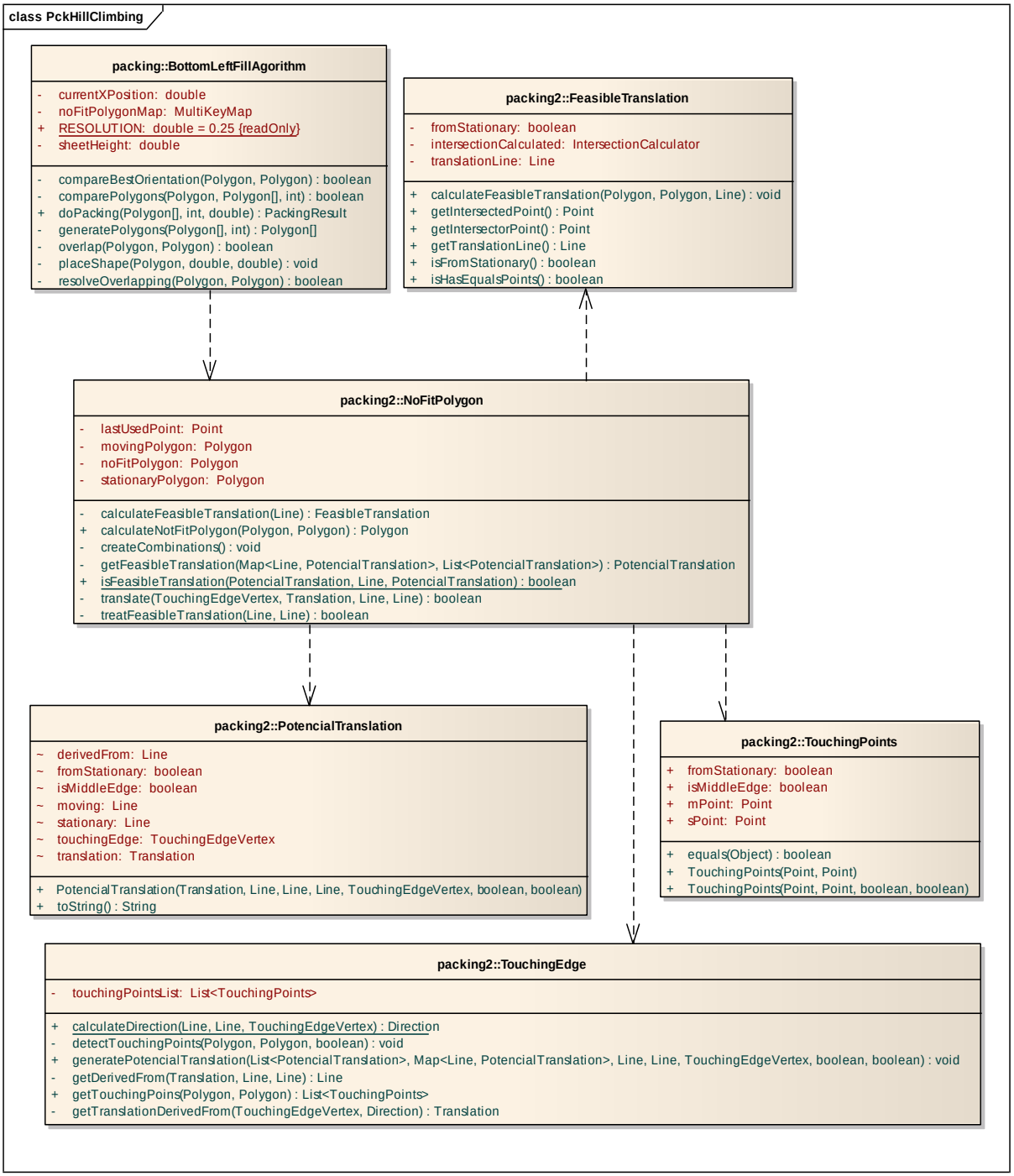


Figura 17 – Diagrama de classes da geração do *no-fit polygon* e sua utilização

3.2.2.1 Interface `IStopCriteria` e classes `LoopStopCriteria` e `TimeStopCriteria`

A interface `IStopCriteria` define o método que deve ser implementado para o critério de parada utilizado nas estratégias de otimização usadas nas classes `HillClimbingAlgorithm` e `TabuSearch`. Foram implementadas duas classes obedecendo

esta interface, que são a `LoopStopCriteria` e `TimeStopCriteria`. A primeira controla se uma determinada quantidade de iterações foi atingida, a qual foi previamente informada no construtor da classe. A cada chamada do método `continueRun` é incrementado um contador interno e o retorno deste método é a verificação do atingimento do limite de iterações. Já a segunda classe, a `TimeStopCriteria`, utiliza-se de um temporizador que, a cada chamada do método `continueRun`, verifica se o tempo é menor ao passado no construtor. Caso o tempo seja maior ou igual ao passado no construtor, significa que a execução deve ser interrompida. O temporizador é iniciado ao instanciar a classe e é calculado em milissegundos.

3.2.2.2 Classe `PackingResult`

A classe `PackingResult` é responsável por armazenar o resultado do empacotamento realizado. Ela guarda os polígonos em suas novas posições definidas pelo empacotamento e a maior altura encontrada entre os polígonos.

3.2.2.3 Classe `HillClimbingAlgorithm`

A classe `HillClimbingAlgorithm` realiza a implementação da estratégia descrita na seção 2.6.1, onde são utilizados operadores descritos na seção 2.6.3 para a geração de novas vizinhanças de polígonos. Também é utilizado um critério de parada, definido pela interface `IStopCriteria`, que conterá a implementação de um dos dois critérios de parada (tempo ou iterações). Como resultado tem-se uma instância da classe `PackingResult` com o resultado do empacotamento.

3.2.2.4 Classes `TabuSearch` e `TabuList`

A classe `TabuSearch` realiza a implementação da estratégia descrita na seção 2.6.2. Ela estende a classe `HillClimbingAlgorithm` e também utiliza operadores descritos na seção 2.6.3 para a geração de novas vizinhanças de polígonos, assim como o critério de parada definido pela interface `IStopCriteria`. A classe `TabuList` é utilizada no controle da lista

tabu, onde são armazenadas as vizinhanças já executadas. Nela é controlado se a lista de polígonos pode ou não ser executada.

Como resultado do `TabuSearch`, tem-se uma instância da classe `PackingResult` com o resultado do empacotamento.

3.2.2.5 Classe `BottomLeftFillAlgorithm`

A classe `BottomLeftFillAlgorithm` implementa a definição descrita na seção 2.5. É percorrida a lista de polígonos passados para o principal método da classe, o `doPacking`, onde os polígonos são posicionados o mais abaixo e à esquerda possível. Após isso, é verificado se existe intersecção com algum dos polígonos já posicionados. Existindo alguma sobreposição, ela é eliminada movendo o polígono conforme descrito na seção 2.5.1. Como resultado do `BottomLeftFillAlgorithm`, tem-se uma instância da classe `PackingResult` com o resultado do empacotamento.

3.2.2.6 Classe `TouchingPoints`

A classe `TouchingPoints` é uma classe de controle onde são armazenados um ponto do polígono móvel e um do polígono fixo. Esses pontos podem ser iguais ou não, dependendo da situação conforme descrito na seção 2.4.2.2, na Figura 6. Também é armazenado se as arestas se tocam em um ponto no meio da aresta e se esse ponto é originário do polígono móvel ou do fixo.

3.2.2.7 Classe `PotencialTranslation`

A classe `PotencialTranslation` armazena informações referentes às arestas que se tocam e o tipo de translação que pode ser feito entre estas arestas. Isso é descrito na seção 2.4.2.2, no Quadro 2.

3.2.2.8 Classe `TouchingEdge`

Na classe `TouchingEdge`, o método `getTouchingPoins` é responsável percorrer os pontos do polígono fixo e verificar os pontos que tocam o polígono móvel. O mesmo é feito para o polígono móvel, que tem seus pontos percorridos e comparados com os do polígono fixo. A cada ponto encontrado que toca o outro polígono, é criada uma instância da classe `TouchingPoints`, que é armazenada em uma lista.

Nesta classe existe também o método `generatePotencialTranslation`, onde é criada uma instância da classe `PotencialTranslation` que, por sua vez, é armazenada em uma lista.

E, finalmente, existe o método `calculateDirection`, que identifica se a aresta móvel está à esquerda ou à direita da aresta fixa, tendo como referência o fim da aresta.

3.2.2.9 Classe `FeasibleTranslation`

A classe FeasibleTranslation é encarregada de verificar se a translação a ser feita pode fazer com que o polígono móvel sobreponha o polígono fixo. Caso isso ocorra, essa classe ajusta a translação até o limite onde não ocorre intersecção entre os polígonos. Tal procedimento é descrito na seção 2.4.2.4.

3.2.2.10 Classe `NoFitPolygon`

A classe `NoFitPolygon` é a responsável pela geração de um novo polígono usado no testes de intersecção entre dois polígonos. Ela recebe os dois polígonos e gera um terceiro. O método `calculateNoFitPolygon` é o método principal da classe. Ele realiza a translação inicial da maior coordenada y do polígono móvel até a menor coordenada y do polígono fixo. A partir disso, o polígono móvel é deslizado ao redor do fixo até que chegue novamente ao ponto de partida, sempre armazenando o ponto de referência no novo polígono. Este procedimento é descrito da na seção 2.4.

3.2.3 Diagrama de sequência

A Figura 18 apresenta um diagrama de sequência onde é demonstrada a forma como é feita a implementação do algoritmo *hill climbing*, mostrando sua dependência com outras classes.

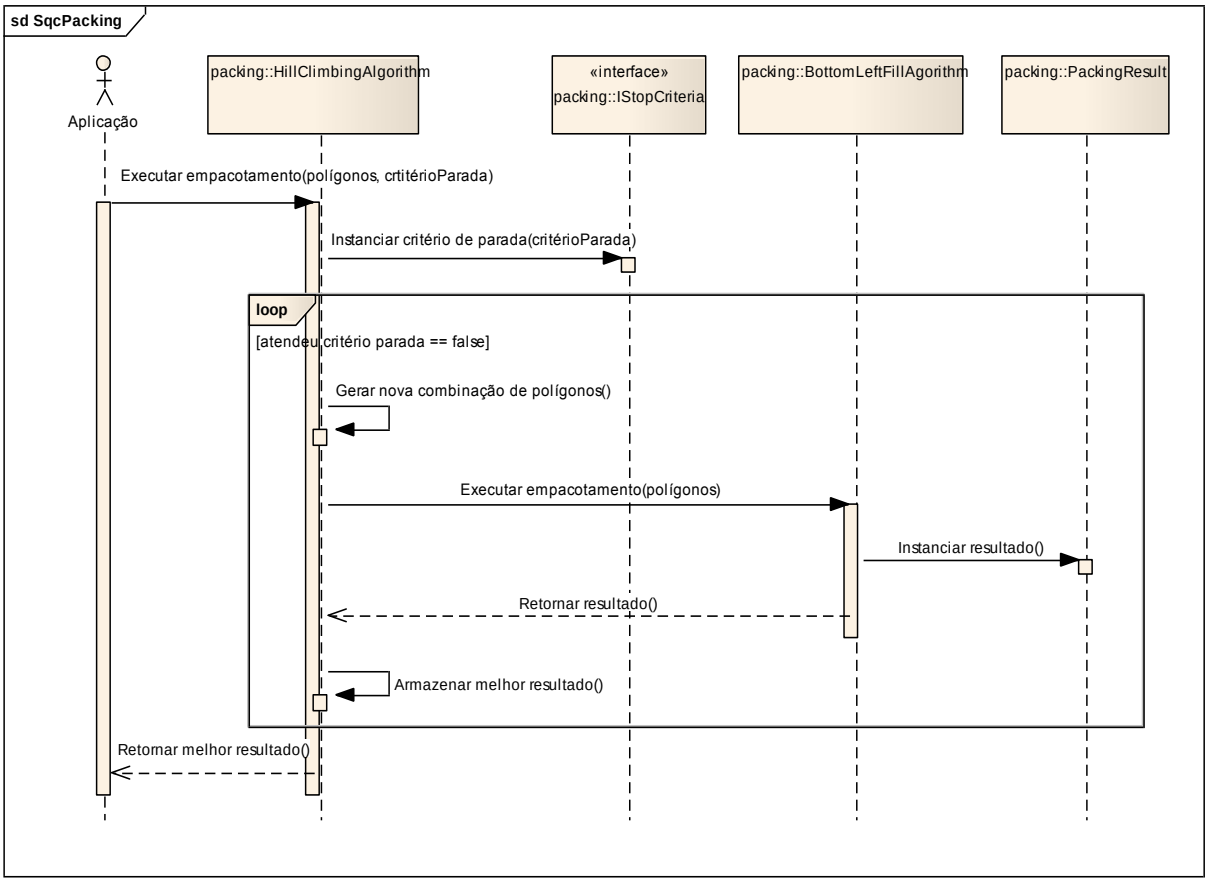


Figura 18 - Diagrama de sequência da execução do empacotamento com *hill climbing*

3.2.4 Diagrama de atividades

A Figura 19 demonstra a execução do laço principal do algoritmo *tabu search*, onde são geradas possíveis soluções ainda não exploradas e dentre elas é escolhida a melhor, que será usada para reiniciar o processo. Este processo e repete até que a condição de parada seja atendida.

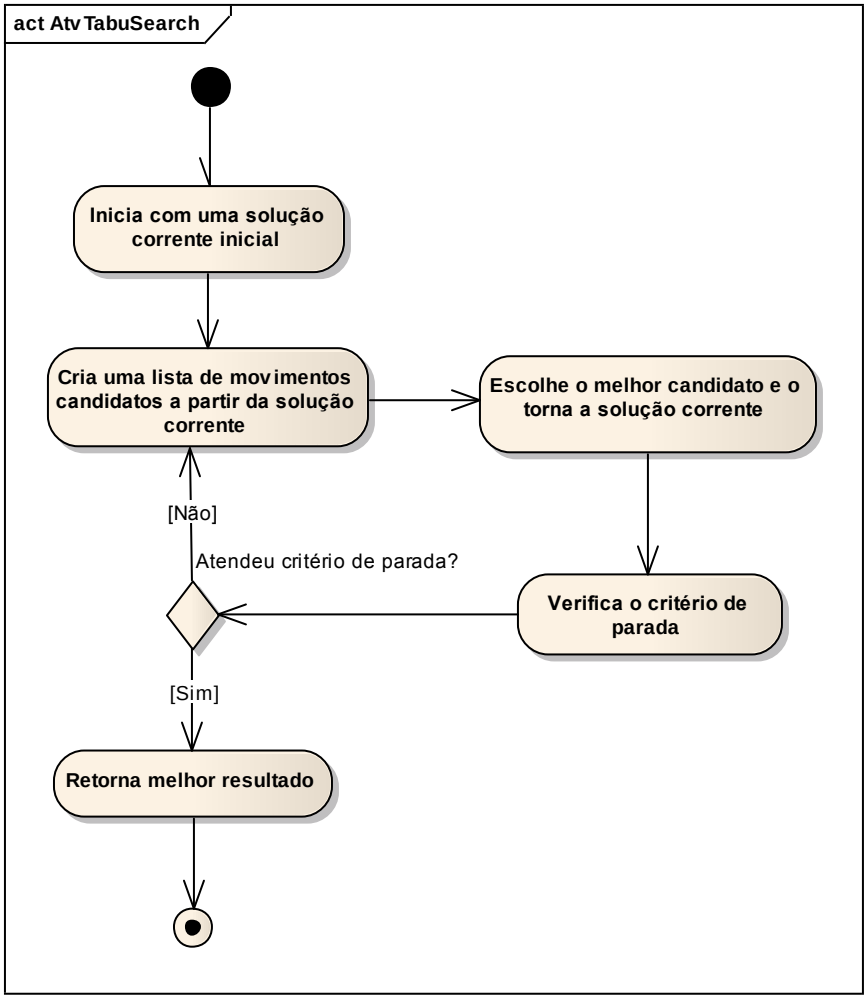


Figura 19 - Diagrama de atividades do tabu search

3.3 IMPLEMENTAÇÃO

A seguir são mostradas as técnicas e ferramentas utilizadas e a operacionalidade da implementação.

3.3.1 Técnicas e ferramentas utilizadas

Para a implementação da ferramenta foi utilizada a linguagem Java e a biblioteca gráfica OpenGL 1.1.1. O ambiente de desenvolvimento utilizado foi o Eclipse.

3.3.1.1 Implementação do *no-fit polygon*

O *no-fit polygon* é implementado pela classe `NoFitPolygon` descrita na seção 3.2.2.10. No Quadro 12 é demonstrado um trecho do código fonte onde são detectados os pontos que se tocam (linha 4). Logo após são criadas as potenciais translações a partir dos pontos que se tocam (linha 16 até 79). Em seguida, com todas as potenciais translações criadas, é detectada a translação viável (linha 81) e a mesma é usada para transladar o polígono móvel (linha 83).

```
1  private void createCombinations() {
2        TouchingEdge touchingEdge = new TouchingEdge();
3
4        List<TouchingPoints> touchingPoints =
5  touchingEdge.getTouchingPoins(stationaryPolygon, movingPolygon);
6
7        Map<Line, PotencialTranslation> derivedFromMap = new
8  HashMap<Line, PotencialTranslation>();
9
10       List<PotencialTranslation> potencialTranslations = new
11 ArrayList<PotencialTranslation>();
12
13       Line[] stationariesLines = null;
14       Line[] movingLines = null;
15
16       for (TouchingPoints touchingPt : touchingPoints) {
17             if (touchingPt.isMiddleEdge) {
18                   stationariesLines = null;
19                   movingLines = null;
20                   if (touchingPt.fromStationary) {
21                         Line stationarieLine = new
22 Line(touchingPt.sPoint, touchingPt.sPoint.next);
23                         Line movingLine0 = new
24 Line(touchingPt.mPoint.prior, touchingPt.mPoint);
25                         Line movingLine1 = new Line(touchingPt.mPoint,
26 touchingPt.mPoint.next);
27
28       touchingEdge.generatePotencialTranslation(potencialTranslations,
29 derivedFromMap, stationarieLine, movingLine0,
30 TouchingEdgeVertex.START_END, touchingPt.isMiddleEdge,
31 touchingPt.fromStationary);
32
33       touchingEdge.generatePotencialTranslation(potencialTranslations,
34 derivedFromMap, stationarieLine, movingLine1,
35 TouchingEdgeVertex.START_START, touchingPt.isMiddleEdge,
36 touchingPt.fromStationary);
37                   } else {
38                         Line stationarieLine0 = new
39 Line(touchingPt.sPoint.prior, touchingPt.sPoint);
40                         Line stationarieLine1 = new
41 Line(touchingPt.sPoint, touchingPt.sPoint.next);
42                         Line movingLine0 = new Line(touchingPt.mPoint,
43 touchingPt.mPoint.next);
44
45       touchingEdge.generatePotencialTranslation(potencialTranslations,
46 derivedFromMap, stationarieLine0, movingLine0,
47 TouchingEdgeVertex.END_START, touchingPt.isMiddleEdge,
```

```
48  touchingPt.fromStationary);
49
50        touchingEdge.generatePotencialTranslation(potencialTranslations,
51  derivedFromMap, stationarieLine1, movingLine0,
52  TouchingEdgeVertex.START_START, touchingPt.isMiddleEdge,
53  touchingPt.fromStationary);
54                  }
55              } else {
56                  stationariesLines = new Line[2];
57                  stationariesLines[0] = new
58  Line(touchingPt.sPoint.prior, touchingPt.sPoint);
59                  stationariesLines[1] = new Line(touchingPt.sPoint,
60  touchingPt.sPoint.next);
61
62                  movingLines = new Line[2];
63                  movingLines[0] = new Line(touchingPt.mPoint.prior,
64  touchingPt.mPoint);
65                  movingLines[1] = new Line(touchingPt.mPoint,
66  touchingPt.mPoint.next);
67              }
68
69              if (stationariesLines != null) {
70                  for (int k = 0; k < stationariesLines.length; k++) {
71                      for (int l = 0; l < movingLines.length; l++) {
72
73        touchingEdge.generatePotencialTranslation(potencialTranslations,
74  derivedFromMap, stationariesLines[k], movingLines[l], null, false,
75  false);
76                      }
77                  }
78              }
79          }
80
81              PotencialTranslation feasibleTranslation =
82  getFeasibleTranslation(derivedFromMap, potencialTranslations);
83              translate(feasibleTranslation.touchingEdge,
84  feasibleTranslation.translation, feasibleTranslation.stationary,
85  feasibleTranslation.moving);
86  }
```

Quadro 12 – Implementação do método `createCombinations` da classe `NoFitPolygon`

3.3.1.2 Implementação do *bottom-left fill*

O *bottom-left fill* é implementado pela classe `BottomLeftFillAlgorithm`, que é descrita na seção 3.2.2.5. No Quadro 13 é apresentado o método principal da classe `BottomLeftFillAlgorithm`. Nele é calculado atributo `resolution` (linha 4), que se refere a resolução, descrita na seção 2.5. Nesta implementação, ele é calculado com base na altura da área total pra o empacotamento. Foi utilizado 1% da altura total.

No início do método, é criado um mapa onde são armazenados os polígonos com cada uma de suas possíveis rotações (linha 14 até 17). Este mapa é usado para as outras execuções,

com o intuito de economizar tempo. O mesmo ocorre para o mapa de *no-fit polygons* usados na identificação e resolução de sobreposições.

```
 1  public PackingResult doPacking(Polygon[] polygonsList, int
 2  rotationsNumber, double sheetHeight) {
 3
 4        resolution = sheetHeight * 0.01;
 5
 6        this.sheetHeight = sheetHeight;
 7
 8        int sheetShapeIndex = 0;
 9        int bestorientation = 0;
10        double maxHeight = 0;
11
12        Map<Integer, Polygon[]> rotadedPolygonMapCopy;
13
14        if (rotadedPolygonMap == null) {
15              generatePolygons(polygonsList, rotationsNumber);
16              loadPolygonMap(rotadedPolygonMap);
17        }
18        rotadedPolygonMapCopy = cloneMap(rotadedPolygonMap);
19
20        Polygon[] sheetShapes = new Polygon[polygonsList.length];
21
22        for (int i = 0; i < polygonsList.length; i++) {
23
24              Polygon[] polygonsRotaded =
25  rotadedPolygonMapCopy.get(polygonsList[i].getId());
26              assert polygonsRotaded != null : polygonsList[i].getId();
27
28              // try all rotations configured
29              for (int j = 0; j < polygonsRotaded.length; j++) {
30                    currentXPosition = 0;
31
32                    // place shape[i][j] at bottom left of sheet;
33                    placeShape(polygonsRotaded[j], currentXPosition, 0);
34
35                    boolean overlapped = false;
36                    do {
37                          overlapped = comparePolygons(polygonsRotaded[j],
38  sheetShapes, sheetShapeIndex);
39                    } while (overlapped);
40
41                    if (compareBestOrientation(
42  polygonsRotaded[bestorientation],polygonsRotaded[j])) {
43                          bestorientation = j;
44                    }
45              }
46              // assign shape i in best orientation to sheet
47              sheetShapes[sheetShapeIndex] =
48  polygonsRotaded[bestorientation];
49              if (sheetShapes[sheetShapeIndex].maxX().x > maxHeight) {
50                    maxHeight = sheetShapes[sheetShapeIndex].maxX().x;
51              }
52              sheetShapeIndex++;
53        }
54        return new PackingResult(sheetShapes, maxHeight);
55  }
```

Quadro 13 – Implementação do método `doPacking` da classe `BottomLeftFillAlgorithm`

O que define a quantidade de rotações é o parâmetro `rotationsNumber`. As rotações são calculadas dividindo o número de rotações, recebido por parâmetro, por 360º. Caso o número de rotações seja um, significa que os polígonos permanecem em seu ângulo original. Caso seja dois, os polígonos serão encaixados em seu ângulo original e também posicionados em 180º, mas para o resultado é considerado apenas o ângulo que gerar o melhor resultado, ou seja, a coordenada x que tiver sido a menor. Caso o número de rotações seja quatro, a mesma regra é aplicada, sendo os polígonos posicionados em 90º, 180º e 270º.

3.3.1.3 Implementação do *hill climbing* e *tabu search*

O algoritmo *hill climbing* é implementado pela classe `HillClimbingAlgorithm` descrita na seção 3.2.2.3. É executado o método `doPacking` que recebe vários parâmetros, dentre eles os polígonos. Em buscas locais é comum a utilização de alguma ordenação antes de realizar o empacotamento. Nesta implementação, quanto na do *tabu search*, os polígono são ordenados por largura antes de serem executadas as buscas.

No corpo do método, inicialmente é criado o critério de parada de acordo com o critério recebido como parâmetro. É efetuado um primeiro empacotamento antes mesmo de entrar no laço. O objetivo é ter uma base de comparação para os novos resultados e executar o empacotamento com os polígonos ordenados por largura. No Quadro 14 é apresentado o código fonte do método `doPacking`.

O método `generateNeighbour` é compartilhado entre as implementações do *hill climbing* e *tabu search*. Responsável pela geração de novas combinações de vizinhanças de polígonos, este método é implementado de acordo com a seção 2.6.3. Os percentuais definidos para a implementação dos operadores `1-Opt`, `2-Opt`, `3-Opt`, `4-Opt` e `N-Opt` é de 30%, 25%, 20%, 15% e 10%, respectivamente. O código fonte da escolha do operador está no Quadro 15.

O parâmetro `randomInt` é gerado pelo método `nextInt` da classe `java.util.Random`, que retorna um número pseudo-randômico, uniformemente distribuído entre 0 (incluído) e 100 (não incluído).

O algoritmo *tabu search* é implementado pela classe `TabuSearch` conforme descrito na seção 3.2.2.4. O Quadro 16 apresenta o método principal da classe, onde primeiramente é

realizado o empacotamento com os polígonos na mesma ordem passada como parâmetro. O resultado deste empacotamento será considerado o melhor até que sejam encontrados outros.

O tamanho da lista tabu foi definido com o valor duzentos. Isso significa que as duzentas últimas combinações de polígonos não serão reavaliadas. A quantidade de vizinhanças foi definida como cinco. Quer dizer que a cada iteração serão geradas cinco novas vizinhanças, onde a melhor será usada como base para novas combinações na próxima iteração.

```
public PackingResult doPacking(Polygon[] polygonsList,int rotationsNumber,
double sheetHeight, StopCriteria stopCriteria, int stopValue) {

	IStopCriteria stopControl =
StopCriteriaControl.getStopCriteria(stopCriteria, stopValue);

	PackingResult bestResult = null;
	BottomLeftFillAgorithm bottomLeftFill = new BottomLeftFillAgorithm();

	PackingResult packingResult = bottomLeftFill.doPacking(polygonsList,
rotationsNumber, sheetHeight);
	bestResult = packingResult;
	Polygon[] bestPolygonList = polygonsList;
	Polygon[] currentPolygonList;
	for (; stopControl.continueRun();) {

		EnOpt opt = selectOperator();
		currentPolygonList = generateNeighbour(opt, bestPolygonList);
		packingResult = bottomLeftFill.doPacking(currentPolygonList,
rotationsNumber, sheetHeight);

		if (packingResult.getHeight() < bestResult.getHeight()) {
			bestResult = packingResult;
			bestPolygonList = currentPolygonList;
		}
	}
	return bestResult;
}
```

Quadro 14 – Implementação do método `doPacking` da classe `HillClimbingAlgorithm`

```
public static EnOpt getOpt(int randomInt) {
	EnOpt opt = null;
	if (randomInt < 30) {// 30%,
		opt = Opt1;
	} else if (randomInt < 55) {// 25%,
		opt = Opt2;
	} else if (randomInt < 75) {// 20%,
		opt = Opt3;
	} else if (randomInt < 90) {// 15%,
		opt = Opt4;
	} else if (randomInt < 100) {// 10%,
		opt = OptN;
	} else {
		throw new IllegalStateException("Valor incorreto: "+randomInt);
	}
	return opt;
}
```

Quadro 15 - Código fonte da escolha do operador

```
private static final int NEIGHBOURHOOD_SIZE = 5;

private static final int TABU_LENGTH = 200;

private TabuList<Polygon> tabuList;
public PackingResult doPacking(Polygon[] polygonsList,int rotationsNumber,
double sheetHeight, StopCriteria stopCriteria, int stopValue) {

	IStopCriteria stopControl =
StopCriteriaControl.getStopCriteria(stopCriteria, stopValue);

	tabuList = new TabuList<Polygon>(TABU_LENGTH);
	PackingResult[] neighbour = new PackingResult[NEIGHBOURHOOD_SIZE];
	tabuList.add(polygonsList);

	BottomLeftFillAgorithm bottomLeftFill = new BottomLeftFillAgorithm();
	PackingResult packingResult = bottomLeftFill.doPacking(polygonsList,
rotationsNumber, sheetHeight);
	PackingResult bestEvaluation = packingResult;
	PackingResult currentEvaluation = packingResult;

	for (; stopControl.continueRun();) {

		EnOpt opt = selectOperator();

		for (int j = 0; j < NEIGHBOURHOOD_SIZE; j++) {
			Polygon[] polygonsNeighbour =
generateNotTabuNeighbour(currentEvaluation.getPacking(), opt);
			PackingResult doPacking =
bottomLeftFill.doPacking(polygonsNeighbour, rotationsNumber, sheetHeight);
			neighbour[j] = doPacking;
		}
		currentEvaluation = getBestNeighbour(neighbour);

		if(currentEvaluation.getHeight() < bestEvaluation.getHeight()){
			bestEvaluation = currentEvaluation;
		}
	}

	return bestEvaluation;
}
```

Quadro 16 - Implementação do método `doPacking` da classe `TabuSearch`

A Figura 20 demonstra a dependência que os algoritmos *hill climbing* e *tabu search* têm com o algoritmo *bottom-lef fill* que, por sua vez, depende do *no-fit polygon*.

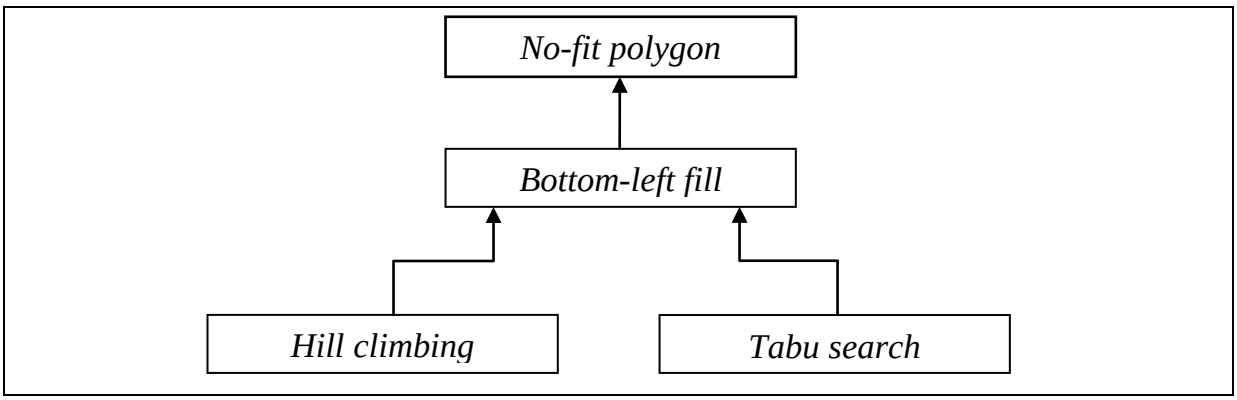


Figura 20 – Dependência entre os algoritmos utilizados no processo de empacotamento

3.3.2 Operacionalidade da implementação

Esta seção tem por objetivo mostrar a operacionalidade da implementação em nível de usuário e para tanto aborda as principais funcionalidades da ferramenta.

A ferramenta inicia sem polígonos carregados, carrega apenas a barra de botões (Figura 21).

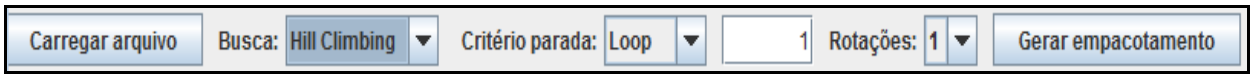


Figura 21 – Barra de botões ao carregar a ferramenta

O primeiro passo a ser feito é carregar um arquivo XML por meio do botão `Carregar arquivo`. Após a seleção do arquivo, os polígonos são carregados na parte superior da tela, onde é possível verificar os polígonos existentes no arquivo (Figura 22).

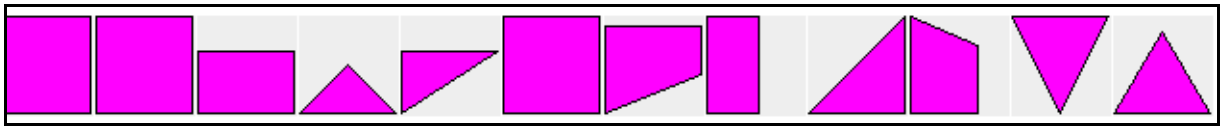


Figura 22 - Polígonos carregados do arquivo `fu.xml` obtido em Euro (2011)

O próximo passo é selecionar o algoritmo de busca, que pode ser `Hill Climbing` ou `Tabu Search`. Em seguida é selecionado o critério de parada, existindo as opções `Loop` e `Tempo`. Caso a opção de parada seja `Loop`, é necessário informar o número de iterações que serão executadas. Caso seja escolhida a opção `Tempo`, deve ser informado o tempo em milissegundos. A última configuração é o número de rotações, que pode ser `1`, `2` ou `4`. Tendo as opções escolhidas, o último passo é clicar no botão `Gerar empacotamento`, que realizará o empacotamento respeitando a configuração.

Após finalizar a execução, o empacotamento gerado é exibido no centro da tela. Também são exibidos no rodapé da tela o estado da execução, o tempo e a altura total do empacotamento (Figura 23).

Figura 23 – Informações do rodapé da tela

Na Figura 24 é exibido o resultado da execução do empacotamento do arquivo `fu.xml` obtido em Euro (2011), que possui apenas polígonos convexos, utilizando o `Tabu Search`, critério de parada `Loop` com 10 iterações e 1 rotação.

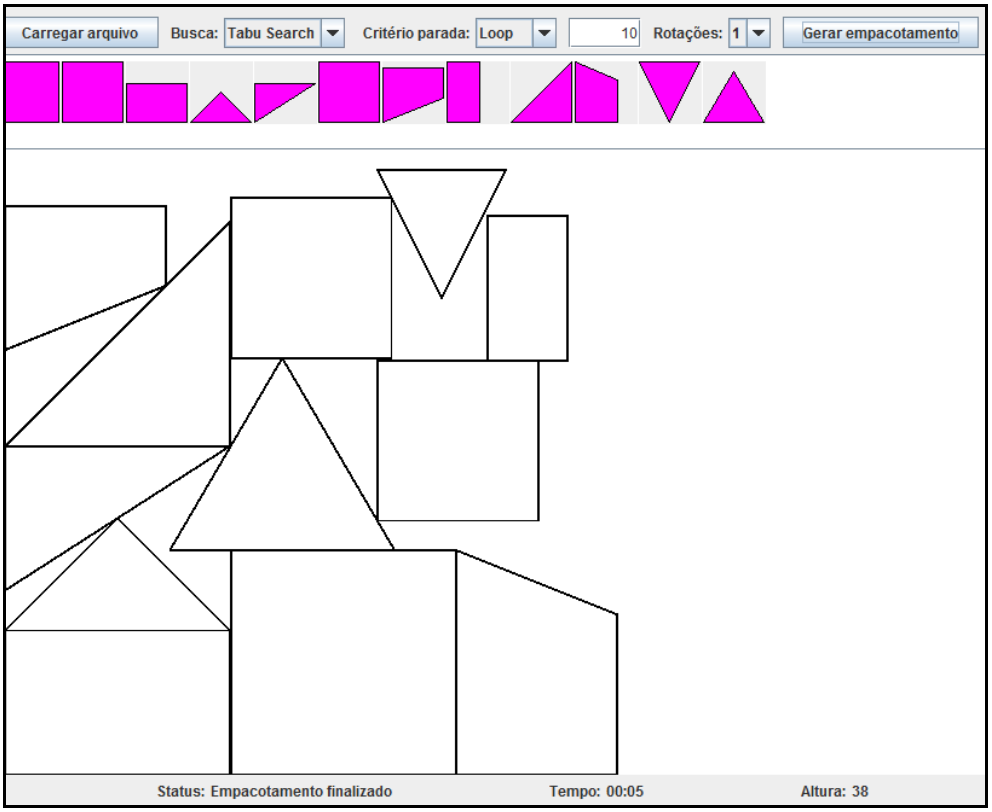


Figura 24 – Resultado da execução utilizando o arquivo `fu.xml` obtido em Euro (2011)

Na Figura 25 é exibido o empacotamento do arquivo `poly2b.xml` obtido em Euro (2011) que possui polígonos não-convexos utilizando o critério de parada o tempo de 30.000 milissegundos.

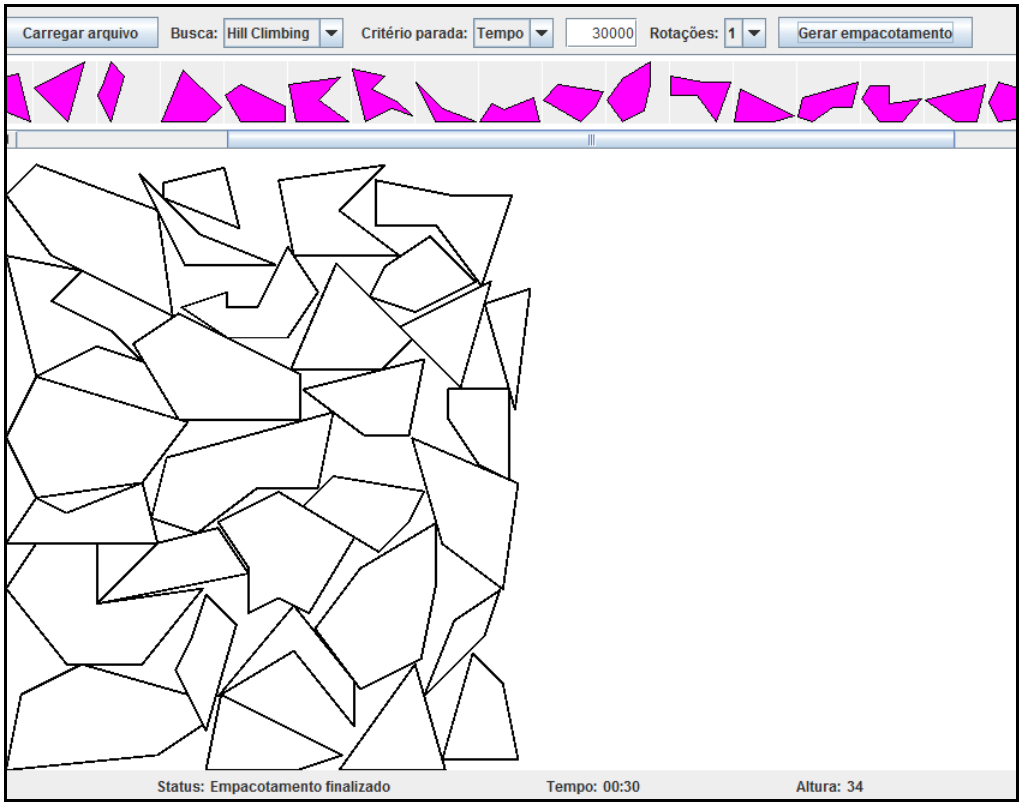


Figura 25 – Resultado da execução utilizando o arquivo `poly2b.xml` obtido em Euro (2011)

Para a leitura dos arquivos é utilizado um formato simplificado baseado no anexo A. São utilizados apenas os itens dois, onde são definidas as dimensões da área de encaixe, e o quatro, onde são definidos os polígonos a serem usados no empacotamento.

3.4 RESULTADOS E DISCUSSÃO

Para a implementação de detecção de colisão entre os polígonos foi utilizado o algoritmo *no-fit polygon*, pois, segundo Whitwell (2004, p. 136), é uma excelente ferramenta para a realização de testes de interseção entre pares de polígonos. Além disso, o uso do *no-fit polygon* pode ser muitas vezes mais rápido que a mais otimizada rotina trigonométrica. Por exemplo, em casos onde são executadas várias iterações com o mesmo problema, a pré-geração dos *no-fit polygons* pode melhorar significativamente o tempo total de computação. Ao executar várias iterações, soluções trigonométricas precisam repetidamente detectar e resolver os mesmo polígonos sobrepostos em diversas orientações e posições.

Durante a implementação, foram detectados problemas devido à representação binária de números de ponto flutuante. As coordenadas dos pontos dos polígonos são representadas por um tipo de dado de ponto flutuante, que sofre de inexatidão. Para contornar o problema, foi utilizada uma margem de erro ao verificar se pontos são iguais ou não.

Outro problema detectado durante o desenvolvimento é o de descobrimento de translações viáveis. Não foi possível testar todas as combinações existentes, considerando que pode haver vários pontos que se tocam nos polígonos fixo e móvel e que isso pode gerar várias possibilidades de translações diferentes.

Existem otimizações que podem ser realizadas a fim de reduzir o tempo de execução na geração do *no-fit polygon*. O primeira otimização é usar um algoritmo diferente para gerar o *no-fit polygon* para polígonos convexos. Segundo Whitwell (2004, p. 140), o algoritmo consiste em ter dois polígonos convexos, um com orientação anti-horária e o outro com orientação horária. Após isso, transladar todas as arestas dos dois polígonos para um único ponto. Em seguida, as arestas devem ser concatenadas em sentido anti-horário para gerar o *no-fit polygon*. Esta técnica é mais simples e com menos testes e verificações.

O segundo ponto de otimização é no teste de intersecção das arestas descrito na seção 2.4.2.4. Poderia ser implementado um teste com *bounding boxes*, o que evitaria que fosse

realizado o teste de intersecção com todas as arestas e eliminaria rapidamente processamento computacional desnecessário.

Outra otimização que pode ser feita é na execução do algoritmo de *bottom-left fill* descrito na seção 2.5. Caso uma cópia de um polígono tenha sido anteriormente adicionada à área de empacotamento, a nova cópia deste polígono pode iniciar a partir do ponto de onde a cópia anterior foi colocada. Isso economizaria processamento.

Para os testes, foi utilizado um computador com processador AMD Turion(tm) 64 Mobile Technology MK-38 (2,2 GHz) e 2 GB RAM. Foram focados os critérios de performance e qualidade do resultado obtido.

Os testes são divididos em três cenários. São analisados o tempo e a altura do empacotamento. O tempo representa o tempo total de processamento do empacotamento. Ele é mostrado em milissegundos. Já a altura representa a maior coordenada x entre todos os polígonos. Considera-se o melhor resultado aquele que possuir a menor altura, uma vez que o objetivo é diminuir a quantidade de matéria-prima a ser utilizada.

O primeiro cenário utiliza os polígonos convexos do arquivo fu.xml obtido em Euro (2011). São doze polígonos ao todo. Para este cenário, o empacotamento é realizado para os tipos de busca hill climbing e tabu search. São utilizadas 1, 25, 50, 75 e 100 iterações nas medições. Nas tabelas Tabela 1 e Tabela 2 são mostrados os resultados das medições deste cenário usando as buscas *hill climbing* e *tabu search* respectivamente. São realizadas cinco execuções para cada valor de iterações e após isso é realizada a média dos tempos em milissegundos e das alturas finais. No apêndice A é exibido uma imagem do resultado do empacotamento.

Tabela 1 – Execução do cenário um usando a busca *hill climbing*

EXECUÇÃO DO CENÁRIO UM USANDO A BUSCA HILL CLIMBING												
Itera-ções	**T1**	**T2**	**T3**	**T4**	**T5**	**Tempo(T) médio (milisse-gundos)**	**A1**	**A2**	**A3**	**A4**	**A5**	**Altura(A) média**
1	842	609	609	702	686	689,6	42,68	38,12	38,12	41,16	42,68	40,55
25	3291	3073	2918	3120	2917	3063,8	38,12	37,36	38,12	38,12	36,54	37,65
50	4292	4727	5007	4808	4711	4709,0	38,12	37,78	37,68	37,68	38,32	37,91
75	7270	7176	7270	6849	6802	7073,4	38,32	37,36	38,32	37,94	38,32	38,05
100	9095	9001	8970	8986	8658	8942,0	37,36	37,74	36,54	36,16	37,94	37,14

Tabela 2 – Execução do cenário um usando a busca *tabu search*

EXECUÇÃO DO CENÁRIO UM USANDO A BUSCA TABU SEARCH												
Itera-ções	**T1**	**T2**	**T3**	**T4**	**T5**	**Tempo(T) médio (milisse-gundos)**	**A1**	**A2**	**A3**	**A4**	**A5**	**Altura(A) média**
1	1107	1216	1217	1201	1326	1213,4	38,12	38,12	38,12	39,64	40,78	38,95
25	11232	12199	12215	11325	11809	11756,0	38,12	38,12	37,36	36,16	38,32	37,61
50	21700	22776	23213	22371	23244	22660,8	38,12	38,12	36,60	34,70	36,42	36,80
75	35225	34288	31918	33415	32011	33371,4	37,56	34,52	37,56	37,36	36,80	36,76
100	42947	43353	46831	42323	43618	43814,4	36,42	35,08	35,78	37,74	37,18	36,44

A Figura 26 exibe o gráfico dos tempos obtidos pelo empacotamento utilizando *hill climbing* e *tabu search.* É possível observar que o empacotamento realizado com o *tabu search* ficou em torno de cinco vezes mais demorado que o *hill climbing.* Isso se deve à estratégia utilizada pelo *tabu search* que, a cada iteração, cria cinco novas vizinhanças e executa o empacotamento de cada uma delas a fim de descobrir a que possui melhor resultado. Ambos os tempos crescem linearmente conforme aumenta a quantidade de iterações.

A Figura 27 apresenta o gráfico das alturas obtidas pelo empacotamento utilizando *hill climbing* e *tabu search.* Observa-se que o *tabu search* obteve melhores resultados que o *hill climbing* em todos os casos testados.

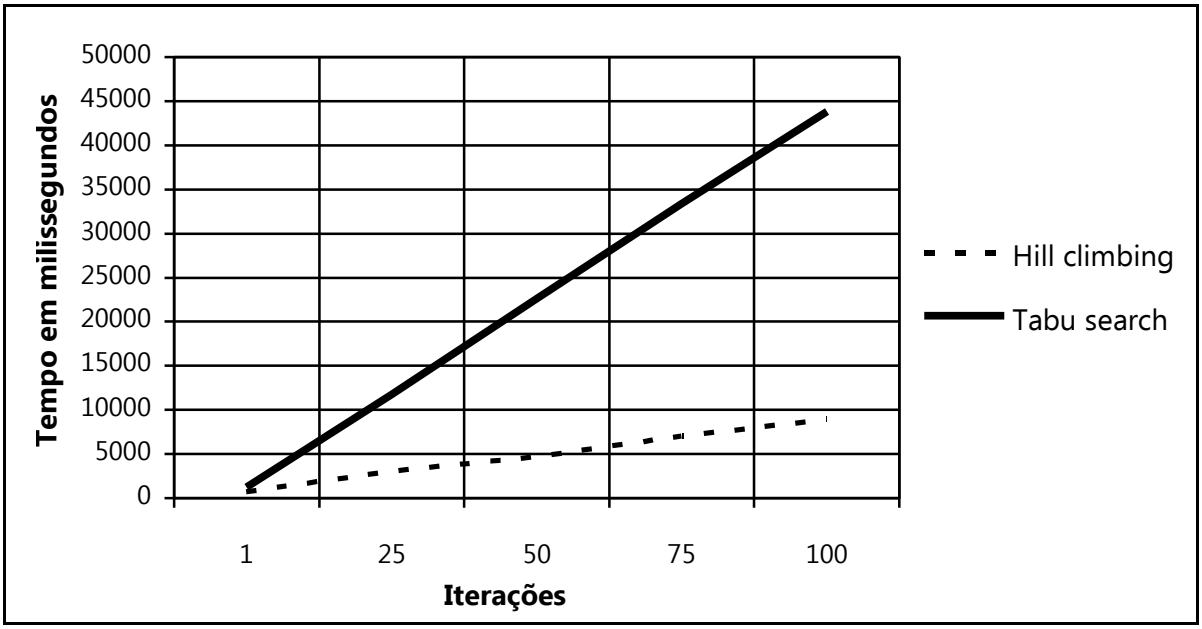


Figura 26 – Gráfico com o tempo de execução do empacotamento usando *hill climbing* e *tabu search*

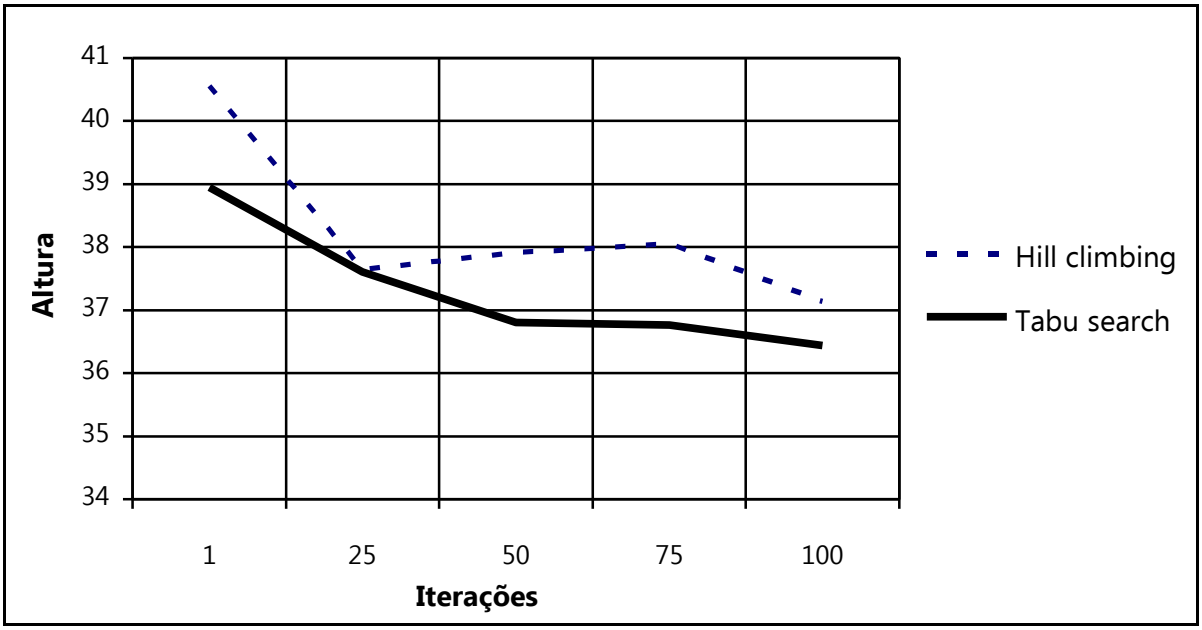


Figura 27 – Gráfico com a altura resultante do empacotamento usando *hill climbing* e *tabu search*

O segundo cenário utiliza polígonos convexos e não-convexos do arquivo `poly3b.xml` obtido em Euro (2011). Ao total são quarenta e cinco polígonos. Neste cenário o empacotamento é realizado utilizando o tempo como o critério de parada. Os tempos utilizados foram 20.000, 40.000, 60.000, 80.000 e 100.000 milissegundos. É avaliada a altura do empacotamento e são comparados os resultados do *hill climbing* com os do *tabu search*. No apêndice A é exibido uma imagem do resultado de um empacotamento. O Quadro 17 mostra os resultados obtidos.

Milisse-gundos	*Hill climbing*						*Tabu search*					
	Alturas					Média	Alturas					Média
20000	47,0	46,6	46,6	45,2	46,6	**46,40**	46,2	45,0	46,2	46,0	45,8	**45,84**
40000	45,4	46,2	46,2	46,2	45,8	**45,96**	45,4	45,0	46,0	46,2	45,8	**45,68**
60000	46,2	45,8	46,2	46,2	45,4	**45,96**	46,2	45,4	46,6	45,4	45,8	**45,88**
80000	46,2	46,2	45,4	45,6	45,8	**45,84**	46,6	45,8	45,6	45,0	45,4	**45,68**
100000	45,4	45,0	44,6	45,0	45,4	**45,08**	45,4	45,8	45,4	45,4	45,0	**45,40**

Quadro 17 – Execução do cenário dois

A Figura 28 exibe o gráfico gerado a partir dos resultados das medições do cenário dois. Observa-se que na maioria dos casos testados o *tabu search* teve melhor resultado que o *hill climbing*, exceto pelo último caso testado. Isso pode ser explicado pela natureza instável da estratégia utilizada pela busca local, que pode acabar sofrendo variações conforme as novas vizinhanças que são geradas, conforme descrito na seção 2.6.

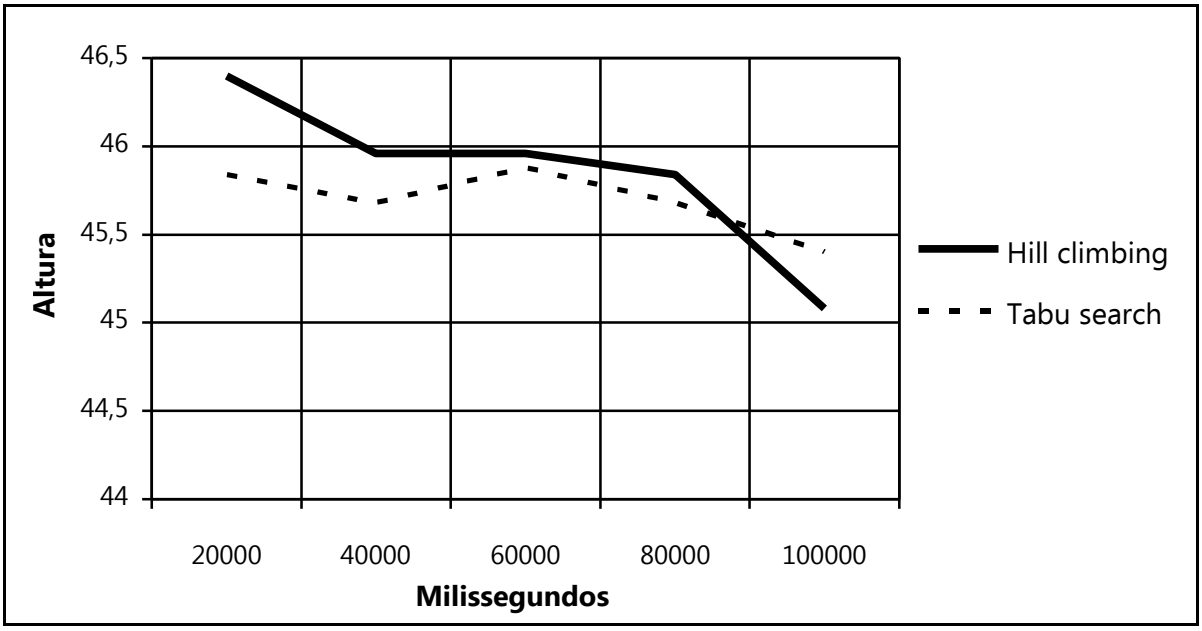


Figura 28 - Gráfico com a altura resultante do empacotamento usando *hill climbing* e *tabu search*

O terceiro cenário utiliza cinco arquivos diferentes com 10, 20, 30, 40 e 50 polígonos, respectivamente. Os polígonos utilizados nestes arquivos foram obtidos do arquivo fu.xml. Para a realização do empacotamento, foi utilizada a busca *hill climbing* e apenas uma iteração. Na tabela Tabela 3 são mostrados os resultados das medições deste cenário. São realizadas cinco execuções para cada arquivo e após isso é realizada a média dos tempos em milissegundos.

Na Figura 29 é exibido o gráfico gerado a partir das médias dos tempos de execução dos arquivos do cenário três. O tempo total de execução de uma iteração tem complexidade assimptótica $O(N^2)$, pois o algoritmo de geração do *no-fit polygon* é executado para cada par de polígonos.

Tabela 3 – Execução do cenário três usando a busca *hill climbing*

EXECUÇÃO DO CENÁRIO TRÊS USANDO A BUSCA HILL CLIMBING						
Polígonos	**T1**	**T2**	**T3**	**T4**	**T5**	**Tempo(T) médio (milissegundos)**
10	577	578	624	593	594	593,2
20	1654	1834	1700	1747	1731	1733,2
30	3105	3026	3058	2980	3104	3054,6
40	5538	5569	5632	5569	5601	5581,8
50	8877	8955	8845	9172	9626	9095,0

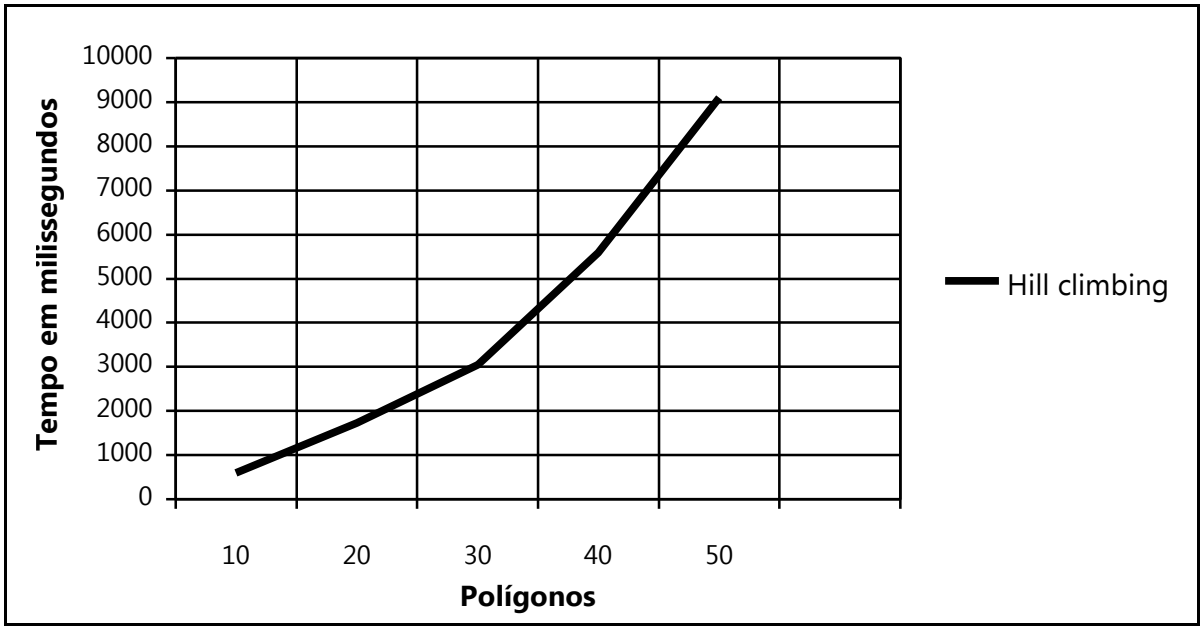


Figura 29 - Gráfico com o tempo médio do empacotamento usando *hill climbing*

4 CONCLUSÕES

O presente trabalho possibilitou desenvolver uma ferramenta capaz de realizar o encaixe de polígonos tanto convexos quanto côncavos em uma área pré-definida. O algoritmo *no-fit polygon* foi de extrema importância para a detecção e resolução de colisões entre os polígonos. No entanto, nem todas as possibilidades existentes foram testadas, estando limitado aos conjuntos de dados usados nos testes.

Os algoritmos de busca local *hill climbing* e *tabu search* se mostraram efetivos e satisfatórios para a geração de novas combinações utilizadas no empacotamento. Para a geração da solução inicial utilizada em comparações posteriores, foram utilizados sempre os polígonos ordenados por largura. Para a parada dos algoritmos de busca, são utilizados critérios de tempo e de iterações, que são configuradas pelo usuário antes de cada execução.

A utilização do *bottom-left fill* foi significativa para o resultado do trabalho através da resolução de sobreposição de polígonos no eixo vertical. Isso permitiu gerar empacotamentos densos, onde os polígonos ficam próximos uns ao outros, sempre com o objetivo de minimizar a área ocupada.

Para o desenvolvimento da parte matemática, que envolve translação e rotação, e parte da área gráfica, foram reaproveitadas trabalhos realizados anteriormente, desenvolvidos pela autora deste trabalho durante o curso de graduação. Isso se mostrou importante devido ao adiantamento do trabalho nesta parte.

A ferramenta implementada atendeu seu propósito, pois ela é capaz de realizar o carregamento de arquivos e gerar o empacotamento em tempo hábil, além de gerar um bom resultado, atendendo assim o seu objetivo principal. O resultado do empacotamento é mostrado ao final da execução, sendo possível ver a posição de cada polígono na área de empacotamento.

A linguagem de programação Java não demonstrou ser um entrave para o desenvolvimento de uma ferramenta que envolve cálculos matemáticos, pois atendeu o requisito de desempenho e se fez prática no desenvolvimento da solução.

4.1 EXTENSÕES

Sugerem-se as seguintes extensões para a continuidade do trabalho:

a) otimização do algoritmo *no-fit polygon* em dois pontos. O primeiro consiste na criação de um algoritmo para polígonos convexos, que é um algoritmo mais simples e rápido. O segundo ponto é a utilização de *bounding boxes* para testar intersecções no ajuste da translação viável;
b) otimização do algoritmo *bottom-left fill*, utilizando o última posição de um polígono já empacotado para iniciar o encaixe de uma cópia desse polígono;
c) suportar arcos e buracos nos polígonos;
d) permitir que sejam configuradas rotações individuais para os polígonos e exibir estas informações na tela;
e) exibir, durante a execução do processo de empacotamento, os resultados intermediários (soluções descartadas durante a execução);
f) permitir que o usuário ajuste manualmente o resultado, caso desejar;
g) apresentar o percentual de ocupação da área de empacotamento;
h) preencher os polígonos com uma cor para facilitar a visualização do resultado do empacotamento.

REFERÊNCIAS BIBLIOGRÁFICAS

AUDACES. **Melhor aproveitamento de tecido com o** V**estuário Encaixe Especialista**. Florianópolis, 2010. Disponível em: <www.audaces.com/novo/pt/produtos/vestuario_encaixe_espe.php#abas>. Acesso em: 7 set. 2010.

CHEN, Jianer. **Computational geometry**: methods and applications. Texas: A&M University, 1996. Disponível em: <faculty.cs.tamu.edu/chen/notes/geo.pdf>. Acesso em: 13 set. 2010.

EURO Special Interest Group on Cutting and Packing. **Data sets 2D irregular**. Portugal, 2011. Disponível em: <http://paginas.fe.up.pt/~esicup/tiki-list_file_gallery.php?galleryId=2>. Acesso em: 22 maio 2011.

GENDREAU, Michel; POTVIN, Jean-Yves. Tabu search. In: GENDREAU, Michel; POTVIN, Jean-Yves. **Handbook of metaheuristics**. 2nd. ed. New York: Springer, 2010. p. 41-56.

GLOVER, Fred; KOCHENBERGER, Gary A. Preface to first edition. In: GENDREAU, Michel; POTVIN, Jean-Yves. **Handbook of metaheuristics**. 2nd. ed. New York: Springer, 2010. p. ix-xi.

GOLDBARG, Marco C.; LUNA, Henrique P. L. **Otimização combinatória e programação linear**: modelos e algoritmos. Rio de Janeiro: Campus, 2000.

MARIANO, Rodrigo C. G. et al. MoldesView: uma ferramenta para visualização gráfica de soluções em problemas de corte e empacotamento. In: MOSTRA ACADÊMICA E CIENTÍFICA, 2., 2009, Viçosa. **Anais eletrônicos...** Viçosa: Faculdade de Viçosa, 2009. Não paginado. Disponível em: <correio.fdvmg.edu.br/downloads/MostraAcad2009/MoldesView_UmaFferramentaVisualiza%E7%E3oGr%E1fica.pdf>. Acesso em: 13 set. 2010.

MOUNT, David M. **Computational geometry.** Maryland, 2002. Disponível em: <http://www.cs.umd.edu/~mount/754/Lects/754lects.pdf>. Acesso em: 13 set. 2010.

THORNTON, Christopher; BOULAY, Benedict. **Artificial intelligence**: strategies, applications, and models through search. 2nd ed. New York: Amacom, 1998. Disponível em: <preterhuman.net/texts/science_and_technology/artificial_intelligence/Artificial%20Intelligence%20Strategies,%20Applications,%20and%20Models%20Through%20Search%20%202d%20ed%20-%20Christopher%20Thornton.pdf>. Acesso em: 13 set. 2010.

WHITWELL, Glenn. **Novel heuristic and metaheuristic approaches to cutting and packing**. 2004. 314 f. Thesis (Doctor of Philosophy) - School of Computer Science and Information Technology, University of Nottingham, Nottingham. Disponível em: <www.cs.nott.ac.uk/~gxk/papers/gxwPhDthesis.pdf>. Acesso em: 13 set. 2010.

WEISE, Thomas. **Global optimization algorithms:** theory and application. 2nd. ed. USA, 2002. Disponível em: < http://www.it-weise.de/projects/book.pdf>. Acesso em: 22 maio 2011.

APÊNDICE A – Execução dos cenários um e dois

As figuras Figura 30, Figura 31 e Figura 32 exibem os resultados de empacotamentos realizados com os arquivos `fu.xml`, `poly3b.xml` e `poly4b.xml` respectivamente, todos obtidos em Euro (2011).

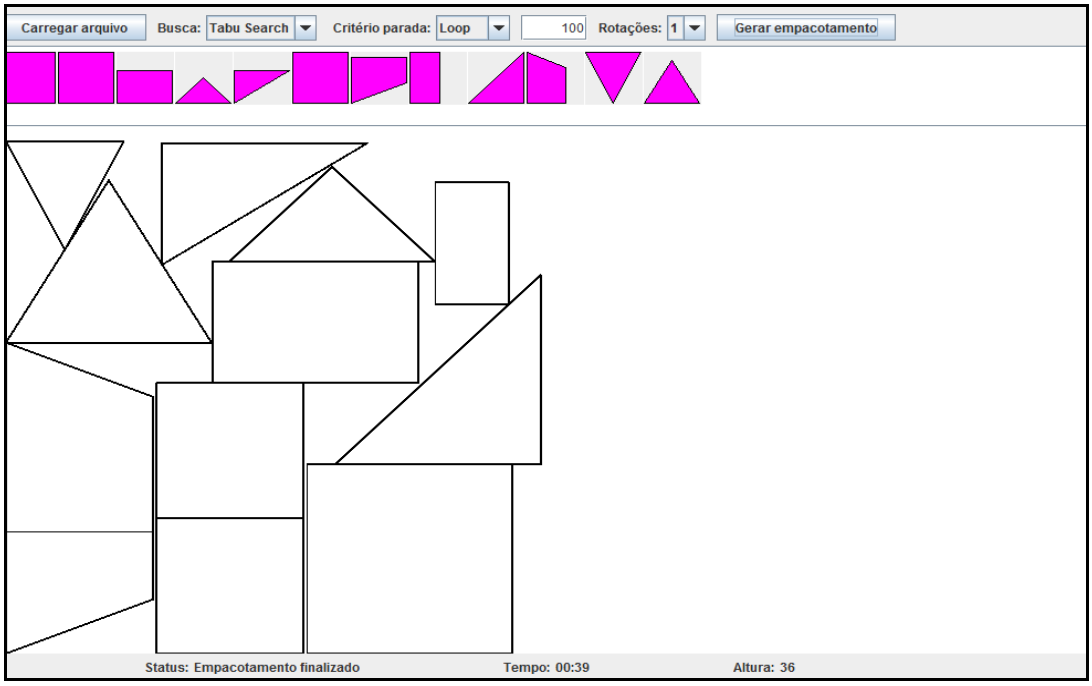


Figura 30 – Resultado de um empacotamento realizado com o arquivo `fu.xml`

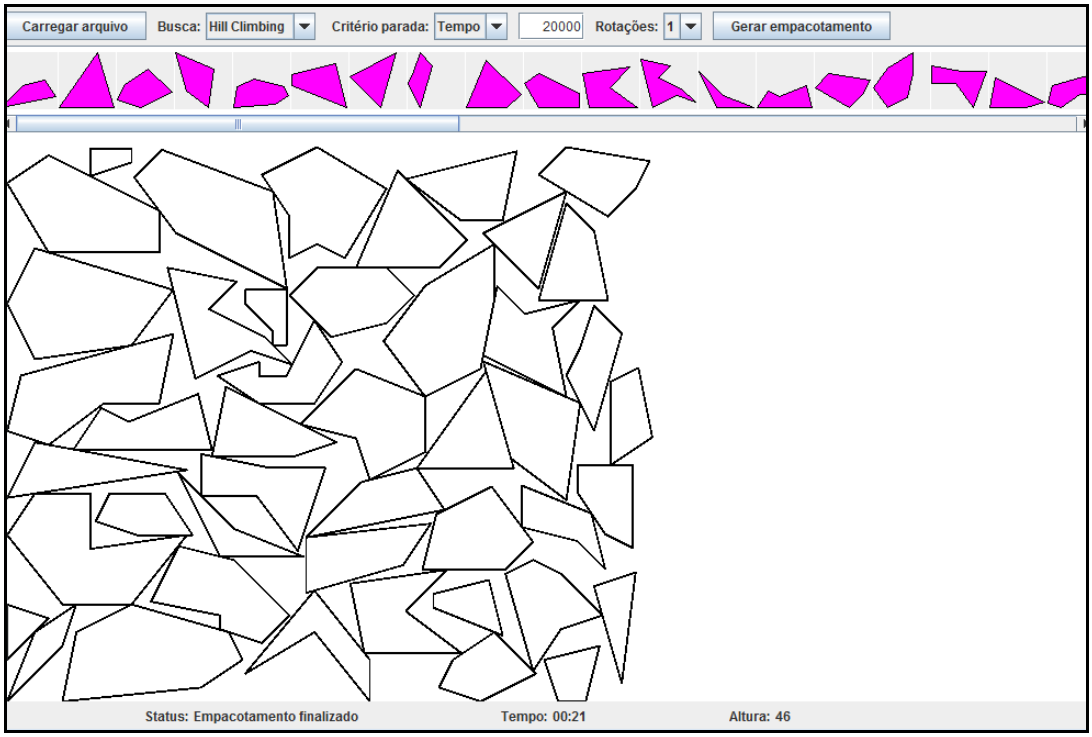


Figura 31 - Resultado de um empacotamento realizado com o arquivo `poly3b.xml`

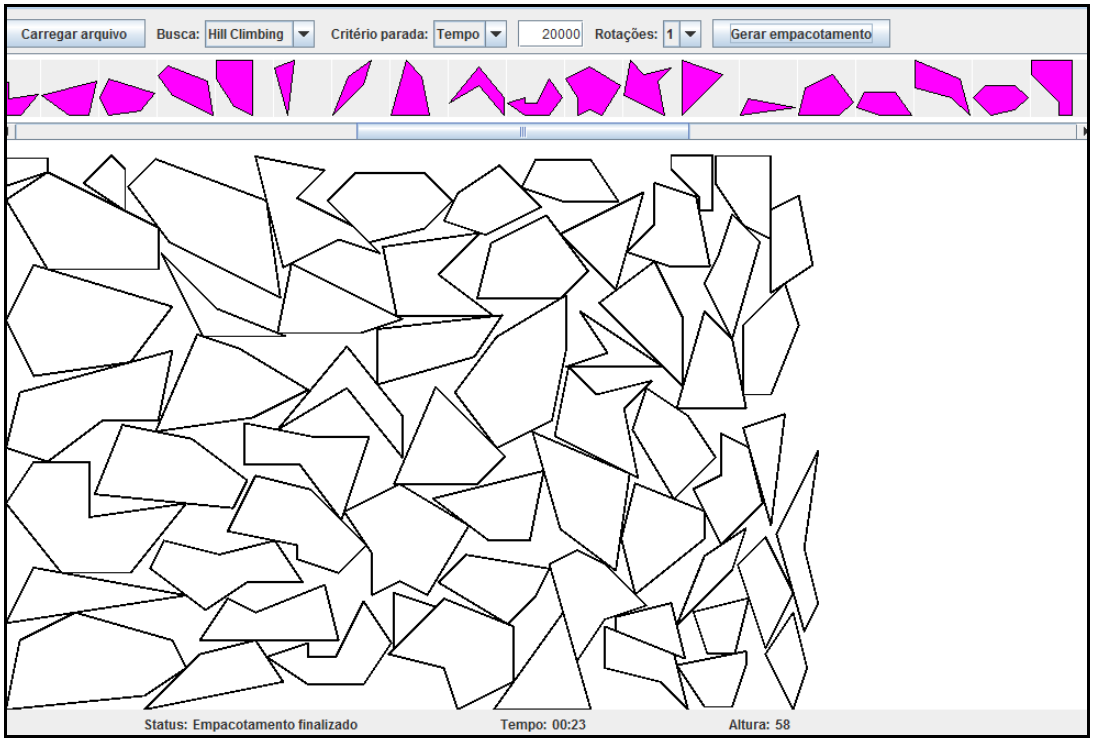


Figura 32 - Resultado de um empacotamento realizado com o arquivo `poly4b.xml`

ANEXO A – Formato do XML para a representação dos dados

A Figura 33 define o formato do XML proposto pela Euro (2011) para ser usado para testar o repositório de problemas. As informações utilizadas neste formato incluem não apenas os dados dos polígonos originais, mas também a descrição dos *no-fit polygons* e informações a respeito da melhor solução.

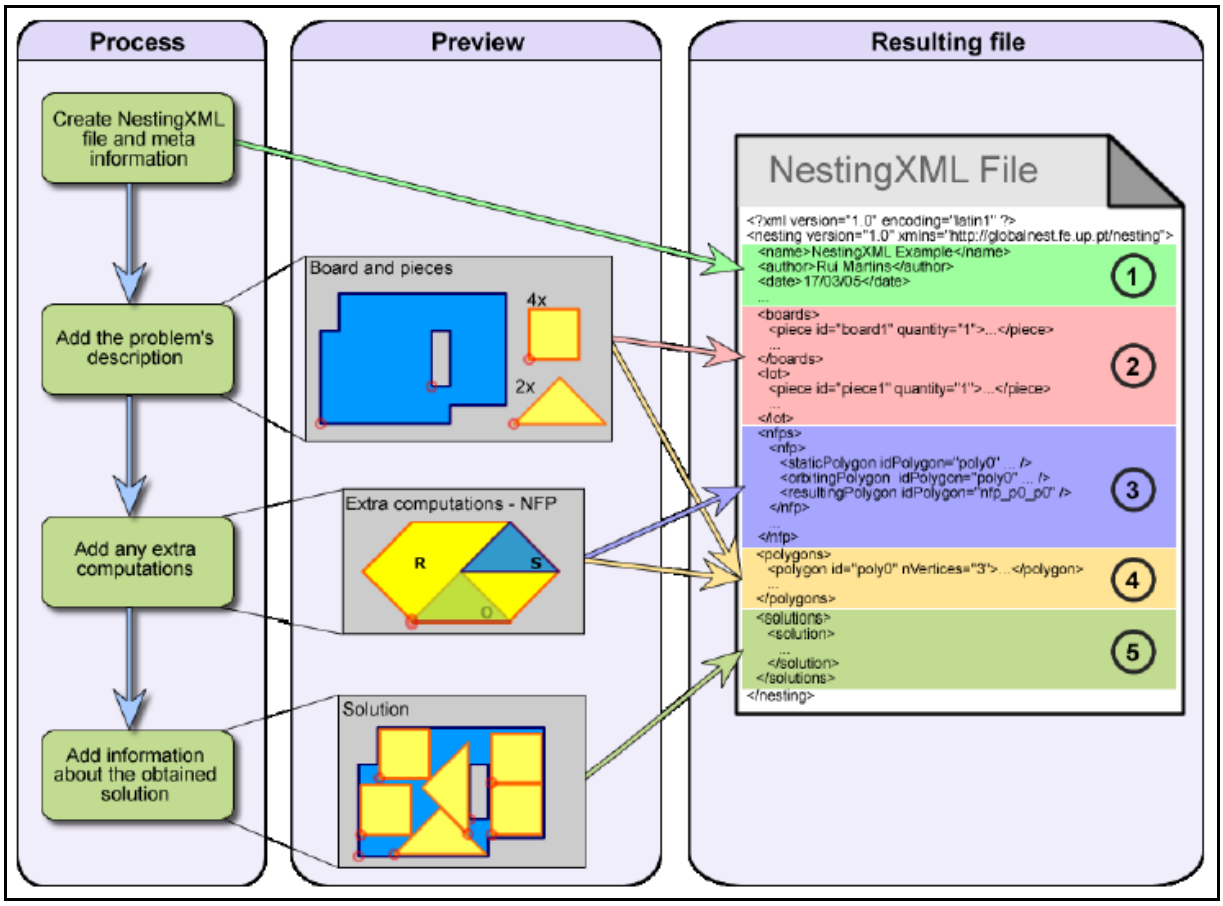


Figura 33 – Formato do arquivo XML