

FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO



FEUP

Exploração de processamento gráfico para posicionamento de formas irregulares em problemas de corte

Sofia Sampaio

Mestrado Integrado em Engenharia Informática e Computação

Orientadores:

Rui Rodrigues (Professor Auxiliar)

A. Miguel Gomes (Professor Auxiliar)

28 de Junho de 2012

Exploração de processamento gráfico para posicionamento de formas irregulares em problemas de corte

Sofia Sampaio

Mestrado Integrado em Engenharia Informática e Computação

Aprovado em provas públicas pelo Júri:

Presidente: Doutor António Augusto de Sousa (Prof. Associado da FEUP)

Arguente: Doutora Maria Teresa do Valle Moura Costa (Prof. Adjunta do Instituto Superior de Engenharia do Porto)

Vogal: Doutor Rui Pedro Amaral Rodrigues (Prof. Auxiliar Convidado da FEUP)

28 de Junho de 2012

Resumo

Os problemas de *nesting* são problemas cujo objetivo é otimizar o uso de um determinado espaço. Estudaram-se, nesta dissertação, os problemas de corte, onde se procura compactar um conjunto de formas irregulares num espaço a duas dimensões. Um exemplo da aplicação destes algoritmos é na indústria têxtil, onde é necessário determinar a disposição mais compacta de peças de vestuário numa peça de tecido, de forma a minimizar o desperdício. A uma disposição de peças no espaço disponível chama-se padrão de corte (*layout*). Neste âmbito, existem algoritmos que constroem soluções parciais de forma iterativa até obter uma solução final completa; e algoritmos que recebem uma solução completa à qual efetuam melhoramentos sucessivos (movendo as peças de sítio). Existem ainda duas representações possíveis para a geometria destes problemas: a representação numa grelha de *pixels* e a representação poligonal. A busca de soluções melhores encontra-se limitada pelo desempenho dos algoritmos, havendo que restringir as operações a efetuar e as peças a mover de forma a reduzir o custo computacional. Os principais estrangulamentos destes algoritmos relacionam-se, sobretudo, com o processamento geométrico necessário para garantir soluções admissíveis, por exemplo, padrões sem sobreposições. No caso das representações poligonais, verifica-se um custo computacional acrescido na resolução de problemas com formas complexas que implicam cálculos trigonométricos demorados e repetitivos. Neste sentido, é proposto neste trabalho transferir o processamento da geometria para o GPU, usando uma representação em grelha, de forma a diminuir o tempo de execução. Optou-se por efetuar o desenho do *layout* no dispositivo de processamento gráfico e processar a imagem do *layout* obtida, de forma a identificar a região de posicionamentos admissíveis. Esta análise permitiu-nos identificar os posicionamentos possíveis para as peças a colocar de uma forma mais eficiente e independente da geometria. Comparou-se o tempo de construção de um *layout* da implementação gráfica com os tempos de algumas abordagens poligonais referidas na literatura e observou-se uma redução significativa do tempo de execução para problemas com peças de geometria complexa.

Abstract

Nesting problems are problems whose goal is to optimize the usage of a certain space. This dissertation deals with two dimensional cutting problems. One possible application of these algorithms is the textiled manufacturing, where it is necessary to pack pieces of clothing tightly in a stock sheet in a way that minimizes the waste of cloth. A certain distribution of the pieces inside the limits of the stock sheet is called layout or cutting pattern. Some algorithms build partial solutions iteratively until they reach a final complete solution; and others take a complete solution and make successive improvements to it (by changing the positions of one or more pieces). There are two possible representations for the geometry in the context of nesting problems: raster representation (a grid of pixels) and polygonal representation. The search for better solutions is limited by the algorithms' performance. Hence, there is a need to constrain the possible moves and/or the pieces to move in order to reduce the computational cost. The main bottlenecks of these algorithms are related to the geometry processing which is necessary to produce feasible solutions, this is, patterns without overlapping. Polygonal representations have a greater computational cost when solving problems with complex shapes which imply repetitive and slow trigonometric calculations.

It is proposed to transfer the geometry processing to the GPU, using a raster representation of the cutting pattern, in order to reduce run time. The layout was rendered in the graphical device and the image produced was processed. An image processing step detects the feasible placements for a piece, called the configuration space. This analysis identifies the possible placements of a piece in a more efficient way which is also independent from the geometry.

The time it takes to build a complete layout -nesting time- in the graphical implementation was compared to the nesting times of other polygonal approaches referred on the bibliography. We noticed a significative reduction in the positioning algorithm run time for problems with complex shapes.

Agradecimentos

Agradeço ao meu orientador, o professor Rui Rodrigues, e co-orientador, o professor A. Miguel Gomes, todo o tempo dedicado e o auxílio prestado, indispensável para a realização desta dissertação. Agradeço, ainda, a revisão tão antecipada quanto possível do texto da tese.

Agradeço finalmente à minha família e amigos pelo apoio e motivação.

Sofia Sampaio

Conteúdo

1	Introdução	1
1.1	Problemas de <i>nesting</i>	2
1.2	Motivação e Objetivos	3
1.3	Estrutura do Documento	3
2	Algoritmos de <i>nesting</i>: revisão bibliográfica	5
2.1	Representação da geometria em algoritmos de <i>nesting</i>	5
2.1.1	Representações em grelha	5
2.1.2	Representações poligonais	7
2.2	Algoritmos para o cálculo dos <i>no-fit-polygons</i>	9
2.2.1	Algoritmos baseados no algoritmo de deslizamento	10
2.2.2	Algoritmos baseados na soma de Minkowski	13
2.3	Algoritmos construtivos	17
2.3.1	Regras de posicionamento	17
2.3.2	Sequências de posicionamento	20
2.4	Algoritmos de melhoramento	24
2.4.1	Movimentos	24
2.4.2	Alterações em sequências	24
2.4.3	Alterações no <i>layout</i>	26
2.5	Resumo	28
3	Dispositivos e tecnologias de processamento gráfico	31
3.1	Dispositivos de processamento gráfico	31
3.2	Tecnologias de auxílio ao desenho e processamento de imagens	33
3.2.1	<i>OpenGL</i>	33
3.2.2	<i>OpenCL</i>	36
3.2.3	<i>OpenCV</i>	36
3.2.4	<i>CGAL</i>	36
3.3	Resumo	37
4	Processamento gráfico para posicionamento de formas irregulares	39
4.1	Plataforma-base	39
4.1.1	Leitura e representação do problema	40
4.1.2	Representação da geometria	42
4.1.3	Leitura e processamento de imagem	43
4.2	Implementação de operações para posicionamento de formas irregulares	45
4.2.1	Cálculo dos NFP's	45
4.2.2	Algoritmo de posicionamento	48

CONTEÚDO

4.3	Resumo	51
5	Testes e análise dos resultados	57
5.1	Cálculo dos NFP's	57
5.2	Algoritmo de posicionamento	68
5.3	Resumo	72
6	Conclusões e trabalho futuro	75
6.1	Conclusões	75
6.2	Trabalho futuro	77
A	Características dos algoritmos de posicionamento	79
B	Tempo de execução dos algoritmos de posicionamento e para o cálculo dos NFP's	85
	Referências	103

Lista de Figuras

2.1	Exemplo de representação em grelha. [BO06]	5
2.2	Superfície irregular com uma peça colocada no canto inferior direito.[BO06]	6
2.3	Exemplo do uso da função-D. [BO06]	7
2.4	NFP de A com B.[BO06]	8
2.5	IFP da superfície A com a peça B.[GO02]	9
2.6	Duas instâncias da função-phi ($\Phi(x,y)$).[BO06]	9
2.7	Secção do vetor de deslizamento, correspondente à distância que é possível avançar sem encontrar os limites da peças A. [BHKW06]	10
2.8	Exemplo de um polígono cuja concavidade é detectável pelo algoritmo de deslizamento. [BO06]	11
2.9	Exemplos de polígonos com concavidades não detetáveis pelo algoritmo de deslizamento (a) e com encaixes perfeitos (b).	11
2.10	Arestas candidatas à translação seguinte no algoritmo de deslizamento de [BHKW06].	12
2.11	Busca de um posicionamento possível, que permita a B contornar a concavidade de A.	12
2.12	Soma de Minkowski entre os polígonos A e B. [Wik12a]	13
2.13	NFP_{AB} : soma de Minkowski entre os polígonos A e -B. [Mar09]	14
2.14	Soma de Minkowski com diagrama de declives entre um polígono côncavo e um polígono convexo. [BO06]	15
2.15	Composição do NFP_{AB} . [BO06]	16
2.16	O buraco B não é detetado pelo algoritmo. [LM11]	17
2.17	Deteção dos limites da soma de Minkowski recorrendo a <i>orthogonal e flood fill</i> . [LM11]	17
2.18	A aplicação de uma heurística de posicionamento a uma sequência determina a disposição das peças no espaço. [GO02]	18
2.19	Heurística de posicionamento <i>bottom-left</i> . [BO09]	18
2.20	Posicionamentos admissíveis num <i>layout</i> .	19
2.21	Heurística de posicionamento TOPOS (origem-flutuante). [BO09]	20
2.22	Seleção dinâmica da peça a colocar.	21
2.23	Rasterização da superfície de corte por defeito e das peças a colocar por excesso, em [BBGM12].	22
2.24	Posicionamento de uma peça numa superfície irregular.	22
2.25	Movimentos possíveis das peças numa sequência. [BO09]	24
2.26	Troca de um par de peças, dentro de um intervalo Δ . [GO02]	25
2.27	Iteração do algoritmo de <i>Jostle</i> . [BO09]	26
2.28	Pontos candidatos ao ponto de sobreposição mínima. [BO09]	26
2.29	Cálculo da sobreposição em Egeblad et al (2007). [BO09]	27
2.30	Restrições usadas para a colocação de um polígono. [BO09]	28

LISTA DE FIGURAS

2.31	Restrições definidas pelas arestas de uma concavidade num problema de programação linear. [BO09]	28
3.1	<i>Pipeline</i> de processamento gráfico. [OLG ⁺ 05]	32
3.2	Sequência de operações realizadas por um dispositivo gráfico.	33
3.3	<i>Tessellation</i> em <i>OpenGL</i>	34
3.4	Acesso a PBO's em <i>OpenGL</i>	35
3.5	Sequência de processamento de um conjunto de <i>pixels</i>	35
4.1	Diagrama de atividade da aplicação.	40
4.2	Diagrama de classes da aplicação.	41
4.3	Algoritmo de Douglas Peucker. [Wik12b]	44
4.4	Pares de peças e respectivos NFP's, calculados e desenhados recorrendo aos algoritmos implementados.	46
4.5	Exemplo das decomposições <i>angle-bisector</i> [CD85] (a) e <i>small side angle bisector</i> [AFH00] (b).	47
4.6	Posicionamento de uma peça do problema <i>swim</i> no respetivo <i>layout</i> , a partir de uma sequência dada pela heurística de corte WIDER.	49
4.7	Definição do IFP de uma superfície.	53
4.8	Comparação entre uma disposição com a margem de um <i>pixel</i> entre as peças (situação a) e sem nenhum espaçamento (situação b).	54
4.9	Avaliação do desperdício de uma disposição.	55
5.1	Conjunto de peças <i>shapes</i>	58
5.2	Conjunto de peças <i>swim</i>	58
5.3	Número de polígonos por peça (do problema <i>swim</i>) originados para as decomposições ótima, de Hertel e Mehlhorn, de Greene e <i>small side angle bisector</i>	59
5.4	Tempo das decomposições ótima, de Hertel e Mehlhorn, de Greene e <i>small side angle bisector</i> para cada peça de <i>swim</i>	59
5.5	Tempo da soma de Minkowski dos sub-polígonos resultantes das 4 decomposições, sem contabilizar o tempo da decomposição. (<i>swim</i>)	60
5.6	Tempo da soma de Minkowski dos sub-polígonos resultantes das 4 decomposições, contabilizando o tempo da decomposição. (<i>swim</i>)	61
5.7	Tempo da soma de Minkowski de [CGA] dos sub-polígonos resultantes das 4 decomposições. (<i>shapes</i>)	62
5.8	Tempo da soma de Minkowski de [CGA] dos sub-polígonos resultantes das 4 decomposições. (<i>swim</i>)	62
5.9	Tempo da soma de Minkowski e do diagrama de declives, em função do número de pares de vértices a somar para cada NFP.	63
5.10	Tempo da soma de Minkowski e do diagrama de declives, em função do número de pares de sub-polígonos a processar para cada NFP.	64
5.11	Tempos do algoritmo de deslizamento, da soma de Minkowski e do diagrama de declives em função do número de vértices côncavos de cada polígono.	65
5.12	Tempos (em milissegundos) de desenho e da aplicação da regra de posicionamento em função do número de peças colocadas, na resolução do problema <i>shirts</i> usando uma heurística estática (WIDER).	70

LISTA DE FIGURAS

- 5.13 Tempos (em milissegundos) de desenho e de escolha do melhor *layout* em função do número de tipos de peça disponíveis (em cima) e do número de peças colocadas (em baixo), usando a heurística dinâmica. As linhas verticais sinalizam que um tipo de peça foi retirado da lista dos disponíveis. 70

LISTA DE FIGURAS

Lista de Tabelas

5.1	Tempos (em segundos) de cálculo dos NFPs dos conjuntos de peças indicados do algoritmo de deslizamento de [BHKW06], soma de Minkowski e diagrama de declives.	66
5.2	Tempo (em milissegundos) da execução de $n \times n$ somas em <i>OpenCl</i> (passagem dos argumentos, execução do <i>kernel</i> e leitura dos resultados), comparado com uma implementação sequencial.	67
5.3	Número de somas de pares de pontos efetuadas por cada decomposição (uma soma corresponde à adição das coordenadas x e y de dois pontos de sub-polígonos de peças distintas).	68
5.4	Comparação das características dos 3 algoritmos testados. A avaliação da eficiência e do rigor é feita numa escala de 1 a 5.	73
A.1	Heurísticas para o posicionamento das peças em soluções parciais.	80
A.2	Regras de posicionamento das peças na construção de soluções parciais.	81
A.3	Algoritmos para melhorar soluções completas movendo as peças de uma sequência.	82
A.4	Algoritmos para melhorar soluções completas movendo as peças no espaço.	83
B.1	Tempo (em milissegundos) de cálculo dos NFPs de <i>shapes</i> recorrendo à soma de Minkowski com decomposição, disponível na biblioteca CGAL.	85
B.2	Tempo (em milissegundos) de cálculo dos NFPs de <i>swim</i> recorrendo à soma de Minkowski com decomposição, disponível na biblioteca CGAL.	86
B.3	Tempo (em milissegundos) da decomposição e do cálculo do invólucro convexo da soma de Minkowski dos sub-polígonos de <i>shapes</i>	87
B.4	Tempo (em milissegundos) da decomposição e da construção do diagrama de declives dos sub-polígonos de <i>shapes</i>	87
B.5	Tempo(em milissegundos) da decomposição e do cálculo do invólucro convexo da soma de Minkowski dos sub-polígonos de <i>swim</i>	88
B.6	Tempo (em milissegundos) da decomposição e da construção do diagrama de declives dos sub-polígonos de <i>swim</i>	89
B.7	Tempo (em milissegundos), número de sub-polígonos e número médio de vértices por subpolígono das decomposições ótima, de Hertel e Mehlhorn e <i>Small side angle bisector</i> dos polígonos de <i>shapes</i>	90
B.8	Tempo (em milissegundos), número de sub-polígonos e número médio de vértices por subpolígono das decomposições ótima, de Hertel e Mehlhorn e <i>Small side angle bisector</i> dos polígonos de <i>swim</i>	90
B.9	Tempo (em milissegundos) da soma de Minkowski (sem incluir o tempo das decomposições) dos sub-polígonos de <i>shapes</i> resultantes das decomposições ótima, de Hertel e Mehlhorn.	90

LISTA DE TABELAS

B.10	Tempo (em milissegundos) da construção do diagrama de declives dos sub-polígonos de <i>shapes</i> resultantes das decomposições ótima, de Hertel e Mehlhorn.	91
B.11	Tempo (em milissegundos) do cálculo da soma de Minkowski (sem incluir o tempo da decomposição) dos sub-polígonos de <i>swim</i> resultantes das decomposições ótima, de Hertel e Mehlhorn.	92
B.12	Tempo (em milissegundos) da construção do diagrama de declives (sem incluir o tempo da decomposição) dos sub-polígonos de <i>swim</i> resultantes das decomposições ótima, de Hertel e Mehlhorn.	93
B.13	Tempo (em milissegundos) da geração dos NFP's de <i>shapes</i> com o algoritmo de deslizamento.	94
B.14	Tempo (em milissegundos) da geração dos NFP's de <i>swim</i> com o algoritmo de deslizamento.	95
B.15	Tempo (em milissegundos) da criação da <i>display list</i> correspondente aos NFP's de <i>shapes</i> calculados na experiência anterior (tabela B.9).	96
B.16	Tempo (em milissegundos) da criação das <i>display list</i> correspondente aos NFP's de <i>swim</i> calculados na experiência anterior (tabela B.11).	97
B.17	Tempo (em milissegundos) da criação da <i>display list</i> correspondente aos NFP's de <i>shapes</i> resultantes da soma de Minkowski com decomposição (tabela B.1).	98
B.18	Tempo (em milissegundos) da criação da <i>display list</i> correspondente aos NFP's de <i>swim</i> resultantes da soma de Minkowski com decomposição (tabela B.2).	99
B.19		100
B.20	Tempo (em segundos) da representação de uma solução para os algoritmos estudados e os implementados.	101

Abreviaturas e Símbolos

API	Aplication Programming Interface
CGAL	Computational Geometry Algorithms Library
C&P	Cutting and Packing
CPU	Central Processing Unit
DL	Display List
DRAM	Dynamic Random Access Memory
GPU	Graphical Processing Unit
IFP	Inner-Fit-Polygon
NFP	No-Fit-Polygon
PBO	Pixel Buffer Object
PL	Programação Linear
RGB	Red Green Blue
RGBA	Red Green Blue Alpha
SIMD	Single Instruction Multiple Data

Capítulo 1

Introdução

Os problemas de corte e empacotamento (Cutting & Packing - C&P) consistem em encontrar a disposição ótima de um conjunto de peças dentro de um dado espaço, de forma a maximizar um determinado objetivo. São problemas que se caracterizam por apresentarem simultaneamente uma forte componente combinatória e uma forte componente geométrica, o que torna a sua resolução particularmente difícil. Wäscher et al. [WHS07] classificam estes problemas segundo:

- o tipo de objetivo - sendo classificados como de *output maximization* (quando se pretende maximizar o número ou valor das peças empacotadas) ou de *input minimization* (quando se pretende minimizar o espaço não utilizado);
- o número de dimensões relevantes - 1D, 2D, 3D, nD ($n > 3$);
- a heterogeneidade das peças - sendo classificados como homogêneos (quando as peças a serem colocadas são todas iguais), vagamente heterogêneos ou fortemente heterogêneos (quando todas as peças apresentam formas diferentes);
- a geometria das peças - sendo classificados de regulares (quando as peças apresentam formas regulares: círculos, retângulos, triângulos, etc) ou irregulares (quando as peças apresentam formas irregulares).

O objeto de estudo são os problemas de *nesting* em que o objetivo é minimizar o desperdício. Os problemas de *nesting* são problemas a 2 dimensões cujas peças são irregulares, isto é, de dimensão e forma variáveis. Pretende-se encontrar uma distribuição compacta para um conjunto de polígonos irregulares que serão cortados numa dada superfície de forma a minimizar o desperdício. Grande parte destes problemas enquadram-se na categoria de *strip packing*, em que existe um conjunto de peças a ser cortado numa peça de tecido retangular de largura definida e comprimento indeterminado. São problemas de *input minimization* em que se pretende minimizar o comprimento de tecido necessário para acomodar todas as peças, reduzindo o espaço inutilizado entre elas (desperdício).

Outro tipo de desafios que tiram partido da abordagem utilizada são problemas em que a superfície de corte é irregular, isto é, tem uma forma poligonal com defeitos (concauidades e buracos) e há que alojar as peças tendo em conta as reentrâncias.

Este tipo de problemas têm uma forte componente geométrica que se torna de difícil processamento dada a irregularidade das peças. No sentido de suavizar o impacto da complexidade das formas no tempo de resolução do problema, recorrer-se-á a técnicas da Computação Gráfica para lidar com a geometria. Nomeadamente, procurar-se-á tirar o máximo partido dos dispositivos de processamento gráfico que têm vindo a provar-se eficientes e cada vez mais versáteis.

Os dispositivos de processamento gráfico dedicam-se ao tratamento dos elementos gráficos e geométricos, produzindo as imagens apresentadas no ecrã. Possuem uma *pipeline* de processamento gráfico que converte um conjunto de representações vetoriais de objetos poligonais em matrizes de *pixels* (*rasterization*). Este *hardware* disponibiliza ainda inúmeras opções de desenho, permitindo configurar os detalhes da representação dos distintos tipos de formas. Atualmente, os GPU's (**Graphics Processing Unit**) permitem ainda tirar partido da sua arquitetura SIMD para a realização de processamento paralelo em problemas com um grande paralelismo de dados.

No âmbito desta dissertação, tirar-se-á partido, sobretudo, da capacidade de processamento geométrico destes dispositivos visto o tratamento da geometria ser um dos principais *bottlenecks* (isto é, uma das etapas que mais limita a *performance*) dos algoritmos de *nesting*.

1.1 Problemas de *nesting*

Os problemas de *nesting* são np-completos, pelo que não é possível encontrar a solução ótima em tempo útil. Desta forma, é essencial agilizar o mais possível a busca de soluções que aproximem a ótima. Os métodos utilizados até ao momento são pouco flexíveis, limitando, frequentemente, a rotação das peças. Para além das limitações impostas pelo algoritmo utilizado, há que considerar as restrições do próprio problema que obrigam as peças a estar alinhadas segundo o padrão do tecido.

As principais dificuldades surgem no processamento da geometria dos polígonos a colocar a fim de identificar sobreposições e saber a posição relativa de uns em relação aos outros. Algumas abordagens admitem um certo grau de sobreposição entre as peças em fases intermédias da busca de soluções, mas numa solução final praticável, essas sobreposições não podem existir. Inicialmente, era vulgar representar o padrão de corte numa grelha, o que simplificava os cálculos mas era demasiado custoso em termos de memória e de processamento, para além de restringir a precisão. Atualmente, recorre-se a representações poligonais e a conceitos da geometria (NFP's) para evitar alguns dos cálculos complexos sem perder informação relevante. Porém ambas as abordagens têm limitações e tardam a produzir uma boa solução. A maioria das implementações consideram apenas superfícies retangulares, o que é suficiente para grande parte das indústrias pois é essa a forma da matéria prima. No entanto, no que diz respeito, por exemplo, ao corte de

peles há que considerar superfícies de corte com irregularidades e defeitos, detalhes para os quais a maioria dos algoritmos existentes se revela inadequada.

1.2 Motivação e Objetivos

No seguimento das limitações enunciadas na secção anterior (1.1), torna-se interessante recorrer a uma ferramenta adequada ao processamento geométrico, como o GPU, para processar certos detalhes da geometria nos algoritmos de *nesting*.

A abordagem a implementar basear-se-á no algoritmo de Gomes e Oliveira [GO02]. Este algoritmo utiliza uma representação poligonal das peças e da superfície de corte. As rotações admissíveis são pré-definidas. Uma solução é representada por uma sequência de peças que são colocadas por ordem na superfície, segundo uma regra que vai encostar cada peça o mais possível à esquerda e abaixo (até encontrar outra peça ou os limites da superfície).

Partindo deste algoritmo, procurou-se identificar as tarefas que mais prejudicam o rendimento dos algoritmos existentes e, avaliar as potencialidades dos GPU's para as processar de forma mais eficiente. O tempo de execução dos algoritmos que utilizam representações poligonais é muito elevado se as peças tiverem formas complexas, com um grande número de vértices e concavidades. A implementação realizada aproveitará a facilidade dos dispositivos gráficos na representação da geometria e a memória que estes disponibilizam. Opta-se por uma representação em grelha do *layout* que, nos GPU's, equivale a uma matriz de *pixels*. Espera-se que esta representação torne o algoritmo mais flexível e independente da geometria, tornando mais rápido o processo de colocação de uma peça nova no conjunto e agilizando a geração de soluções. As diferenças principais entre a nova abordagem e a original serão:

- a utilização de uma matriz de *pixels* em vez da representação poligonal original;
- o recurso a algoritmos para o cálculo dos NFP's que facilitem e beneficiem de uma representação gráfica;
- o recurso a processamento de imagem para determinar espaços vazios e sobreposições entre as peças colocadas;

Avaliar-se-á, posteriormente, o desempenho da implementação realizada a fim de a comparar com as abordagens tradicionais.

1.3 Estrutura do Documento

Para além da introdução, onde é enunciado o problema e apresentadas a motivação e os objetivos deste trabalho, esta dissertação contém mais 4 capítulos. No capítulo 2, é apresentado o estado da arte. São descritas as formas de representar a geometria neste tipo de problemas (secção 2.1) e os principais algoritmos de posicionamento. Sendo um conceito de maior importância para a resolução deste tipo de problemas, a secção 2.2 deste capítulo é dedicada aos algoritmos

Introdução

para o cálculo dos NFP's. Os algoritmos de posicionamento podem ser construtivos (secção 2.3), gerando uma solução através de um critério estipulado, ou de melhoramentos (secção 2.4), melhorando uma solução completa. No final do documento (apêndice A) encontram-se as tabelas que resumem as características dos algoritmos estudados. O capítulo 3 introduz os dispositivos de processamento gráfico (secção 3.1), referindo, ainda, as tecnologias utilizadas no desenvolvimento da plataforma (secção 3.2). No capítulo 4 encontram-se os detalhes relativos à implementação: inicia-se com uma explicação da arquitetura e do funcionamento da aplicação (secção 4.1), seguindo-se a descrição dos algoritmos implementados para o cálculo dos NFP's e para o posicionamento das peças no *layout* (secção 4.2). Os resultados dos testes e a comparação dos resultados com os de outros algoritmos disponíveis na literatura encontram-se no capítulo 5. Este capítulo encontra-se dividido em duas secções: secção 5.1, correspondente aos testes dos algoritmos destinados ao cálculo dos NFP's e secção 5.2, destinada aos testes realizados com o algoritmo de posicionamento, para vários critérios de posicionamento. Grande parte das tabelas com os resultados dos testes referidos encontram-se no anexo B. Finalmente, o capítulo 6 resume os principais pontos da investigação realizada e sintetiza as conclusões mais relevantes.

Capítulo 2

Algoritmos de *nesting*: revisão bibliográfica

Estudaram-se as principais técnicas para resolver problemas de *nesting* a fim de compreender a evolução deste tipo de algoritmos e identificar as abordagens mais promissoras. Neste capítulo descrevem-se os algoritmos estudados, bem como os métodos utilizados para representar o padrão de corte (*layout*). Inicialmente, resumem-se as várias formas de representar a geometria e refere-se o impacto das mesmas na resolução do problema. De seguida, explica-se o conceito de *no-fit-polygon*, referindo a sua importância para a resolução deste tipo de problemas. As abordagens para construir e melhorar um *layout* são listadas nas duas secções seguintes.

2.1 Representação da geometria em algoritmos de *nesting*

Para ser possível gerar soluções satisfatórias para este tipo de problemas é essencial representar a sua geometria de forma eficiente [BO06].

2.1.1 Representações em grelha

Nesta abordagem, o *layout* é representado numa grelha, cujas células assumem diferentes valores consoante aquele ponto esteja vazio ou, pelo contrário, ocupado por uma ou mais peças (sobrepostas) (figura 2.1).

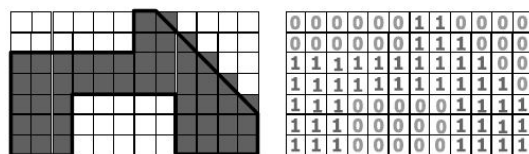


Figura 2.1: Exemplo de representação em grelha. [BO06]

Esta representação tem a vantagem de ser simples de implementar e rápida de processar. Porém, traz problemas ao nível da precisão (pois o nível máximo de detalhe corresponde à resolução da grelha) e ao nível da memória. Na busca de soluções, a falta de precisão pode eliminar posicionamentos vantajosos uma vez que apenas se efetua a pesquisa nos pontos definidos pela grelha.

Para solucionar o problema do consumo de memória é possível recorrer a *quadtrees*. As *quadtrees* são estruturas de dados – árvores que possuem exatamente quatro filhos por nodo - que armazenam informação relativa a uma imagem (a duas dimensões). Os GPU's são capazes de mapear estas estruturas, porém, aceder e atualizar a informação torna-se um processo mais lento. Uma forma de atualizar eficientemente informação muito dinâmica é utilizar tabelas de páginas (*page tables*) que, apesar de necessitarem de mais espaço de armazenamento, permitem acessos à memória em tempo constante e admitem modificações dos dados em paralelo [OLG⁺05].

Independentemente da estrutura usada para o armazenamento em memória, as representações em grelha divergem no que diz respeito aos valores usados para indicar o conteúdo de uma célula. Diferentes algoritmos representam as células vazias, a existência de uma peça ou sobreposições de maneiras distintas.

No trabalho de Oliveira e Ferreira [OF93a] um *pixel* vazio assume o valor 0. Por cada peça colocada soma-se 1 unidade aos valores dos *pixels* por ela ocupados. Desta forma, 1 representa a existência de uma peça naquela célula e um valor superior a 1 indica a sobreposição entre várias peças.

Segenreich e Braga [Seg86] optam por representar a fronteira das peças com valor 1 e o seu interior com o valor 3. Seguindo-se o mesmo processo em que se somam os valores de cada vez que se coloca uma peça, conclui-se que se os limites de duas peças estiverem em contacto o valor desses *pixels* será 2; por outro lado, se existir sobreposição, os *pixels* com peças sobrepostas terão valores superiores a 4.

Outra alternativa, proposta por [RR01], é representar o interior da superfície com zeros e os seus limites com um número que indique quantas células se tem que avançar para a direita até encontrar um *pixel* vazio. As peças por colocar representam-se de forma semelhante ao *layout*, mas a fronteira da peça também é codificada por zeros; apenas os *pixels* exteriores à peça contam o número de células ocupadas à direita (figura 2.2).

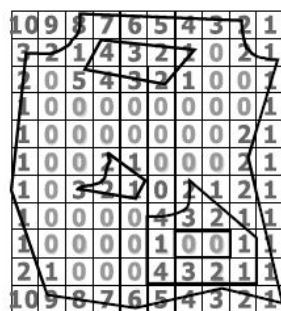


Figura 2.2: Superfície irregular com uma peça colocada no canto inferior direito.[BO06]

Desta forma, torna-se mais eficiente aplicar uma regra de posicionamento tipo *bottom-left* (2.3.1), que escolhe a posição vazia mais abaixo e mais à esquerda para colocar a próxima peça, avançando várias células ocupadas de uma só vez (o número indicado pelo *pixel*).

Esta técnica é eficaz a reduzir o tempo de busca da próxima posição livre para uma nova peça, mas a tarefa de atualizar o *layout* após a sua colocação revela-se uma operação morosa.

2.1.2 Representações poligonais

Neste tipo de representações uma peça é representada por um conjunto de vértices, sendo possível construir a figura através das arestas por eles definidas. É uma representação económica em termos de memória mas que dificulta o processamento geométrico, nomeadamente no que diz respeito à identificação das posições livres no *layout*.

2.1.2.1 Conceitos utilizados pelas representações poligonais

No sentido de evitar cálculos trigonométricos complexos e repetitivos que atrasam o processamento na detecção de sobreposições e identificação da posição relativa das peças surge uma série de conceitos, nomeadamente o da função-D, *no-fit-polygon* e *inner-fit-polygon* e a função-phi. As arestas do NFP(*no-fit-polygon*) de duas peças correspondem às posições em que existe contacto entre elas, mas sem sobreposição (ver parágrafo 2.1.2.1). O IFP(*inner-fit-polygon*) delimita a região de uma superfície onde se pode colocar uma dada peça sem que esta ultrapasse os limites da superfície (ver parágrafo 2.1.2.1). A função-D, por sua vez, permite determinar rapidamente a posição de um ponto relativamente a um vetor enquanto que a função-phi traduz a posição relativa entre dois polígonos indicando, não só se existe contacto entre eles (como os NFP's), mas também a distância a que se encontram.

Função-D Esta função permite determinar o posicionamento de um ponto P relativamente a um vetor AB (figura 2.3). É dada pela seguinte fórmula

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

que não é mais do que uma simplificação da fórmula de cálculo da distância entre um ponto e uma reta.

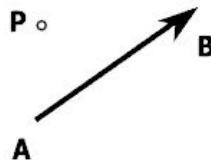


Figura 2.3: Exemplo do uso da função-D. [BO06]

Da observação da fórmula pode-se concluir que:

- se $D_{ABP} > 0$, P encontra-se do lado esquerdo de AB ;
- se $D_{ABP} = 0$, P está sobre a linha de AB ;
- se $D_{ABP} < 0$, P está do lado direito.

Esta abordagem revela-se útil no teste de interseção entre duas retas uma vez que, analisando as posições entre os pontos que definem dois vetores distintos, é possível inferir a posição relativa entre eles e saber se se intersectam [Mah84].

No-fit-polygon O NFP entre dois polígonos A e B (NFP_{AB}), com os respetivos pontos de referência R_A e R_B , representa a área em que se pode colocar R_B sem que haja sobreposição entre os dois polígonos. Pode-se obter fazendo deslizar B em torno das arestas de A de forma a que ambos os polígonos estejam sempre em contacto mas nunca se sobreponham; o NFP é o polígono descrito pelo trajeto do ponto de referência de B, R_B .

Desta forma, se o ponto de referência de B estiver dentro de NFP_{AB} , A e B estão sobrepostos; se R_B estiver na fronteira de NFP_{AB} , os polígonos estão em contacto (figura 2.4).

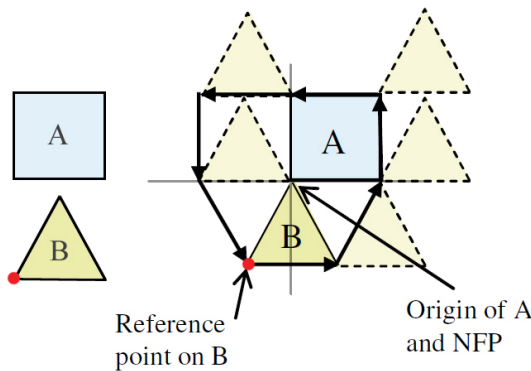


Figura 2.4: NFP de A com B.[BO06]

Inner-fit-polygon O IFP entre uma superfície A e uma peça B (IFP_{AB}), com o ponto de referência em R_B , representa a área em que se pode colocar R_B sem que o polígono B ultrapasse os limites da superfície A. Pode-se obter fazendo deslizar B pelo interior da superfície A, mantendo sempre o contacto com as suas arestas. O trajeto descrito pelo ponto de referência de B, R_B , constitui o IFP_{AB} .

Desta forma, se o ponto de referência de B estiver dentro de IFP_{AB} , B encontra-se dentro dos limites da superfície A; se R_B estiver na fronteira de IFP_{AB} , a peça B encontra-se encostada aos limites da superfície A (figura 2.5).

Função-Phi A função-phi é uma expressão matemática que representa a posição relativa entre dois objetos. Assume valores maiores que 0 se os dois objetos estão separados, igual a 0 se

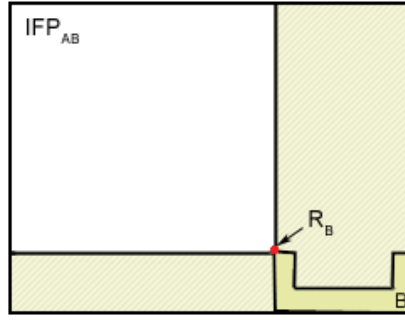


Figura 2.5: IFP da superfície A com a peça B.[GO02]

estiverem em contacto e menor que 0 se estiverem sobrepostos ([STS⁺01],[SSGR02]). Trata-se de uma generalização do NFP, isto é, o NFP corresponde ao nível 0 da função-phi, onde os polígonos estão encostados (figura 2.6).

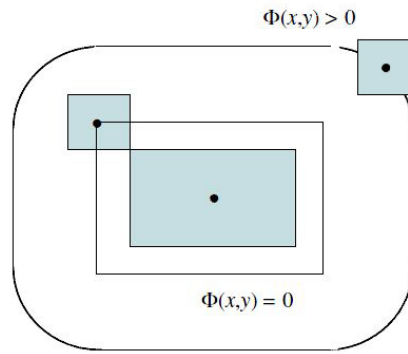


Figura 2.6: Duas instâncias da função-phi ($\Phi(x,y)$).[BO06]

Numa das instâncias $\Phi(x,y) = 0$ coincide com o NFP e na outra $\Phi(x,y) > 0$.

2.2 Algoritmos para o cálculo dos *no-fit-polygons*

As representações poligonais recorrem ao conceito dos NFP's para identificar a área do *layout* onde é possível colocar novas peças sem que ocorra sobreposição. Estes polígonos podem ser calculados em pré-processamento (quando os polígonos estão fixos ou apenas admitem rotações discretas) pois é um passo que demora algum tempo, sobretudo se se tratar do caso de dois polígonos irregulares não convexos.

O NFP pode ser obtido com base nas seguintes duas abordagens distintas: as baseadas no **algoritmo de deslizamento**, em que o segundo polígono desliza em redor do primeiro ocupando as posições em que existe contacto mas não sobreposição (ver figura 2.4); e as baseadas na **soma**

de **Minkowski** (soma de todos os pares de pontos de dois polígonos) através da qual se pode deduzir o NFP.

2.2.1 Algoritmos baseados no algoritmo de deslizamento

Algoritmo de deslizamento O algoritmo de deslizamento, inicialmente proposto por Mahadevan [Mah84], começa com as duas peças, A e B, em contacto, por exemplo fazendo com que o vértice com a maior coordenada y de B esteja em contacto com o vértice com a menor coordenada y de A. A partir desta configuração, a peça B desliza em torno de A, descrevendo o NFP com o seu ponto de referência. No caso de ambos serem polígonos convexos, B avança, sucessivamente, todo o comprimento de cada aresta de A. Este processo repete-se até o polígono B ter contornado completamente o polígono A.

No caso de polígonos côncavos há que ter atenção que pode não ser possível avançar todo o comprimento de uma aresta de cada vez (por exemplo, no caso de B se encontrar numa concavidade de A). A fim de evitar a sobreposição entre os dois polígonos, os vértices de B são projetados na direção e no comprimento da aresta segundo a qual B desliza. Utiliza-se a função-D para testar a interseção destas projeções com o polígono A. Avança-se apenas a distância mais curta das interseções identificadas, evitando, desta forma, a sobreposição dos dois polígonos (figura 2.7). Na figura 2.8 encontra-se o exemplo de uma concavidade detetada pelo algoritmo de deslizamento.

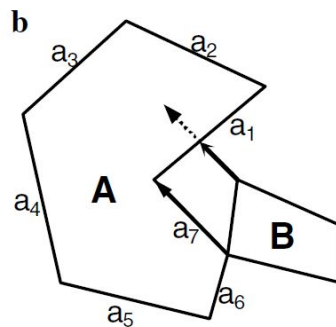


Figura 2.7: Secção do vetor de deslizamento, correspondente à distância que é possível avançar sem encontrar os limites da peças A. [BHKW06]

Algoritmo de deslizamento com detecção de buracos O algoritmo anterior não identifica os posicionamentos possíveis de B dentro dos buracos do polígono A; de forma semelhante, também não são detetados posicionamentos nas concavidades de A que sejam demasiado estreitas para que B deslizar para dentro delas apesar de, no seu interior, serem suficientemente largas para acomodar B (figura 2.9).

Burke et. al. [BHKW06] estende o algoritmo de Mahadevan de forma a contemplar estes casos particulares, implementando, ainda, algumas melhorias relativamente ao procedimento original.

O primeiro passo do algoritmo de deslizamento é detetar as arestas de A que estão em contacto com as de B, dada a posição inicial dos dois polígonos e avaliar as translações possíveis a

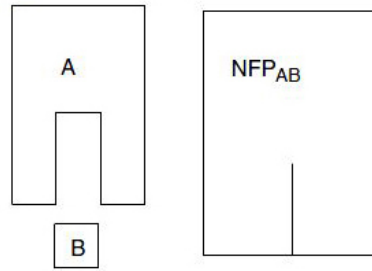


Figura 2.8: Exemplo de um polígono cuja concavidade é detectável pelo algoritmo de deslizamento. [BO06]

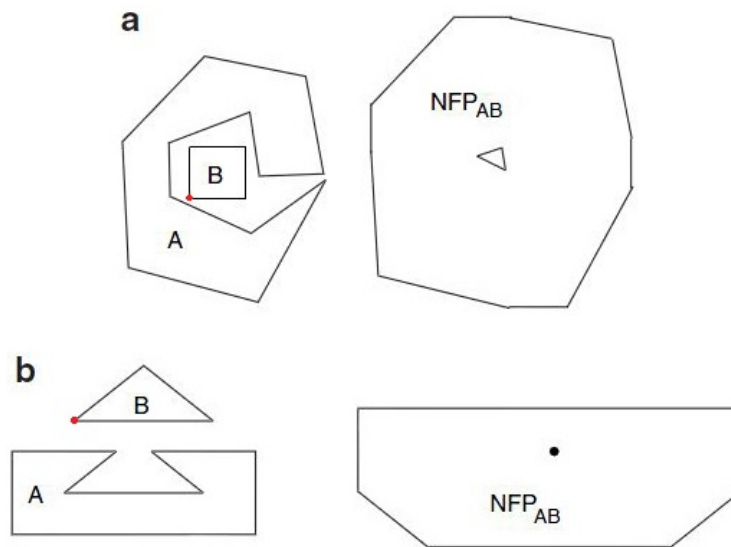


Figura 2.9: Exemplos de polígonos com concavidades não detectáveis pelo algoritmo de deslizamento (a) e com encaixes perfeitos (b).

O ponto de referência do polígono móvel encontra-se assinalado no canto inferior esquerdo. [BO06]

partir dessa posição. O algoritmo de [BHKW06] simplifica este processo, analisando todos os pares possíveis de arestas em vez de considerar todo o conjunto de uma só vez como [Mah84]. Nesta implementação tem-se ainda em atenção que, na eventualidade de existirem várias arestas sobre as quais se possa efetuar o deslizamento, é escolhida a que estiver mais próxima da aresta correspondente ao deslizamento anterior (ver figura 2.10).

Tanto Mahadevan como Burke têm em atenção as situações em que é necessário "cortar" o vetor de deslizamento dada a existência de concavidades que limitam a translação (figura 2.7), porém, enquanto que o último vai reduzindo a distância a avançar à medida que testa as projeções possíveis, Mahadeven apenas estipula a norma do vetor após testar todas as opções. A abordagem

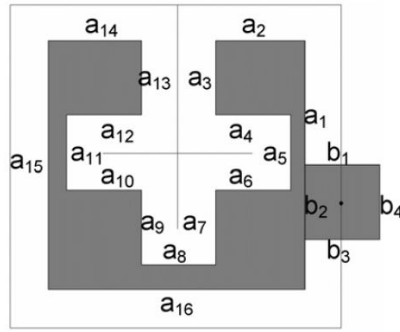


Figura 2.10: Arestas candidatas à translação seguinte no algoritmo de deslizamento de [BHKW06].

Quando o quadrado se encontra no meio da concavidade, tem 4 possibilidades por onde deslizar: a_4 , a_7 , a_{10} e a_{13} , porém, visto a última aresta pela qual deslizou ser a_3 , é escolhida a aresta a_4 para a translação seguinte.

de [BHKW06] pode constituir uma melhoria já que, como o comprimento do vetor vai diminuindo, a probabilidade de encontrar novas intersecções é mais reduzida.

Uma vez detetado o contorno exterior, procede-se à detecção de buracos. Este passo baseia-se no facto de que, se uma aresta de A não foi percorrida, pode pertencer a um buraco do NFP. Desta forma, dada uma aresta por percorrer, identifica-se o vértice de B que pode deslizar sobre ela.

Assumindo que as arestas do polígono A têm o sentido anti-horário, o vértice de B escolhido deve cumprir a condição de que as arestas que o formam se encontram do lado direito da aresta de A (isto é, nenhuma delas se sobrepõe a A). Para avaliar se o resto do polígono está sobreposto a A segue-se um procedimento análogo ao usado para evitar sobreposições em polígonos com concavidades: projetam-se em B (na direção e no comprimento da aresta sobre a qual B desliza) os vértices de A e projetam-se em A (na mesma direção e comprimento) os vértices de B. Calculam-se as respetivas intersecções e avança-se a menor distância entre elas. Repete-se este processo até encontrar um posicionamento livre de sobreposições (figura 2.11).

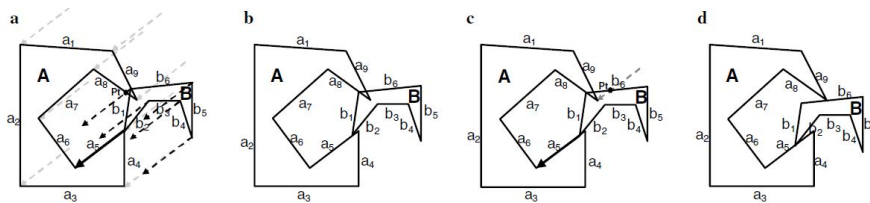


Figura 2.11: Busca de um posicionamento possível, que permita a B contornar a concavidade de A.

Após encontrar uma posição inicial para ambos os polígonos, aplica-se o algoritmo de deslizamento original para encontrar o NFP para aquele conjunto de arestas.

Este processo permite detetar buracos, encaixes perfeitos e retas mas, para polígonos complexos (com um grande número de concavidades, buracos e arestas), pode não ser o procedimento mais eficiente.

2.2.2 Algoritmos baseados na soma de Minkowski

Soma de Minkowski A soma de Minkowski obtém-se somando todos os pares de pontos de dois conjuntos (polígonos). A sua definição formal é a seguinte:

$$A \oplus B = \{a + b \mid a \in A, b \in B\}$$

Por exemplo, a soma de Minkowski dos polígonos $A(0,1), (1,0), (0,-1)$ e $B(0,0), (1,1), (1,-1)$ seria $A \oplus B(1,0), (2,1), (2,-1), (0,1), (1,2), (1,0), (0,-1), (1,0), (1,-2)$ (ver figura 2.12). Note-se que o ponto $(1,0)$ não pertence ao contorno da soma de Minkowski (assinalado a vermelho). Isto porque não são considerados os pontos que se encontram dentro do seu contorno exterior. No caso da soma de polígonos convexos, estes limites coincidem com o invólucro convexo (*convex hull*)¹ dos pontos resultantes, sendo fáceis de obter; tal já não acontece quando os polígonos possuem concavidades e buracos.

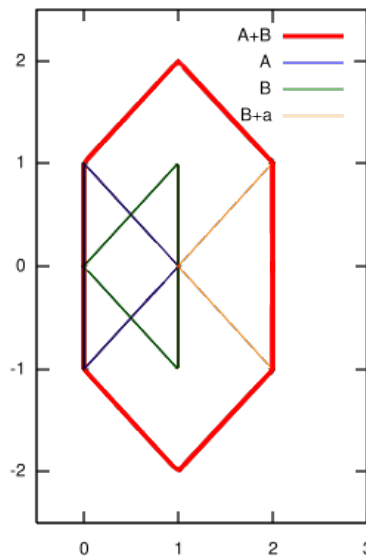


Figura 2.12: Soma de Minkowski entre os polígonos A e B. [Wik12a]

O NFP para dois polígonos A e B é $A \oplus -B$, isto é, A mais o simétrico de B em relação à origem (figura 2.13). Esta relação foi enunciada por Stoyan e Ponomarenko [Sto77].

¹o invólucro convexo de um conjunto de pontos X é o subconjunto mínimo desses pontos que contém X. Esse subconjunto forma um polígono convexo que inclui todos os pontos de X.

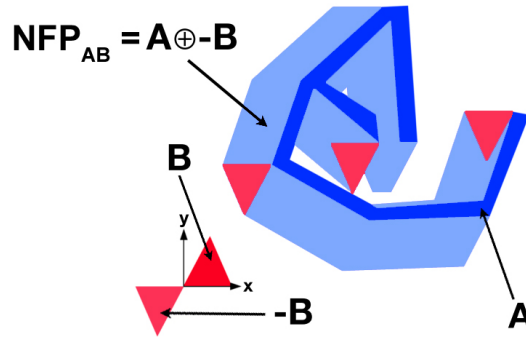


Figura 2.13: NFP_{AB} : soma de Minkowski entre os polígonos A e -B. [Mar09]

Soma de Minkowski com diagrama de declives Nas situações em que pelo menos um dos polígonos for côncavo, é necessário usar um procedimento diferente para obter o NFP. Ghosh [Gho91] opta por construir um diagrama de declives com as arestas de ambos os polígonos.

Se ambos os polígonos forem convexos, o diagrama de declives mantém a sequência de arestas de cada um. O NFP obtém-se percorrendo o diagrama numa direção qualquer (assume-se que o sentido anti-horário é positivo) e adicionando as arestas encontradas.

Se, por outro lado, um dos polígonos for côncavo, as arestas do polígono côncavo, se percorridas na sequência em que se encontram no polígono, não estarão ordenadas por declive, devido à concavidade.

Por este motivo, as arestas do segundo polígono B (convexo) que, no diagrama de declives, se encontrarem entre as arestas de A que formam a concavidade, repetir-se-ão na construção do NFP de cada vez que, ao percorrer a concavidade, se passar por elas. Quando se atravessam no sentido positivo (entrar na concavidade), adicionam-se ao NFP com sinal positivo, quando se atravessam no sentido negativo (sair da concavidade) adicionam-se ao NFP com sinal negativo (figura 2.14).

Este método é eficaz a encontrar os NFP para todos os pares de polígonos côncavos ou convexos, desde que, no caso de serem ambos côncavos, não haja interação entre as duas concavidades pois existirão áreas do diagrama de declives em que nenhuma das arestas de nenhum dos polígonos estarão ordenadas por declive.

Soma de Minkowski com um perímetro convexo A proposta de Bennell et al [BDD01] para resolver as limitações do algoritmo de Gosh quando duas concavidades interagem é substituir as arestas côncavas do segundo polígono B por arestas *dummy* que simulem que se trata de um polígono convexo, $conv(B)$.

Feita a substituição aplica-se o algoritmo acima, obtendo o NFP de A, $conv(B)$. Finalmente, para concluir o NFP_{AB} , substituem-se, em $NFP_{Aconv(B)}$ as arestas *dummy* pelas originais. Contudo, esta solução revela-se falaciosa no que diz respeito à substituição das arestas *dummy*, nomeadamente quando existem colisões.

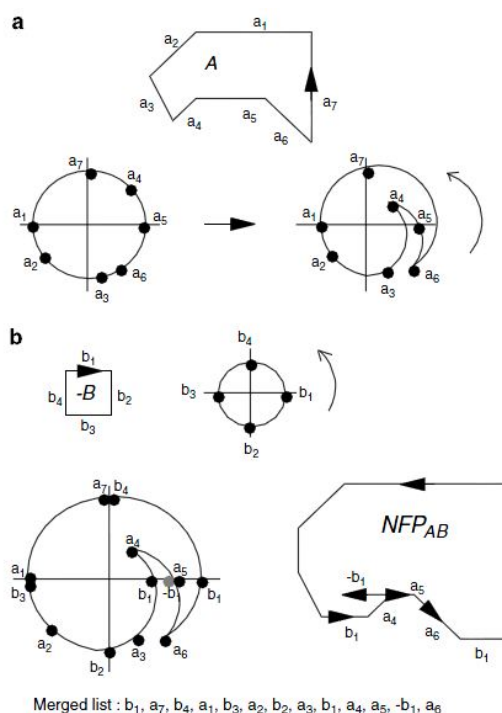


Figura 2.14: Soma de Minkowski com diagrama de declives entre um polígono côncavo e um polígono convexo. [BO06]

Soma de Minkowski com divisão de um dos polígonos em grupos de arestas A fim de solucionar os problemas que surgiam na altura de substituir as arestas convexas no algoritmo anterior, Bennell e Song [BS08] propõem dividir um dos polígonos (B) em grupos de arestas consoante estas sejam convexas (aparecem no sentido anti-horário) ou côncavas (surgem no sentido horário).

Traçam-se, de seguida, os diagramas de declive do polígono A com cada partição de arestas de B e segue-se o algoritmo original de Gosh. No final ter-se-á vários conjuntos de arestas que será necessário unir para obter o NFP.

O polígono resultante da união simples de todos os conjuntos possui arestas internas e interseções que devem ser eliminadas. Neste passo são identificadas todas as interseções entre as secções de arestas (*polygonal trips*) do NFP, escolhendo aquelas que fazem parte dos limites exteriores.

Soma de Minkowski com decomposição de um dos polígonos Outra alternativa é decompor A e B em subpolígonos convexas.

De seguida calcula-se o NFP para cada par de subpolígonos de A e B e, finalmente, compõem-se todos os NFP obtidos de maneira a formar o NFP_{AB} (figura 2.15).

O passo mais demorado deste processo é combinar os NFP dos sub-polígonos convexas visto que há que calcular interseções e decidir que secções de arestas vão formar o perímetro do NFP_{AB} . Desta forma, é conveniente reduzir a decomposição de polígonos ao mínimo de forma a minimizar estes cálculos. Watson e Tobias [WT99] e Agarwal et al [AFH02] efetuam a decomposição em polígonos convexas.

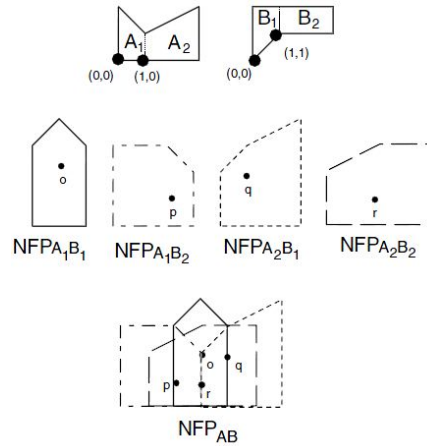


Figura 2.15: Composição do NFP_{AB} . [BO06]

Contudo, obter a decomposição ótima de um polígono em sub-polígonos convexos é um processo demorado e é preferível utilizar um algoritmo guloso que aproxime satisfatoriamente esta decomposição [BO06].

Este método revela potencial para ser implementado na GPU devido à simplicidade dos cálculos e ao facto de o processo de composição dos NFP's poder ser simplificado pelo processo de *rendering*, evitando o cálculo das interseções e sobreposições.

Existe, apesar disso, a questão da precisão, que pode restringir a identificação de detalhes dos NFP como encaixes exatos, que surgem no desenho como pontos ou linhas.

Soma de Minkowski em GPU Uma abordagem recente [LM11] realiza a soma de Minkowski em GPU, recorrendo à *framework CUDA (Compute Unified Device Architecture)* e renderizando diretamente o polígono resultante (recorrendo a *OpenGL - Open Graphics Library*). Este algoritmo realiza somas de formas a 2 ou a 3 dimensões.

Os dois polígonos ou poliedros iniciais são interpretados como uma série de triângulos que formam uma grelha. Os seus vértices são somados, originando um superconjunto dos pontos que constituem a soma de Minkowski. De seguida, os componentes que não são necessários para definir a superfície ou o contorno da soma são descartados, em paralelo (usando um programa em CUDA [KH10]), com base numa série de proposições e os restantes copiados para um *vertex buffer object* e desenhados usando OpenGL.

Numa implementação em GPU, este algoritmo tem a vantagem de não ser necessário transferir os dados (coordenadas dos vértices) para o dispositivo já que estes são calculados no próprio, tirando, adicionalmente, partido do paralelismo.

No entanto, esta estratégia, tendo sido desenvolvida com vista a realizar o planeamento de movimentos (*motion planning*), é orientada a problemas com 3 dimensões e não deteta buracos nos NFP's, o que se revela uma limitação significativa em algoritmos de *nesting* (figura 2.16).

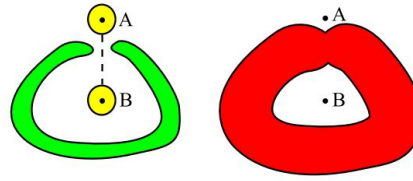


Figura 2.16: O buraco B não é detetado pelo algoritmo. [LM11]

No seguimento desta abordagem, é ainda proposto um algoritmo para determinar os *voxels* (ou *pixels*, se for uma forma a 2D) que definem o contorno externo da soma calculada. Desta forma, desenha-se a superfície correspondente à soma de Minkowski e preenche-se espaço circundante, delimitado pela caixa retangular que a envolve (*bounding box*) usando preenchimento ortogonal (*orthogonal fill*). Para terminar de preencher as concavidades recorre-se a *flood fill*, que, dado um ponto de partida, "espalha" a cor por todas as células vazias adjacentes até encontrar o limite do poliedro. Estas etapas encontram-se ilustradas na imagem 2.17.

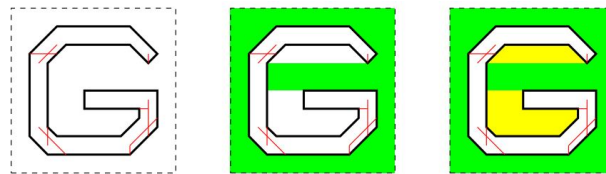


Figura 2.17: Detecção dos limites da soma de Minkowski recorrendo a *orthogonal* e *flood fill*. [LM11]

2.3 Algoritmos construtivos

Os algoritmos que vão construindo o *layout* peça a peça lidam com soluções parciais. A solução final vai-se construindo iterativamente [BO09]. Para determinar a posição da peça a colocar, adotam uma ordem e uma regra de posicionamento.

2.3.1 Regras de posicionamento

A regra de posicionamento define como se vai colocar cada peça. É uma heurística de posicionamento que descodifica uma sequência de peças numa disposição no espaço (figura 2.18).

Bottom-left A regra de posicionamento mais comum é a *bottom-left* (utilizada por [Art96], [DVD02] e [GO02]), que procura a posição horizontalmente mais à esquerda e verticalmente mais

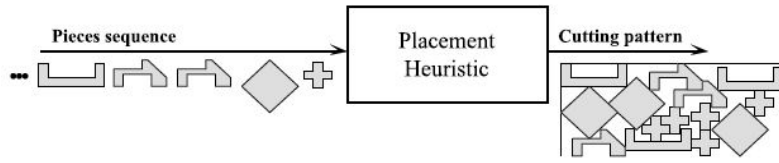


Figura 2.18: A aplicação de uma heurística de posicionamento a uma sequência determina a disposição das peças no espaço. [GO02]

abaixo para a peça a posicionar² (figura 2.19). Pode ser aplicada em outras direções (por exemplo, *top-right*).

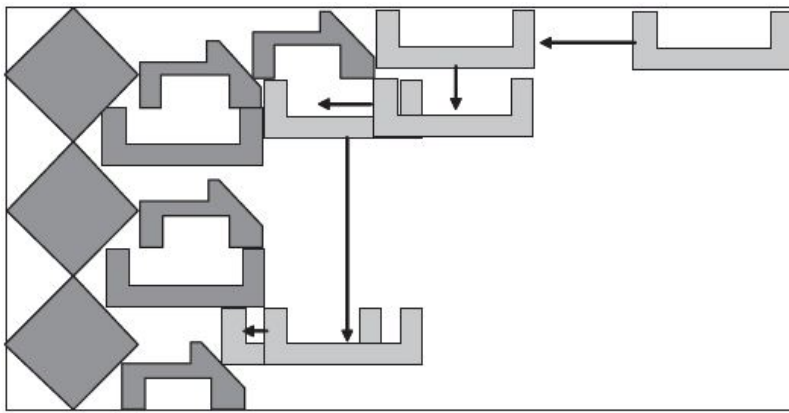


Figura 2.19: Heurística de posicionamento *bottom-left*. [BO09]

Bottom-left com preenchimento de buracos Um buraco é um espaço vazio na superfície rodeado de peças onde podem caber peças mais pequenas. Por vezes é vantajoso começar a procurar espaços vazios desde a esquerda, entre as peças já posicionadas a fim de identificar possíveis posicionamentos nos buracos.

Esta busca pode fazer-se impondo uma grelha sobre o parão de corte discriminando (tal como na representação em grelha) os pontos onde procurar [Seg86] ou efetuando uma busca contínua, recorrendo aos NFP's [BHKW06]. Neste caso, identificam-se todos os possíveis pontos onde colocar o polígono: pontos que não pertençam ao interior de nenhum NFP das peças já colocadas com o polígono a colocar, K , e que, simultaneamente, conduzam a posicionamentos no interior da superfície de posicionamento, i. e., que façam parte do IFP do *layout* (figura 2.20).

$$A = (x, y) : (x, y) \notin interior(NFP_{ik}), \forall i = 1 \dots m \wedge (x, y) \in IFP_k$$

²Primeiro escolhe-se a posição mais à esquerda possível e só depois é que se procura a coordenada mais abaixo, pelo que a regra deveria ser denominada *left-bottom*.

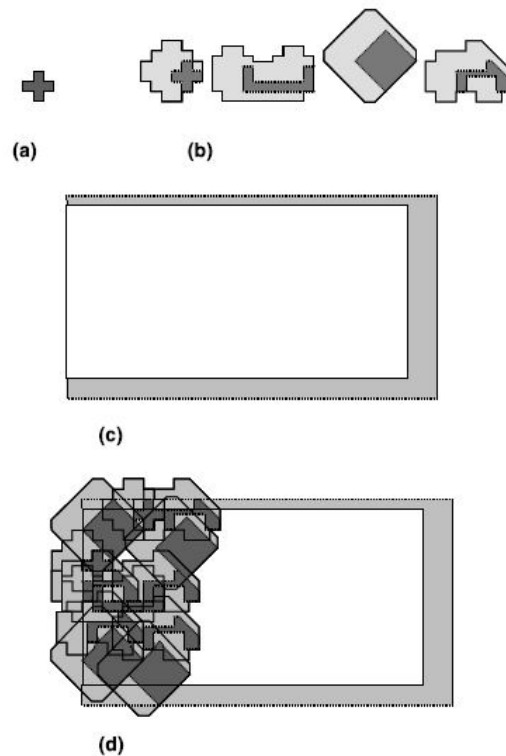


Figura 2.20: Posicionamentos admissíveis num *layout*.

Peça a colocar (a), NFP's das peças do *layout* (b), IFP da superfície (c), regiões admissíveis para o posicionamento (d). [GO02]

Gomes e Oliveira [GO02] reduzem o conjunto de pontos onde é possível colocar K considerando apenas os vértices das regiões livres, isto é, interseções entre os NFP's e entre os NFP's e o IFP.

Dowsland et al [DVD02], para cada aresta de cada NFP, colocam K no vértice mais à esquerda e deslizam K ao longo da aresta até encontrar um ponto que não pertence a nenhum outro NFP.

TOPOS Neste método [OGF00] a posição relativa de cada peça em relação às outras não se modifica, mas o conjunto de polígonos pode mover-se dentro dos limites do espaço de forma a acomodar novas peças, isto é, admite uma origem flutuante. A solução é descrita pelos seus limites externos e os espaços entre as peças são considerados desperdício, não existindo assim possibilidade de colocar peças menores entre elas (figura 2.21).

Em cada iteração há a necessidade de calcular um novo NFP entre o *layout* atual e a nova peça. A colocação de novas peças pode efetuar-se tendo em vista vários objetivos:

- minimizar a área da moldura retangular da nova solução;
- minimizar o comprimento da moldura retangular da nova solução;

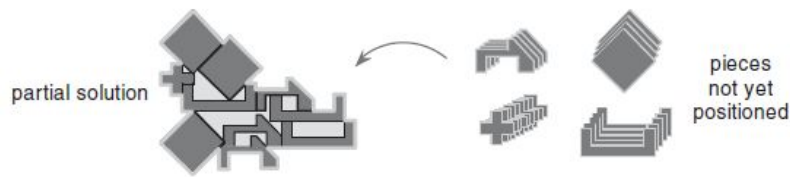


Figura 2.21: Heurística de posicionamento TOPOS (origem-flutuante). [BO09]

- maximizar a sobreposição das molduras retangulares da peça a ser colocada e da solução atual.

Em Bennell e Song [BS10] é acrescentada a possibilidade de preenchimento de buracos ao TOPOS.

2.3.2 Sequências de posicionamento

A sequência de posicionamento define por que ordem vão ser dispostas as peças no *layout*. A sequência de posicionamento indica qual a peça a colocar numa dada iteração do algoritmo. Esta seleção pode ser feita segundo vários critérios, enunciados de seguida.

Seleção Aleatória A peça a colocar pode ser escolhida aleatoriamente, o que é frequente em algoritmos do tipo de Monte Carlo. As soluções geradas são extremamente variáveis e, geralmente, constituem apenas o ponto de partida dos algoritmos que melhoram uma solução completa (2.4).

Seleção Dinâmica Assume-se um dado número de tipos (formas) de peças e várias peças de cada tipo. Para cada tipo de peça considera-se um dado número de rotações discretas (por exemplo, 0° e 180°). Todos os tipos de peça estão disponíveis numa dada iteração desde que haja peças desse tipo por colocar.

A seleção é feita experimentando colocar todos os tipos de peças possíveis numa dada iteração e escolhendo aquela que produz a melhor solução segundo um dado critério (figura 2.22).

Os critérios de seleção (avaliados por [OGF00]) podem ser:

- desperdício relativo;
- sobreposição;
- distância relativa;
- desperdício menos área de peças sobrepostas;
- distância menos área de peças sobrepostas;
- desperdício menos área de peças sobrepostas mais distância.

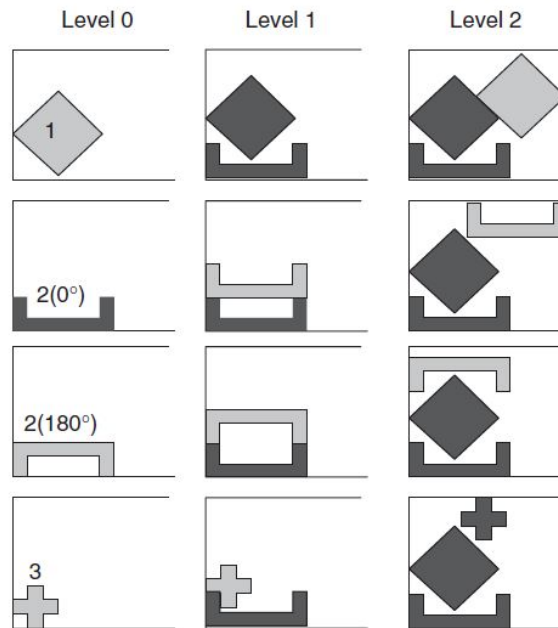


Figura 2.22: Selecção dinâmica da peça a colocar.

Na seleção dinâmica experimenta-se posicionar todos os tipos de peças e escolhe-se o que produzir a melhor solução (a cinzento escuro). Segue-se a mesma lógica em cada iteração (nível). [BO09]

Árvore de pesquisa É possível expandir a seleção dinâmica de peças, construindo uma árvore de pesquisa em que os sucessores de cada nodo correspondem aos tipos de peças que falta colocar. O comprimento da árvore é o número total de peças a colocar. O último nível da árvore possui as soluções finais.

Não é possível, no entanto, construir a árvore na sua totalidade devido ao tamanho do problema. Assim, há que restringir o número de sucessores em cada nível.

Neste algoritmo, Albano e Sapuppo [AS80] admitem a possibilidade de fazer *backtracking* e podam a árvore baseando-se no desperdício (apenas admitem soluções seguintes se o desperdício gerado por estas não for muito maior que o menor obtido até ao momento).

Bennel e Song [BS10] optam por efetuar uma pesquisa em largura, *beam search*, usando duas funções para avaliar a qualidade dos nodos sucessores (uma que avalia a qualidade da solução parcial e outra que realiza uma estimativa com base nas peças que falta colocar).

Heurísticas para o posicionamento em superfícies irregulares Baldacci et. al. [BBGM12] desenvolveram um algoritmo que admite superfícies de corte com outras formas que não rectângulos, com defeitos e outras imperfeições. Esta abordagem destina-se, sobretudo, ao corte de peças em superfícies de couro (cuja forma é necessariamente irregular), mas obteve resultados satisfatórios com tempos de processamento reduzidos para problemas de *strip packing*.

Neste algoritmo é utilizada a representação em grelha. A superfície de corte é aproximada por defeito, isto é, sem ultrapassar o interior dos seus contornos, enquanto que as peças são aproximadas por excesso, incluindo os seus limites (ver figura 2.23).

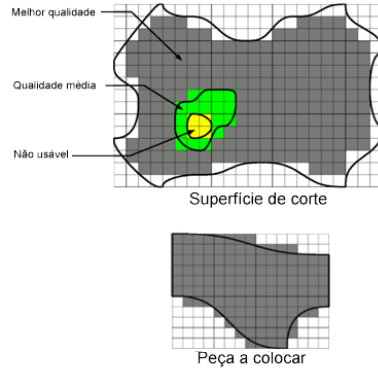


Figura 2.23: Rasterização da superfície de corte por defeito e das peças a colocar por excesso, em [BBGM12].

Dada a natureza do material, existem zonas em que a pele é de pior qualidade. Esta informação é relevante para o posicionamento de determinadas peças e, por isso, a cada *pixel* está associado um valor q que indica a qualidade da pele: se $q=\text{máximo}$ a qualidade é a melhor e se $q=0$ não é possível utilizar aquela pele em nenhuma peça, ou seja, áreas com $q=0$ correspondem a buracos na superfície de corte.

Para aplicar as heurísticas de posicionamento definiu-se uma função $\phi(i, x)$ que mede a vantagem de inserir um polígono i na posição x . Esta função é naturalmente penalizada se não for possível colocar a peça na posição x sem ocorrer sobreposição. Caso o posicionamento seja possível, é descontado o desperdício γ gerado do lado da peça contrário à direção de posicionamento (no caso de *top-left* seria o espaço vazio acima e à esquerda da peça, figura 2.24).

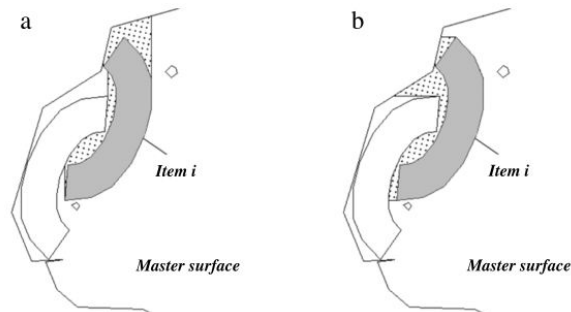


Figura 2.24: Posicionamento de uma peça numa superfície irregular.

$\gamma(i, x, S)$ é a soma dos *pixels* não ocupados à esquerda e acima da peça (se a regra de posicionamento for *top-left*). [BBGM12].

Por outro lado, quanto maior for a densidade α da disposição de peças gerada, maior o número de *pixels* β ocupados por outros polígonos dentro da moldura retangular (*bounding box*) da peça colocada e maior a área δ da própria peça, maior o valor da função ϕ .

$$\phi(i, x, S) = k_1\alpha(i, x, S) + k_2\beta(i, x, S) - k_3\gamma(i, x, S) + k_4\delta(i, x, S)$$

A heurística gulosa utilizada para definir a sequência de peças a colocar é SHN (*Single Knapsack Heuristic*). Segundo esta estratégia, para colocar uma peça, avalia-se todos os posicionamentos possíveis de todas as peças por colocar usando a função ϕ e escolhe-se aquele que tiver o maior valor.

O processo de busca da melhor posição para a peça deste algoritmo implica uma análise do *layout* para todas as posições possíveis. Para evitar uma busca exaustiva, definiram-se 3 **heurísticas de posicionamento**, mais ágeis:

- PH1: *top-left* - escolhe-se a primeira posição possível acima à esquerda;
- PH2: escolhe a melhor posição, entre as possíveis;
- PH3: partindo de um posicionamento inicial, move-se uma peça segundo um conjunto pré-determinado de direções, melhorando o valor de ϕ até atingir um máximo (que pode ser local). Este processo pode ser melhorado usando a pesquisa tabu.

Adicionalmente, desenvolveram-se outras 3 **heurísticas de posicionamento compostas** para escolher as peças a colocar:

- SNH1: gerar um posicionamento para cada polígono com uma das 3 heurísticas de posicionamento anteriores e escolher o melhor;
- SNH2: gerar, iterativamente, *layouts* parciais e completá-los, seguindo os seguintes passos:
 1. obter, para cada polígono, as primeiras k posições com melhores avaliações ϕ ; escolher, entre os posicionamentos admissíveis (de todas as peças) o que gera um *layout* com maior ϕ ;
 2. completar o *layouts* usando uma das 3 heurísticas de posicionamento acima descritas;
 3. comprimir a disposição usando PH3, alternadamente, com uma regra *top-left* ou *bottom-left* (de forma semelhante ao algoritmo de Jostle, figura 2.27), mas sem mudar a posição das peças.
- SNH3: aplicar SNH1 ou SNH2 iterativamente, aumentando a resolução.

Programação em Lógica com Restrições Ribeiro e Carravilla [CR03] recorrem à programação em lógica com restrições para definir o posicionamento das peças e admitem *backtracking* na construção da solução.

2.4 Algoritmos de melhoramento

Os algoritmos de melhoramentos recebem uma solução completa e melhoram-na (alterando a posição das peças, por exemplo), seguindo um processo iterativo. Normalmente, é gerada uma solução admissível mas pouco otimizada de forma aleatória (ver Seleção Aleatória, no início desta secção) à qual são aplicados melhoramentos sucessivos

Os melhoramentos traduzem-se em movimentações das peças no *layout*. É possível efetuar as alterações numa sequência de peças, que pode ser decodificada no *layout* (ver secção 2.3.1), ou diretamente no *layout*.

2.4.1 Movimentos

Os melhoramentos são feitos movendo as peças de lugar. Um movimento pode ser do tipo *swap*, consistindo na troca de duas ou mais peças; ou do tipo *insert*, movendo uma peça de sítio. Pode ainda considerar-se a *rotação* de uma peça de um determinado ângulo (figura 2.25).

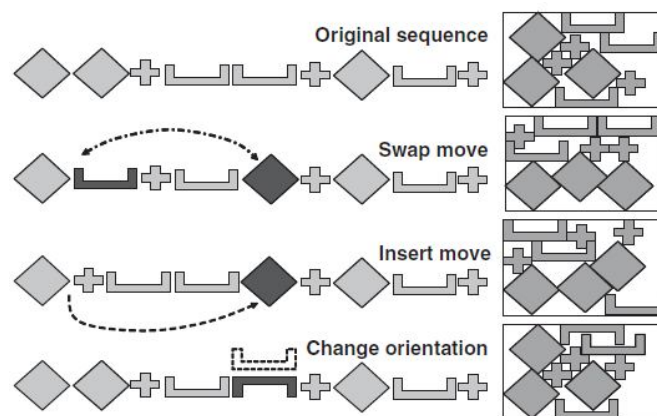


Figura 2.25: Movimentos possíveis das peças numa sequência. [BO09]

Dentro da sequência, trocam-se as peças de posição na sequência, no *layout*, remove-se a peça da sua localização atual e insere-se num novo local.

2.4.2 Alterações em sequências

A solução é representada por uma sequência de peças que é, de seguida, decodificada no *layout* seguindo uma regra de posicionamento (2.3.1).

Decodificar a sequência no *layout* consome tempo de processamento, mas, por outro lado, garante-se que não existem sobreposições entre as peças posicionadas; como tal, todas as soluções geradas são possíveis.

É necessário, no entanto, restringir o número de operações possíveis ou o conjunto de peças a mover. Existem várias regras para aceitar um movimento: pode escolhe-se o primeiro que melhore a solução, o que melhore mais a solução ou um melhoramento aleatório [GO02]. O procedimento

desenvolvido por Gomes e Oliveira apenas admite trocas entre duas peças da sequência, que distem até Δ peças uma da outra (figura 2.26). Como tal, são avaliados os *layouts* gerados pela troca de cada peça com as Δ seguintes, escolhendo-se a disposição com base nos seguintes critérios:

- *first-better* - avança-se com o primeiro *layout* que melhore a solução atual;
- *best* - escolhe-se o melhor *layout* entre todos os analisados;
- *random-better* - escolhe-se uma solução aleatória, entre as soluções geradas que são melhores que a atual.

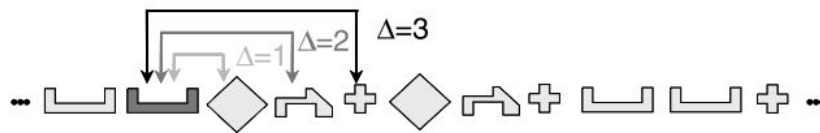


Figura 2.26: Troca de um par de peças, dentro de um intervalo Δ . [GO02]

Bruke et al [BHKW06] utiliza pesquisa tabu e *hill-climbing* para efetuar a busca entre as soluções vizinhas. Em *hill-climbing* apenas aceita a nova solução se esta se verificar melhor do que a atual. A pesquisa tabu, mantém uma lista das últimas 200 soluções visitadas a fim de evitar gerar soluções repetidas e escolhe a melhor solução de uma vizinhança de cinco soluções.

Takahara et al [TKM03] alterna entre dois tipos de pesquisa: normal, em que todas as peças têm a mesma probabilidade de serem movidas, e adaptativo, em que se move as peças que têm mais influência na solução (esta influência é determinada pelo peso da peça, que é incrementado cada vez que uma movimentação da peça gera uma solução melhor).

Jakobs [Jak96] utiliza um algoritmo genético para efetuar a busca. A sequência de peças corresponde ao gene e um cruzamento é feito movendo q peças consecutivas da sua posição atual para o início da sequência. Uma mutação corresponde à rotação de uma peça (de 90°). Babu e Babu [RR01] admite várias superfícies para colocar as peças cuja disposição também está codificada numa sequência que é alterada, à semelhança da sequência de peças, efetuando um cruzamento num certo ponto. Neste algoritmo, uma mutação tanto corresponde à rotação de uma peça (analogamente à implementação anterior) como à troca de peças na sequência (*swap*).

Dowland et al [DVD02] implementam o algoritmo de *Jostle* (figura 2.27). Esta abordagem baseia-se na ideia de que, ao sacudir uma caixa de areia, alisa-se a sua superfície. Da mesma forma, as peças são arrumadas, alternadamente, à esquerda e à direita. A sequência pela qual as peças são dispostas em numa dada iteração é definida pela ordem das suas coordenadas x na iteração anterior. Este algoritmo permite descobrir e preencher buracos e tende a colocar primeiro as peças mais difíceis (isto é, as últimas a serem colocadas na última iteração).

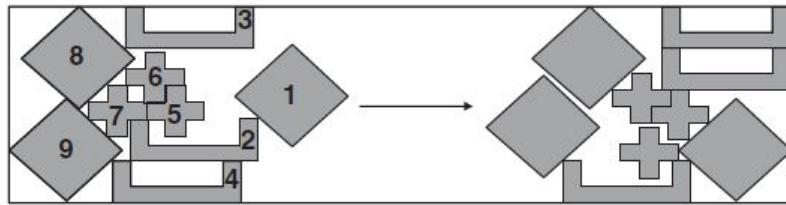


Figura 2.27: Iteração do algoritmo de *Jostle*. [BO09]

2.4.3 Alterações no *layout*

Representar o *layout* diretamente num espaço bidimensional traz certas vantagens, nomeadamente, ao admitir sobreposições entre as peças. Apesar de não serem admissíveis como solução final, as soluções com sobreposições tornam o espaço de busca mais conexo, facilitando a pesquisa. Porém, pode ser complicado, posteriormente, remover as sobreposições.

Para a pesquisa ser possível é necessário reduzir a vizinhança de soluções, que, se não for limitada, é infinita. Para tal, Oliveira e Ferreira [OF93b] recorrem ao algoritmo de arrefecimento simulado, movendo as peças, aleatoriamente, apenas uma unidade da grelha. Admitindo que, numa solução ótima, os polígonos estarão em contacto com outros polígonos ou com os limites da superfície e, à semelhança do método de Gomes e Oliveira [GO02], (secção 2.3.1, *bottom-left* com preenchimento de buracos) restringem-se as posições de posicionamento, não só aos vértices e as intersecções dos NFP's, mas também às projeções dos vértices de NFP's nas arestas de outros NFP's que o intersejam. Efetivamente, o ponto que minimiza a sobreposição do polígono a colocar com o *layout* atual - ponto de sobreposição mínima – encontra-se nesse conjunto de pontos [BHW93] (figura 2.28).

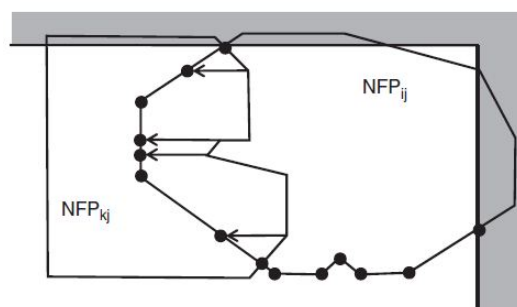


Figura 2.28: Pontos candidatos ao ponto de sobreposição mínima. [BO09]

Blazewicz et al [BHW93] recorre à pesquisa tabu, movendo uma única peça para um buraco ou para o fim do *layout*. No entanto, gerir a lista de buracos é uma tarefa que consome bastante tempo. Alternativamente, em layouts que não sejam discretos, como é o caso de [GA99], permite-se mover as peças numa direção e uma distância aleatórias, limitando a distância máxima que a peça pode avançar.

Sobreposição entre as peças As implementações que admitem um certo grau de sobreposição entre as peças tornam-se problemas multiobjetivo, onde se pretende, simultaneamente, compactar a solução e eliminar a sobreposição. Neste sentido, o grau de sobreposição de uma dada solução é deduzido na função objetivo.

A sobreposição é medida pela distância mínima que é necessário deslocar a peça para a resolver. Como tal implica o cálculo de uma direção oblíqua, geralmente simplifica-se o cálculo fixando a direção como vertical ou horizontal [BO09].

Exemplos de funções objetivo:

- a soma pesada do comprimento do *layout* [HL95];
- a soma da sobreposição entre cada duas peças [HL95];
- a sobreposição ao longo de uma dada direção - vertical ou horizontal [HL95];
- soma da sobreposição e do comprimento do *layout* [BDD01].

É ainda possível separar os dois objetivos, otimizando um de cada vez ([BD99], [IYN07],[ENO07]). Egeblad et al [ENO07] avalia o grau de sobreposição da peça a colocar de forma contínua mas apenas na direção horizontal ou vertical (figura 2.29).

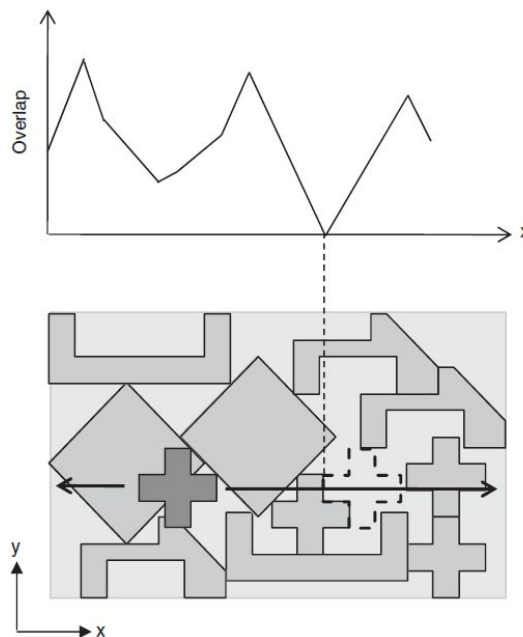


Figura 2.29: Cálculo da sobreposição em Egeblad et al (2007). [BO09]

Programação linear Os métodos que recorrem à programação linear definem como objetivo a minimização do comprimento da solução enquanto as restrições impedem os polígonos de se sobreporem e de saírem fora dos limites da superfície; desta forma, vários polígonos podem mover-se simultaneamente. Geralmente, as restrições são definidas pelas arestas dos NFP's.

Para evitar modelos com variáveis binárias, utiliza-se apenas uma restrição para cada par de peças. Observa-se na figura 2.30 que apenas a restrição indicada pela linha contínua está a ser utilizada. No entanto, se se tratar de um polígono com uma concavidade, será necessário mais do que uma restrição para evitar a sobreposição (figura 2.31).

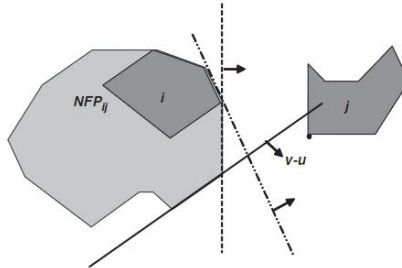


Figura 2.30: Restrições usadas para a colocação de um polígono. [BO09]

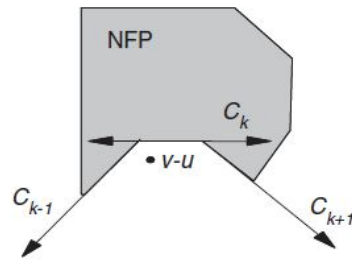


Figura 2.31: Restrições definidas pelas arestas de uma concavidade num problema de programação linear. [BO09]

Existem várias formas de limitar a pesquisa. Milenkovic et al [MDL92] e Bennell e Dowsland [BDD01] limitam a distância que uma peça se pode mover, excluindo, deste modo, várias restrições. Em Bennell e Dowsland a primeira restrição a aplicar é a aresta do NFP mais próxima enquanto que Milenkovic et al a calcula através de interseções e Stoyan et al [SNK96] escolhe a melhor restrição de entre todas as opções. É também possível utilizar a PL para diminuir a sobreposição entre as peças em soluções com sobreposições ([BDD01], [GO06]).

2.5 Resumo

Os algoritmos de *nesting* dividem-se em dois grupos distintos: algoritmos construtivos, que constroem uma solução completa seguindo uma metodologia de distribuição que lhes permita obter, pelo menos, uma disposição melhor do que uma distribuição aleatória; e algoritmos de melhoramentos, que recebem uma solução completa e a melhoram (alterando a posição das peças, por exemplo) seguindo um processo iterativo. A pesquisa de novas soluções tanto se pode fazer no

layout como numa sequência de peças que o codifique. As características dos algoritmos estudados podem ser consultadas nas tabelas que se encontram no final do documento (anexo A).

No que diz respeito à representação da geometria, verificam-se duas técnicas. Uma delas é a representação em grelha do *layout*, em que o valor de cada célula da matriz de *pixels* indica a existência de uma ou mais peças naquele ponto, ou denuncia um espaço vazio. Os valores usados para representar as células vazias ou a existência de uma peça ou de sobreposições variam consoante os autores e os algoritmos implementados.

A outra, utilizada pelos algoritmos mais recentes e mais económica em termos de memória, é a representação poligonal. No entanto, como é computacionalmente pesado usar geometria para calcular sobreposições entre polígonos complexos (com buracos) e inferir a posição relativa entre eles, utilizam-se técnicas que otimizam estes processos. Neste caso, a informação é proporcional ao número de vértices dos polígonos, não dependendo do tamanho das peças.

Nestas implementações recorre-se ao conceito de *no-fit-polygon* para evitar cálculos trigonométricos complexos e repetitivos. Os algoritmos para calcular os NFP's dividem-se em dois grupos distintos: os que recorrem à soma de Minkowski para simplificar o problema e os que simulam o deslizamento de um polígono em torno do outro. Ambas as abordagens têm vantagens e desvantagens. O algoritmo de deslizamento revela-se um processo complexo de aplicar a polígonos com buracos. A soma de Minkowski é eficiente, porém também não é simples extrair o seu contorno para polígonos com concavidades e buracos. Os algoritmos que recorrem à decomposição em polígonos convexos evitam estas dificuldades mas a sua *performance* depende da decomposição efetuada.

Na fase de desenvolvimento desta dissertação, dar-se-á mais ênfase à vertente construtiva dos algoritmos de posicionamento uma vez que se pretende minimizar o tempo de construção de um *layout* (tempo de *nesting*). A representação em grelha do *layout* é a que melhor se adapta aos dispositivos de processamento gráfico e a que se revela mais independente da complexidade da geometria. No que diz respeito ao cálculo dos NFP's, prevê-se ser possível tirar partido dos algoritmos baseados na decomposição.

Capítulo 3

Dispositivos e tecnologias de processamento gráfico

Analisaram-se as capacidades do *hardware* de processamento gráfico (secção 3.1) de forma a identificar os principais benefícios que esta ferramenta traria à resolução de problemas de *nesting*. Estudaram-se, para tal, as tecnologias (secção 3.2) que permitem tirar partido destes dispositivos, nomeadamente *OpenGL* (secção 3.2.1) e *OpenCL* (3.2.2). Adicionalmente, consideraram-se bibliotecas de processamento de imagem (secção 3.2.3) e da geometria (secção 3.2.4).

3.1 Dispositivos de processamento gráfico

Os GPU's têm por objetivo processar e alterar eficientemente informação relativa às imagens a serem apresentadas no ecrã. Trata-se de placas com circuitos que otimizam o processamento gráfico e geométrico, utilizando paralelismo. Atualmente, estes dispositivos permitem, ainda, tirar partido da sua capacidade de processamento paralelo para realizar processamento genérico recorrendo à programação em GPU.

Como tal, os GPU's podem ser utilizados com dois fins distintos: um é o processamento gráfico e outro o processamento paralelo. Para a realização desta dissertação, recorrer-se-á à vertente gráfica deste *hardware* para facilitar o tratamento da geometria.

Para utilizar os recursos de processamento gráfico do GPU recorre-se à interface *OpenGL*. Esta facilita as funcionalidades necessárias para manipular primitivas gráficas e para gerar e transformar imagens. Estas operações são realizadas pela *pipeline* de processamento gráfico do GPU, apresentada de seguida (secção 3.1).

É ainda possível utilizar o GPU como um dispositivo programável adequado a programas que exibam paralelismo ao nível dos dados, visto que segue o modelo SIMD. Para este fim utiliza-se a *framework OpenCL*. Esta vertente da programação em GPU's é referida na secção "Processamento paralelo".

Pipeline de processamento gráfico Os GPU's possuem uma *pipeline* de processamento gráfico que está dividida em várias etapas [OLG⁺05] (figura 3.1). Estas podem divergir consoante as arquiteturas e modelos, porém, a estrutura geral é a seguinte: a primeira etapa processa a geometria, efetuando os cálculos de luz dos vértices, gerando a sua representação no ecrã. A segunda, de rasterização, gera os fragmentos para cada *pixel* e a última calcula, por interpolação, a cor de cada fragmento, podendo, para tal, recorrer às texturas armazenadas em memória. Finalmente, é efetuada a composição dos valores de todos os *pixels*, formando a imagem a apresentar.

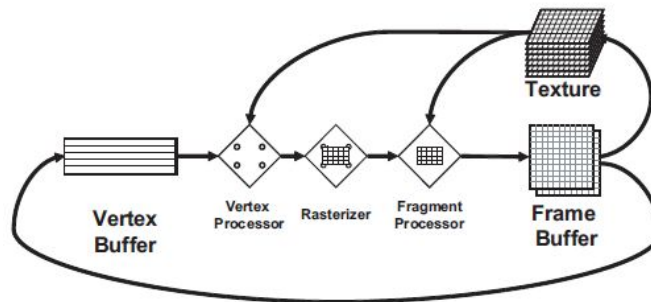


Figura 3.1: *Pipeline* de processamento gráfico. [OLG⁺05]

Processamento Paralelo Os GPU's atuais seguem o modelo de SIMD, isto é, aplicam a mesma série de instruções a um conjunto de dados distintos, por exemplo, a aplicação de uma transformação geométrica a vários pontos 3D. São constituídos por uma série de *streaming multiprocessors* organizados em blocos. Cada multiprocessador possui um dado número de unidades de cálculo (*streaming processors*) que partilham a lógica de controlo e a *cache* de instruções [KH10].

Cada *streaming processor* é capaz de correr, simultaneamente, centenas de *threads*.

Uma aplicação a executar num GPU pode possuir vários programas denominados *kernels*, e que consistem numa grelha de *threads* com o mesmo programa que podem ser executadas simultaneamente, sobre uma matriz de dados.

Quando um determinado conjunto de *threads* (designado *wrap*) tem que aguardar por acessos à memória e outras operações de longa latência, outro *wrap* é escolhido para executar, de forma a camuflar a espera. Este processo dá pelo nome de *latency hiding*. Outra vantagem destes dispositivos é existirem sempre *wraps* prontos a executar, sendo que selecionar um para esse efeito não introduz nenhum tempo de espera – *zero latency thread scheduling*. Esta situação é claramente vantajosa em relação ao lançamento de uma *thread* num CPU introduz um *overhead* que não é possível ignorar já que estes processadores estão otimizados para o processamento sequencial.

Memórias Estes componentes possuem uma memória global (DRAM), à qual podem aceder, também, dispositivos externos. Esta memória armazena, para aplicações gráficas, imagens e informação sobre texturas a desenhar, apresentando uma maior largura de banda do que as memórias comuns, (cerca de 10 vezes mais) apesar de a latência ser maior. Existe ainda a memória constante

que também pode ser lida e escrita por dispositivos externos mas, ao próprio GPU, apenas admite operações de leitura, de forma a evitar modificações.

3.2 Tecnologias de auxílio ao desenho e processamento de imagens

Pretende-se avançar com o desenvolvimento de uma plataforma que permita gerar soluções para problemas de corte que faça uso dos recursos acima enunciados, realizando também algum processamento geométrico e de imagem. Esta plataforma realizará algum pré-processamento da geometria do problema recorrendo à biblioteca CGAL (secção 3.2.4), seguindo-se a aplicação de um algoritmo de posicionamento que realiza o *render* as formas poligonais no GPU usando *OpenGL* (secção 3.2.1), processando as imagens obtidas recorrendo à biblioteca *OpenCV* (secção 3.2.3).

De seguida encontram-se descritas as tecnologias utilizadas, referindo a sua importância na implementação.

3.2.1 OpenGL

OpenGL (Open Graphics Library) é uma API que permite utilizar os recursos fornecidos pelo *hardware* de processamento gráfico [OSW⁺05]. Esta ferramenta permite desenhar não só dados vetoriais (figuras a três dimensões), definidos por um conjunto de vértices, mas também imagens representadas por mapas de *bits*. Define, como tal, dois processos diferentes, um para o tratamento da geometria e outro para o processamento de *arrays* de *pixels*, no qual se enquadra a aplicação de texturas. O processo de rasterização converte os dois tipos de informação no conjunto de *pixels* a ser apresentado no ecrã [OSW⁺05]. Esta API funciona como uma máquina de estados, que permite ativar ou desativar certas funcionalidades do *hardware*, permitindo desenhar e visualizar a imagem de várias formas e com efeitos diferentes.

Trata-se de uma interface que se baseia, essencialmente, na *pipeline* de processamento gráfico referida na introdução. A figura 3.2 detalha a descrição inicial, relacionando-a com as funcionalidades da interface *OpenGL*.

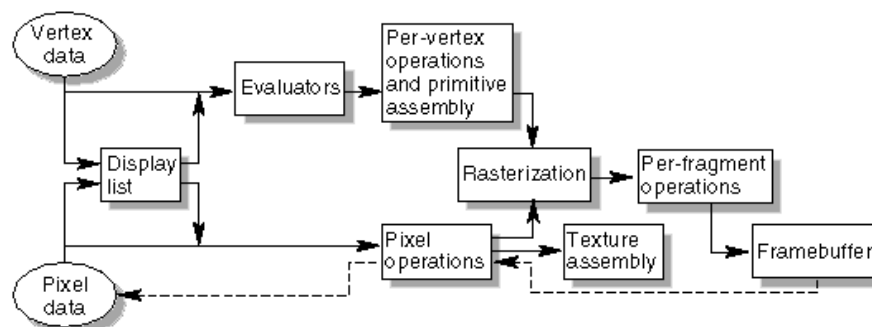


Figura 3.2: Sequência de operações realizadas por um dispositivo gráfico.

Tal como referido anteriormente, existem dois tipos de dados: vetoriais (vértices, linhas, polígonos) e representações constituídas por *pixels* (imagens, texturas). A informação vetorial requer um conjunto de operações de pré-processamento antes de ser apresentada no ecrã. Só são admitidas representações poligonais por isso, no caso da caracterização de curvas ou de superfícies curvas, há que derivar uma aproximação poligonal, determinando os vértices que as definem. Esta tarefa é realizada pelos "Evaluators".

Os dados vetoriais são processados, aplicando-se matrizes de transformação e determinando as suas projeções no plano do ecrã. Adicionalmente, realizam-se cálculos da iluminação, transformação de texturas e materiais. Outra operação a realizar nas etapas de "Per-vertex operations" e "primitive assembly" é a operação de *clipping* que elimina os vértices que não serão visíveis (por se encontrarem fora dos limites do ecrã, por exemplo).

Os dados geométricos pré-processados e eventuais *arrays* de *pixels* contendo texturas ou imagens passam pelo processo de rasterização ("Rasterization") que consiste na conversão dos dois tipos de dados em fragmentos. Para cada fragmento, que designa um *pixel* no *buffer* de desenho (*framebuffer*), são finalmente realizados os cálculos necessários (da cor, *blending*, etc).

Existe um conjunto de conceitos e funcionalidades da interface *OpenGL* - como, por exemplo, as *display lists* - que se revelaram essenciais para a implementação gráfica dos algoritmos de posicionamento e que se encontram detalhados de seguida.

Tessellation A API apenas permite desenhar polígonos convexos. Porém, a maioria das formas dos problemas de corte e empacotamento possui concavidades e buracos que é necessário representar corretamente. A operação *tessellation* consiste em dividir um polígono côncavo, com buracos ou intersecções entre as suas arestas em sub-polígonos convexos (geralmente triângulos) que possam ser desenhados em *OpenGL*. Assim, em vez de enviar as coordenadas diretamente para a *rendering pipeline* do *OpenGL*, estas são remetidas para o *tessellator* que realiza a decomposição em polígonos convexos, efetuando-se o desenho de seguida.

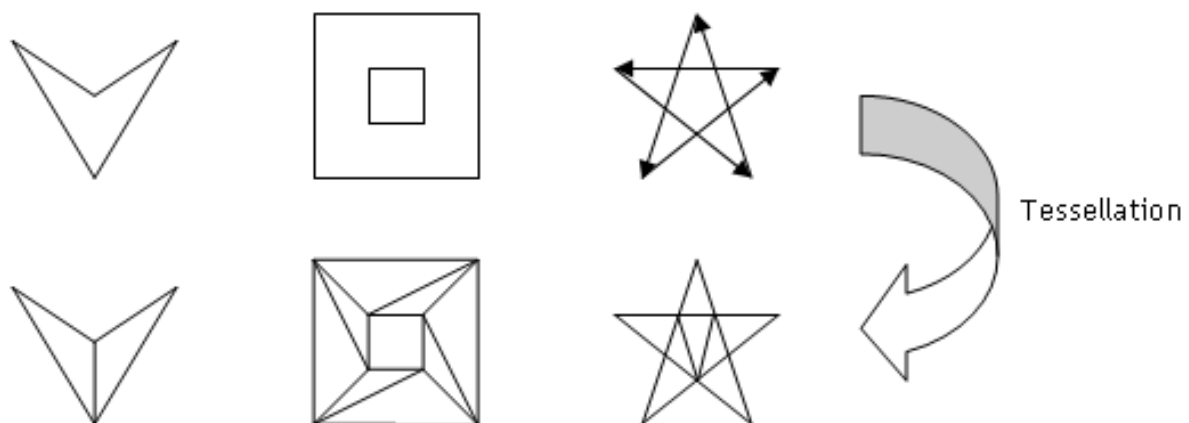


Figura 3.3: *Tessellation* em *OpenGL*.

Display lists As *Display lists* resumem-se a um conjunto de comandos *OpenGL* previamente compilados e prontos para serem executados posteriormente. Tornam o *render* de formas imutáveis muito mais eficiente pois, uma vez que os dados (coordenadas dos vértices, posição e cor) são guardados na memória do dispositivo quando a lista é criada, evita-se a transferência destes dados de cada vez que a peça é desenhada. As listas usar-se-ão para representar tanto os polígonos como os seus NFP's.

Pixel buffer objects Estes objetos, à semelhança das *display lists*, guardam, na memória do GPU, um conjunto de *pixels*, definindo uma imagem e tornando muito mais rápido o acesso a esta informação. Serão usados para guardar as imagens do desperdício originado por um *layout*.

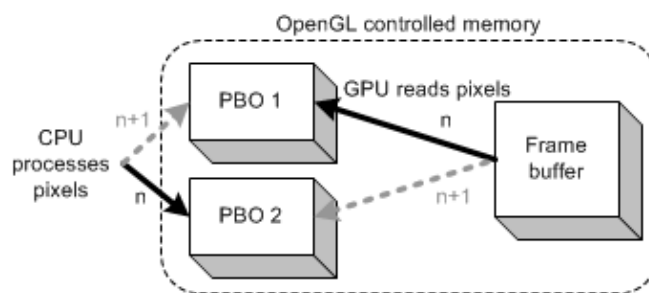


Figura 3.4: Acesso a PBO's em *OpenGL*.

Pipeline de imagem Esta é a *pipeline* que realiza as operações sobre conjuntos de *pixels* (introduzida por "*pixel operations*" na figura 3.2).

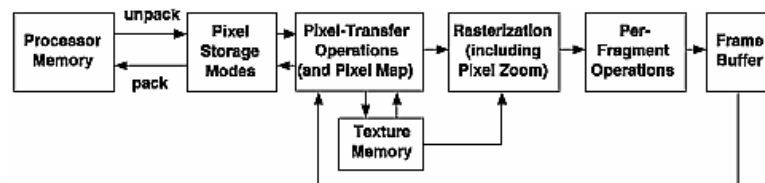


Figura 3.5: Sequência de processamento de um conjunto de *pixels*.

O mais relevante para a plataforma a desenvolver são as operações de transferência de *arrays* de *pixels* do GPU para memória do processador (*pack*) e vice-versa (*unpack*), indicado na figura 3.5. Dependendo do *buffer* a transferir, os valores podem ser inteiros ou *floats*, admitindo representações distintas que dependem do *hardware*. É possível determinar as representações das imagens transferidas de e para a memória do processador, indicando o canal ou canais (se o *buffer* admitir mais que um) a ler/a escrever e a região retangular a copiar.

Este processo deve ser parametrizado de acordo com o *hardware* utilizado de forma a otimizar a transferência.

Accumulation buffer Este *buffer* é utilizado para somar várias imagens, acumulando os valores dos seus *pixels* em todos os canais de cor (RGBA), embora, no contexto desta dissertação se use apenas um. Será usado para acumular os valores do desperdício relativos ao posicionamento de uma peça.

Stencil buffer O *stencil buffer* tem apenas um canal e é utilizado para impedir o desenho em certas áreas do cenário. Assim, quando se efetuar o *render* de uma certa imagem, o espaço que esteja desenhado no *stencil*, vai ser omitido. Este conceito será também usado na determinação do desperdício de uma dada disposição, impedindo a acumulação desperdício na área ocupada por peças.

3.2.2 OpenCL

OpenCL (*Open Computing Language*) é uma *framework* independente da plataforma que permite programar o GPU para a execução de programas que não sejam de natureza gráfica. Deste modo, é possível tirar partido da arquitetura paralela dos GPU's para realizar tarefas em paralelo sobre grandes conjuntos de dados que, se fossem executadas no CPU, tardariam mais. Estes programas denominam-se *kernels* e podem ser executados, simultaneamente, sobre uma matriz de dados. Considera-se a possibilidade de paralelizar determinadas tarefas do algoritmo que calcula os NFP's recorrendo a esta ferramenta.

3.2.3 OpenCV

OpenCV (*Open Source Computer Vision*) é uma biblioteca que fornece um conjunto de funções destinadas ao processamento e análise de imagens e vídeo em tempo real (visão por computador), permitindo, por exemplo, reconhecer objetos ou seguir um dado movimento [Bra00].

As imagens criadas em *OpenGL* podem depois ser processadas usando esta ferramenta. O principal objetivo é ler os NFP's e determinar os seus contornos, delimitando o espaço disponível para a colocação de novas peças. Utilizar-se-ão, para tal, funções de deteção e simplificação de contornos (de uma imagem com apenas um canal de cor). Para avaliar a área ocupada, recorrer-se-á a funções que determinem o *threshold* máximo e contabilizem o número de *pixels* com esse valor.

3.2.4 CGAL

CGAL (*Computational Geometry Algorithms Library*) é uma biblioteca que disponibiliza funções e estruturas dedicadas à representação e ao processamento de elementos geométricos. Revela-se útil na representação de polígonos simples e polígonos com buracos e na rotação das peças. Disponibiliza também funções para o cálculo da soma de Minkowski, que se compararam com as abordagens implementadas para o cálculo dos NFP's. As funcionalidades disponibilizadas por esta biblioteca serão ainda utilizadas para somar pontos e vetores e calcular o invólucro convexo de um conjunto de pontos (a fim de obter a soma de Minkowski de um polígono convexo).

3.3 Resumo

Os GPU's são os dispositivos que se dedicam ao processamento gráfico e geométrico. Possuem múltiplas unidades de processamento simples que operam, em paralelo, sobre um grande conjunto de dados (da imagem). Para utilizar os recursos dos dispositivos gráficos e processar as imagens por eles criadas recorreu-se a um conjunto de tecnologias próprias para o efeito.

O *OpenGL* é uma biblioteca que permite tirar partido do dispositivo de processamento gráfico, disponibilizando as suas funcionalidades sob a forma de uma máquina de estados, em que se pode ativar ou desativar funções e modos de desenho de forma a obter a imagem pretendida. Recorrendo a esta API é possível utilizar a *pipeline* de processamento gráfico para tratar a geometria, convertendo os elementos poligonais numa imagem.

Os modos de desenho disponíveis facilitam ainda muitas das operações executadas pelo algoritmo de posicionamento. *Accumulation buffer* e o *stencil buffer*, por exemplos, podem ser usados para obter as imagens com o desperdício. Conceitos como *display lists* e *pixel buffer objects* permitem tirar partido da memória gráfica e melhorar a performance do algoritmo enquanto que a representação de um polígono complexo é feita recorrendo a *tessellations*.

É possível recorrer às funcionalidades da biblioteca *OpenCV*, dedicada ao processamento de imagem, para analisar as imagens criadas a fim de detetar possíveis posicionamentos. A função de deteção e simplificação de contornos assume particular importância neste contexto.

A biblioteca *CGAL* destina-se ao processamento geométrico, possuindo inúmeras funcionalidades úteis à representação e ao pré-processamento da geometria. Disponibiliza algoritmos para o cálculo da soma de Minkowski que podem ser comparados com as abordagens a implementar.

Este conjunto de tecnologias é fundamental para concretizar a implementação gráfica proposta, simplificando e otimizando as principais operações.

Capítulo 4

Processamento gráfico para posicionamento de formas irregulares

Existe um conjunto de abordagens que utilizam uma representação em grelha para facilitar o processamento geométrico. O trabalho de Baldacci et al [BBGM12] (referido na secção 2.3.2) admite posicionamentos em superfícies irregulares e valoriza a rapidez de uma representação em grelha. Nesta dissertação, também se procurou tirar partido da versatilidade deste tipo de representações aproveitando, adicionalmente, as capacidades dos dispositivos de processamento gráfico para facilitar a geração da imagem do *layout*.

Para explorar o processamento gráfico para o posicionamento de formas irregulares desenvolveu-se uma plataforma-base que disponibilizasse funcionalidades relacionadas com a manipulação de ficheiros, operações gráficas e de cálculo, para além do algoritmo de posicionamento pretendido.

A plataforma desenvolvida é constituída por um conjunto de módulos dedicados à leitura e processamento dos dados relativos ao problema (descritos na secção 4.1), bem como os algoritmos utilizados para a sua resolução (secção 4.2). No que diz respeito aos algoritmos referidos, implementaram-se combinações e variantes das duas principais operações envolvidas na resolução do problema de posicionamento: o cálculo de NFP's (descrito na secção 4.2.1) e o algoritmo de posicionamento (descrito na secção 4.2.2). No algoritmo de posicionamento, distinguem-se dois tipos de heurísticas para a determinação da sequência de peças: estáticas e dinâmicas.

4.1 Plataforma-base

Numa primeira análise dos requisitos do problema, foram identificadas as seguintes necessidades:

- carregamento de ficheiros;
- realização dos cálculos relativos ao pré-processamento de formas geométricas;

- desenho no GPU(*render*) de formas poligonais;
- leitura de memória gráfica;

A plataforma-base foi então desenvolvida para disponibilizar essas funcionalidades. São de seguida apresentados os detalhes de implementação mais relevantes.

Fluxo de dados O diagrama de atividade na figura 4.1 ilustra os passos realizados para gerar uma solução para o problema fornecido. Inicialmente é lida a descrição do problema a partir de um conjunto de ficheiros fornecido pelo utilizador, que especificam quer o problema a resolver quer a forma de o resolver.

De seguida, são realizadas as tarefas necessárias para obter uma representação gráfica (*display list*) das peças e dos seus NFP's em todas as rotações admissíveis; também são geradas as representações dos IFP's da superfície de corte, para cada peça em cada rotação admissível. Este processo implica rodar as peças, decompô-las em polígonos convexos, calcular os NFP's e IFP's e gerar as *display lists*. Note-se que apenas é necessário decompôr as peças em polígonos convexos porque os algoritmos usados para o cálculo dos NFP's se baseiam na decomposição (este processo encontra-se explicado na secção 4.2.1). O passo final é gerar as *display lists* dos polígonos obtidos. Tendo todos os elementos necessários para realizar a representação gráfica, procede-se à execução do algoritmo de posicionamento, que devolve uma solução para o problema. O algoritmo de posicionamento baseia-se no posicionamento de uma sequência de peças. Uma sequência de peças pode ser obtida por regras estáticas ou dinâmicas, enunciadas na secção 4.2.2. Finalmente, é apresentado o *layout* completo.

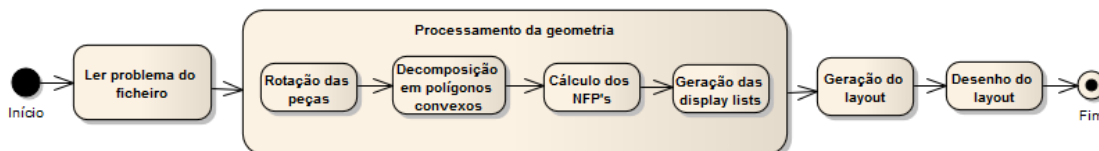


Figura 4.1: Diagrama de atividade da aplicação.

Arquitetura A aplicação está dividida em 3 módulos principais: o de leitura e representação do problema (imagem **a** da figura 4.2) onde são armazenados os vários elementos (peças, superfície de corte, etc...) que o descrevem; o de processamento da geometria (imagem **b** da figura 4.2), onde se encontram os resultados da rotação e decomposição das peças e do cálculo dos NFP's; e o de *render* (imagem **c** da figura 4.2) que possui as estruturas necessárias para desenhar um *layout*.

4.1.1 Leitura e representação do problema

Um problema de *nesting* é definido por um ficheiro de configuração (por exemplo, *swim_config.txt*) que contém os nomes de um segundo e terceiro ficheiros com a lista de peças (*swim.txt*) e a descrição da superfície de corte (*swim_strip.txt*), respetivamente. O ficheiro de configuração indica

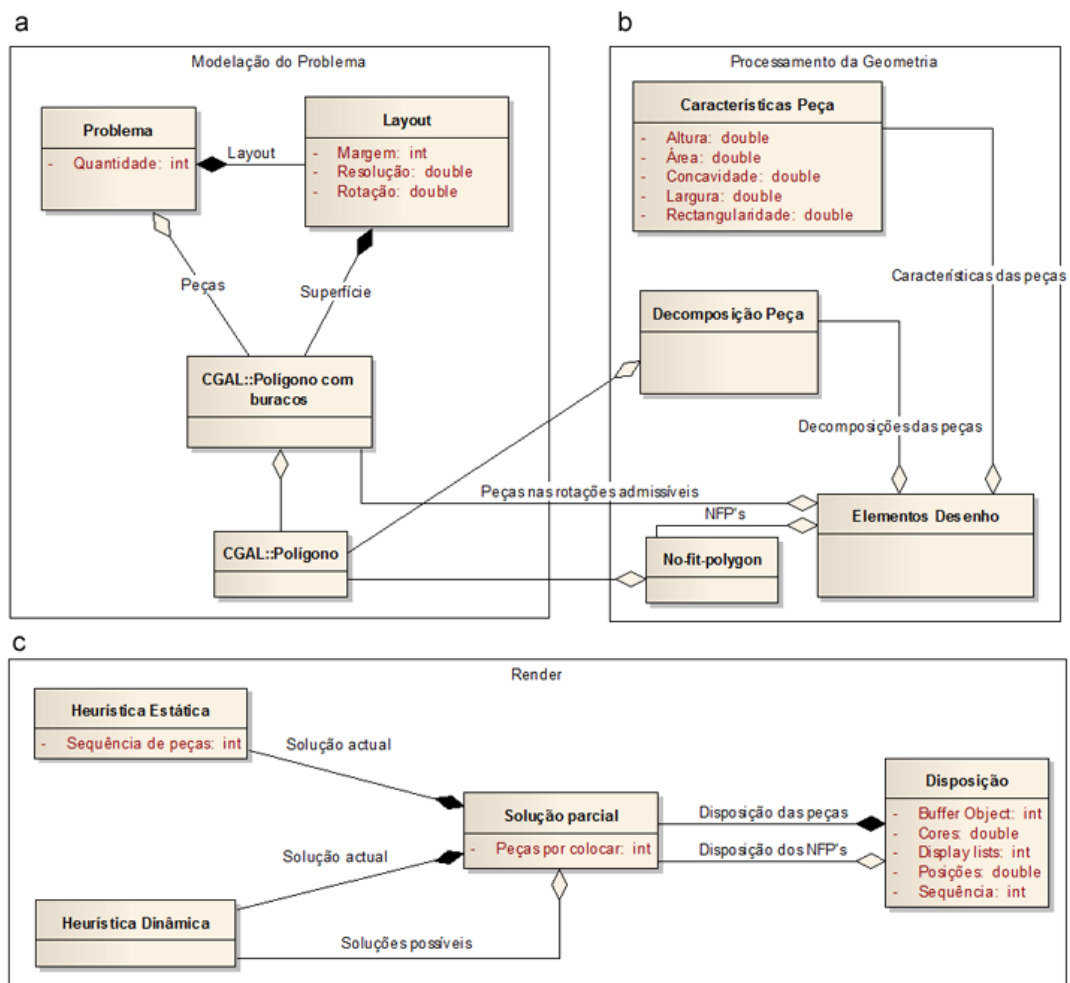


Figura 4.2: Diagrama de classes da aplicação.

a. Modelação do problema. **b.** Elementos da geometria necessários para efetuar o *render*. **c.** Representação gráfica dos padrões de corte.

ainda as rotações admissíveis, a resolução e a margem de separação entre as peças. Estes dois últimos parâmetros não fazem parte da formulação tradicional dos problemas de *nesting*, tendo sido adicionados no âmbito desta dissertação.

A resolução indica o número de *pixels* por unidade de comprimento, determinando o grau de detalhe com que são desenhadas as peças no dispositivo gráfico. A margem de separação indica o espaço a deixar entre as peças colocadas; uma vez que, quando posicionadas na superfície, as peças podem não encostar totalmente, guardando uma estreita margem de separação (definida no ficheiro) umas das outras. A secção 4.2.2 fornece mais detalhes sobre a implementação do algoritmo de posicionamento e os seus parâmetros.

Tanto as peças como a superfície de corte são representadas por polígonos com buracos (usando uma classe da biblioteca CGAL). Um polígono com buracos é definido por uma lista

de componentes: geralmente, o primeiro componente corresponde ao contorno externo, em que os vértices se encontram ordenados no sentido anti-horário, enquanto que os componentes seguintes correspondem aos buracos, com os vértices ordenados no sentido horário. O objetivo desta disposição é saber que a peça se encontra sempre do lado direito dos vetores que definem o seu contorno.

A representação do problema inclui todos os elementos referidos no ficheiro de configuração. A classe "Problema" armazena a lista de peças com a respetiva quantidade. "Layout" inclui, para além do problema, a rotação das peças, a superfície de corte, a resolução e a margem (em *pixels*) de separação entre as peças colocadas (imagem **a** da figura 4.2).

A partir da descrição do problema, obtêm-se os elementos necessários à representação gráfica (classe "Elementos Desenho"). São extraídas e armazenadas (pela classe "Características Peça") as características mais relevantes das peças (altura, largura, área..) que serão utilizadas pelas heurísticas estáticas para ordenar as peças numa sequência de posicionamento. As decomposições dos polígonos (classe "Decomposição") e os NFP's (classe "NFP") são constituídos por um conjunto de polígonos simples (da biblioteca CGAL, imagem **b** da figura 4.2).

4.1.2 Representação da geometria

Esta secção apresenta as estruturas e as funcionalidades necessárias para realizar o *render* da geometria em *OpenGL* o que lhe dá alguma relevância visto que o principal objetivo da dissertação é representar graficamente os *layouts* e tirar partido dessa representação.

A partir da representação poligonal, geram-se as *display lists* de todas as peças em todas as rotações admissíveis. O mesmo é feito para os NFP's e para o IFP's da superfície de corte. As *display lists* das peças são geradas recorrendo a *tessellations*, de forma a permitir a representação de peças e superfícies com concavidades e buracos, enquanto que as *display lists* dos NFP's e IFP's se resumem a um conjunto de polígonos convexos que, combinados, formam o NFP. Estas listas, como referido anteriormente, compilam os comandos *OpenGL* necessários para desenhar cada peça, juntamente com a informação sobre os seus vértices, tornando o *render* mais eficiente.

As heurísticas estáticas e dinâmica (explicadas na secção 4.2.2), necessitam de diferentes tipos de dados, como indicado pela imagem **c** da figura 4.2.

Utilizando uma heurística estática (classe "Heurística Estática"), guarda-se a disposição das peças colocadas (representadas pelas suas *display lists*) e as suas posições na superfície. Para além disso, mantêm-se, enquanto se está a preencher o *layout*, $T \times R$ disposições, cada uma com o conjunto de posicionamentos admissíveis para um determinado tipo de peça numa certa rotação (existindo T tipos de peça e R rotações admissíveis).

Para colocar a peça em questão na rotação determinada, é necessário tornar a imagem do conjunto de posicionamentos admissíveis, criada em *OpenGL*, acessível ao CPU a fim de proceder ao processamento da imagem. Para isso recorreu-se a PBO's (*Pixel Buffer Objects*).

Os PBO's armazenam, na memória gráfica, as imagens geradas. Uma vez associados para leitura, também é possível mapeá-los para a memória do CPU de forma a reduzir a latência de acesso no processamento da imagem.

A alternativa aos PBO's seria forçar a transferência do array de pixels para memória, o que seria mais inconveniente e mais custoso em termos de tempo.

O processo de identificação dos posicionamentos possíveis de uma nova peça no *layout* parcial através da imagem obtida é referido na secção 4.2.2.

Numa dada iteração do algoritmo, a solução parcial construída até ao momento é representada pela classe "Solução Parcial" (na imagem c da figura 4.2). Uma "Solução Parcial" contém a disposição das peças já posicionadas e a configuração dos NFP's (que permite identificar o conjunto de posicionamentos admissíveis) para os tipos de peças disponíveis nas respetivas rotações admissíveis, ambas representadas pela classe "Disposição". Esta classe admite um vetor de *display lists* às quais associa uma posição no espaço e uma cor, fornecendo as informações necessárias para desenhar um conjunto de elementos poligonais.

Para posicionar uma peça recorrendo à heurística dinâmica, experimenta-se colocar, no *layout* parcial obtido até ao momento, todos os T tipos de peças disponíveis em todas as R rotações admissíveis. Assim, origina-se o conjunto de $T \times R$ *layouts* possíveis na próxima iteração e, de entre eles, escolhe-se o que possuir menos desperdício. Tal implica armazenar, para além da solução parcial atual, as $T \times R$ soluções possíveis na iteração seguinte.

O cálculo do desperdício (detalhado na secção 4.2.2.2) é feito avaliando a área da região do *layout* preliminar não ocupada por peças onde já não é possível colocar peças novas. Para definir esta área é necessário acumular as regiões de posicionamentos inviáveis (ocupadas por NFP's e pelo IFP) de todos os tipos de peça possíveis de colocar de seguida. Uma forma de fazer isso é acumular a cor das suas imagens no *accumulation buffer* disponibilizado pelo *OpenGL*. É ainda necessário evitar a acumulação no espaço ocupado pelas peças colocadas, que não é contabilizado como desperdício. Tal é feito recorrendo ao *stencil buffer* que permite definir áreas opacas, onde não é possível desenhar.

À medida que se vai completando a solução, acrescentam-se novas peças ao *layout* parcial, em novas posições, subtraindo os NFP's que lhes correspondem ao conjunto de posicionamentos admissíveis. Estas configurações são rapidamente representadas recorrendo ao GPU.

4.1.3 Leitura e processamento de imagem

O módulo de processamento de imagem recorre, essencialmente, à biblioteca *OpenCV*, disponibilizando um conjunto de funcionalidades que permite ler do dispositivo gráfico uma imagem gerada com recurso a *OpenGL* e manipulá-la de seguida.

O módulo de processamento de imagem recorre, essencialmente, à biblioteca *OpenCV*, disponibilizando um conjunto de funcionalidades que permite manipular uma imagem lida dispositivo gráfico. Esta imagem é binária, isto é, contém apenas um canal de cor cuja gama de valores (inteiros) vai de 0 a 255.

A detecção de contornos é usada para delimitar, numa imagem com os NFP's e o IFP da disposição atual, o espaço vazio onde é possível colocar a nova peça. A região de posicionamentos admissíveis não possui nenhum NFP nem IFP, logo o valor dos seus pixels é 0. As regiões onde o posicionamento é inviável possuem o valor máximo, 255. Desta forma, utilizam-se as funcionalidades da biblioteca *OpenCV* para extrair o contorno que delimita a região de posicionamentos admissíveis, simplificando-o de seguida.

Recorre-se ao algoritmo de Ramer–Douglas–Peucker [Wik12b] para simplificar os contornos obtidos, de forma a eliminar vértices redundantes. Este algoritmo começa por unir os primeiro e último pontos do conjunto dado, definindo uma reta; calculando, de seguida, a distância de todos os restantes pontos a essa reta. Se a maior distância calculada for inferior a um parâmetro ϵ fornecido, a aproximação é suficiente, caso contrário, esse ponto é adicionado ao contorno e repetem-se os testes para os novos segmentos (figura 4.3).

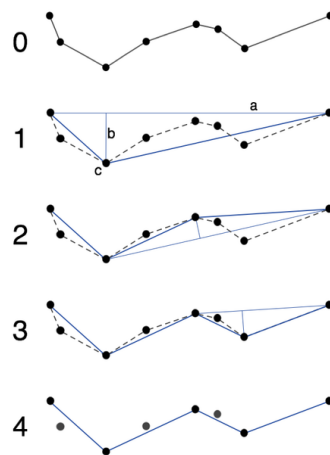


Figura 4.3: Algoritmo de Douglas Peucker. [Wik12b]

Outra funcionalidade importante é a detecção dos valores máximos e mínimos dos *pixels* da imagem, definindo um *threshold*. O *threshold* é um valor limite a partir do qual os *pixels* da imagem são contabilizados ou desprezados. Quando se utiliza a heurística dinâmica, avalia-se uma imagem com a acumulação dos NFP's do *layout* parcial e do IFP da superfície. Uma vez que se pretende apenas a secção onde existe a sobreposição de todas as imagens, é necessário definir um *threshold* que despreze toda a área restante.

A contagem de *pixels* com valores diferentes de 0 é também utilizada na heurística dinâmica para estimar a área correspondente ao desperdício.

Este módulo permite ainda visualizar e gravar as imagens obtidas, registando as várias etapas do processo de posicionamento.

4.2 Implementação de operações para posicionamento de formas irregulares

Para o cálculo dos NFP's usou-se um método distinto. Enquanto que [GO02] usa um algoritmo de deslizamento que não admite a detecção de buracos, a nova implementação disponibiliza dois algoritmos baseados na decomposição (consultar secção 4.2.1 para uma descrição mais detalhada destes algoritmos). Como os NFP's são posteriormente representados graficamente (efetuado o *render* em *OpenGL*), certos passos dos algoritmos baseados na decomposição - como a composição analítica - tornam-se redundantes e desnecessários, como é explicado na secção 4.2.1.

Implementou-se um algoritmo de posicionamento análogo ao de [GO02], que, para colocar uma peça, aplica a regra *bottom-left* a uma lista de pontos que são os vértices que definem a região de posicionamentos admissíveis, isto é, o espaço disponível uma vez subtraídos os NFP's e o IFP (ver figura 2.20 da secção 2.3.1). A diferença é que, enquanto que no algoritmo original é usada uma representação poligonal e a lista é gerada com recurso a cálculos trigonométricos, na implementação realizada representa-se graficamente o *layout* preliminar, com os NFP's e IFP respetivos, usando processamento de imagem para obter a lista de pontos.

4.2.1 Cálculo dos NFP's

Nesta secção são avaliados os algoritmos estudados para o cálculo dos NFP's. Recorreu-se a implementações já existentes - algoritmo de deslizamento [GO02] e soma de Minkowski com decomposição [CGA] - e implementaram-se variantes da soma de Minkowski e do diagrama de declives. Como todos os algoritmos testados - exceto o de deslizamento - recorrem à decomposição de polígonos, na última secção encontram-se enunciadas as estratégias de decomposição utilizadas.

4.2.1.1 Implementações existentes

Algoritmo de deslizamento O algoritmo de deslizamento faz deslizar uma peça B em torno de outra A (mantendo-as sempre em contacto), registando o trajeto do ponto de referência de B. Esse trajeto define os limites do NFP_{AB} . Existem duas versões deste algoritmo: [GO02], que apenas deteta o contorno exterior do NFP e [BHKW06], que identifica também buracos, incluindo encaixes perfeitos (ver figura 2.4, na secção 2.1.2.1).

Soma de Minkowski com decomposição As soluções descritas na literatura efetuam a decomposição dos polígonos em sub-polígonos convexos calculando, de seguida, os NFP's entre estes sub-polígonos e compondo-os posteriormente para formar o NFP das duas peças originais (ver figura 2.15, na secção 2.2). Como referido no capítulo 2, o cálculo dos NFP's pode ser obtido através da soma de *Minkowski*, que é simples de calcular para polígonos convexos.

O NFP dos polígonos A e B corresponde a $A \oplus -B$ (a soma de Minkowski de A com o simétrico de B). Para a obter somam-se todos os pares de pontos de A e B e calcula-se o invólucro convexo (*convex hull*) do conjunto de pontos resultante.

Finalmente, combinam-se todos os NFP's dos sub-polígonos, definindo que secções dos seus contornos constituem o NFP dos polígonos originais. Tal envolve o cálculo de intersecções e outros procedimentos geométricos com maior carga computacional.

4.2.1.2 Implementações adaptadas à representação gráfica

Soma de Minkowski com representação gráfica A implementação em GPU a avaliar é análoga à enunciada acima, porém, como o objetivo final é desenhar os NFP's no *layout*, não existe a necessidade de compôr os NFP's dos sub-polígonos num só; bastando desenhá-los com a mesma cor diretamente no *layout* para obter o mesmo resultado (ver exemplos da figura 4.4). Desta forma, evita-se o passo mais custoso de todo o processo.

Espera-se que estas abordagens, adaptadas representação gráfica, sejam significativamente mais eficientes devido ao facto de em ambos se suprimir a composição dos sub-polígonos.

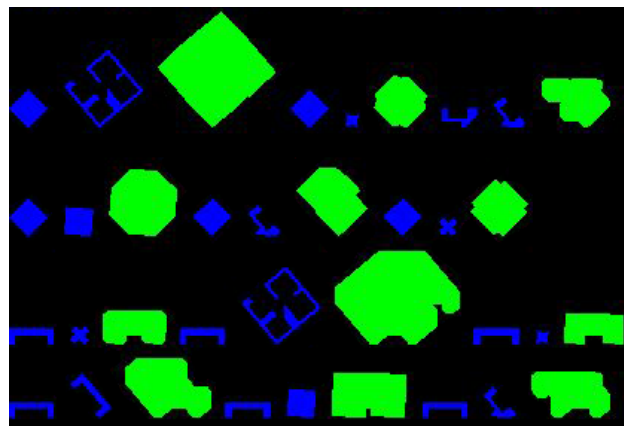


Figura 4.4: Pares de peças e respetivos NFP's, calculados e desenhados recorrendo aos algoritmos implementados.

Diagrama de declives com representação gráfica Nesta variante, também se decompõem os polígonos em sub-polígonos convexos e se desenha os NFP's dos sub-polígonos diretamente no *layout*; o que a distingue da solução anterior é a forma de calcular os NFP's entre peças convexas. Em vez de se realizar a soma de Minkowski entre todos os pontos e obter o invólucro convexo (*convex hull*) do conjunto de pontos resultante, recorre-se ao diagrama de declives enunciado por [CG89] para identificar o NFP das duas peças. Dados dois polígonos convexos *subA* e *subB*, o NFP entre eles pode ser obtido ordenando os vetores formados pelas arestas de *subA* (no sentido anti-horário) e de *subB* (no sentido horário) segundo o seu declive (ver figura 2.14, capítulo 2).

4.2.1.3 Estratégias de decomposição em polígonos convexos

Os três algoritmos baseados na decomposição em polígonos convexos (soma de Minkowski com decomposição e soma de Minkowski/diagrama de declives com representação gráfica) foram testados com os quatro algoritmos disponíveis na biblioteca CGAL para efetuar a decomposição: decomposição Ótima, de Hertel Mehlhron, de Greene e *small side angle bisector*.

A decomposição ótima [Gre83b] recorre a programação dinâmica para obter o menor número possível de sub-polígonos. A sua complexidade temporal é de $O(n^4)$, sendo n o número de vértices do polígono a decompor. O algoritmo de Greene [Gre83b] decompõe a figura em sub-polígonos y -monótonos, isto é, polígonos que não são intersecados em mais de 2 pontos por uma reta horizontal e a sua complexidade temporal é $O(n \log n)$. No pior caso, o número de sub-polígonos é inferior a 4 vezes o ótimo, se bem que na maioria das vezes nunca se chegue a este limite. Hertel Mehlhron [Gre83a] também garante uma decomposição menor que 4 vezes a ótima. Este algoritmo triangulariza o polígono original dividindo-o em subconjuntos de pontos e procurando minimizar o número de sub-polígonos convexos dentro de cada conjunto. Corre em tempo $O(n)$.

O algoritmo *small side angle bisector* [AFH00] baseia-se no método proposto por Chazelle e Dobkin (denominado *angle-bisector* [CD85]). O algoritmo inicial estende a bissetriz dos vértices côncavos (reflexos) do polígono até esta intersecar o seu contorno ou outra bissetriz que já tenha sido estendida anteriormente (figura 4.5). O processo repete-se enquanto existirem sub-polígonos com vértices reflexos.

Small side angle bisector implementa alguns melhoramentos a esta técnica, examinando todos os pares de vértices reflexos do polígono inicial e traçando uma diagonal entre os vértices p_i e p_j de forma a que o número de vértices reflexos entres esses dois pontos seja mínimo. Caso não seja possível ligar 2 vértices côncavos, elimina-se apenas um vértice traçando a diagonal que esteja mais próxima da bissetriz do seu ângulo. Este processo é repetido sucessivamente para os sub-polígonos resultantes até só existirem sub-polígonos convexos. A complexidade temporal deste algoritmo é $O(n^2)$, tendo este revelado-se 50% mais rápido do que o original.

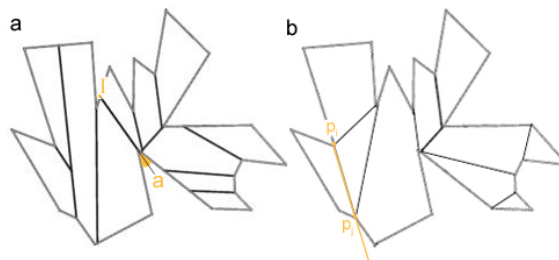


Figura 4.5: Exemplo das decomposições *angle-bisector* [CD85] (a) e *small side angle bisector* [AFH00] (b).

Encontra-se assinalada a bissetriz do vértice a que interseca o polígono original no ponto I , dividindo-o em 2.

4.2.2 Algoritmo de posicionamento

O algoritmo implementado recorre sempre à regra de posicionamento *bottom-left* para colocar uma sequência de peças que pode ser pré-definida ou construída dinamicamente, dependendo da heurística utilizada.

Para colocar a próxima peça (imagem **a** da figura 4.6), desenham-se (recorrendo a *OpenGL*) os NFP's e IFP da peça a colocar com as peças já colocadas nos locais respetivos (imagem **b** da figura 4.6). A imagem obtida representa a área onde a nova peça não pode ser colocada e é, de seguida, lida para memória do CPU. Recorrendo à biblioteca *OpenCV*, identificam-se os contornos da área vazia (imagem **c** da figura 4.6): estes são os pontos onde se pode colocar o ponto de referência da nova peça de forma a que esta fique em contacto com as restantes ou com os limites da superfície de corte, sem ocorrer sobreposição. Visto a regra de posicionamento utilizada ser *bottom-left*, escolhe-se o ponto do contorno mais à esquerda e mais abaixo e o novo posicionamento é adicionado à lista de peças já colocadas. Este processo continua até a lista de peças por colocar se encontrar vazia. A figura 4.6, já referida, ilustra o posicionamento de uma peça do problema *swim*, a partir de uma sequência dada por uma heurística de corte.

Para além dos NFP's das peças é necessário considerar o IFP da peça a colocar com a superfície de corte, para que as peças não ultrapassem os limites desta. O algoritmo de Gomes e Oliveira considera apenas superfícies retangulares, porém, com a representação gráfica, foi possível desenhar os IFP's de superfícies poligonais com defeitos. Considerando a superfície irregular l (representada na imagem **a** da figura 4.7), podemos definir um polígono inverso, i , em que l seja um buraco. Para tal, considera-se uma moldura retangular (*bounding box*) que contenha l (imagem **b** da figura 4.7).

Como os algoritmos que realizam a decomposição não aceitam polígonos com buracos, é necessário simular que existe um contorno único e, para tal, realiza-se a abertura no canto inferior esquerdo. A largura desta abertura é menor do que o número de unidades de comprimento por *pixel*, o que significa que o espaçamento não é visível quando se efetua o *render*.

De seguida são calculados os NFP's do polígono gerado para todos os tipos de peças e respetivas rotações. A imagem **c** da figura 4.7 ilustra o NFP de um losango com o polígono i . O objetivo é colocar o ponto de referência da peça a posicionar num *pixel* que não esteja ocupado por nenhum NFP nem IFP.

Ao colocar as peças, considera-se, entre elas, uma pequena margem que tanto serve para simular a largura da ferramenta de corte, como para evitar que se eliminem posicionamentos viáveis (tais como encaixes perfeitos), entre peças cujos NFP's "encostam", e, por isso, os seus limites não são identificados na representação gráfica. Por outro lado, como se gasta mais espaço a colocar cada peça, alguns posicionamentos deixam de ser possíveis. A figura 4.8 ilustra ambas as situações. Nela estão representadas duas disposições de *shapes* numa superfície poligonal segundo a heurística WIDER: a primeira (imagem **a**) deixa a margem de um *pixel* entre as peças enquanto que a segunda (imagem **b**) coloca as peças encostadas. Comparando ambas as soluções, verifica-se que, se por um lado a margem limita o número de peças que podem ser colocadas em altura na

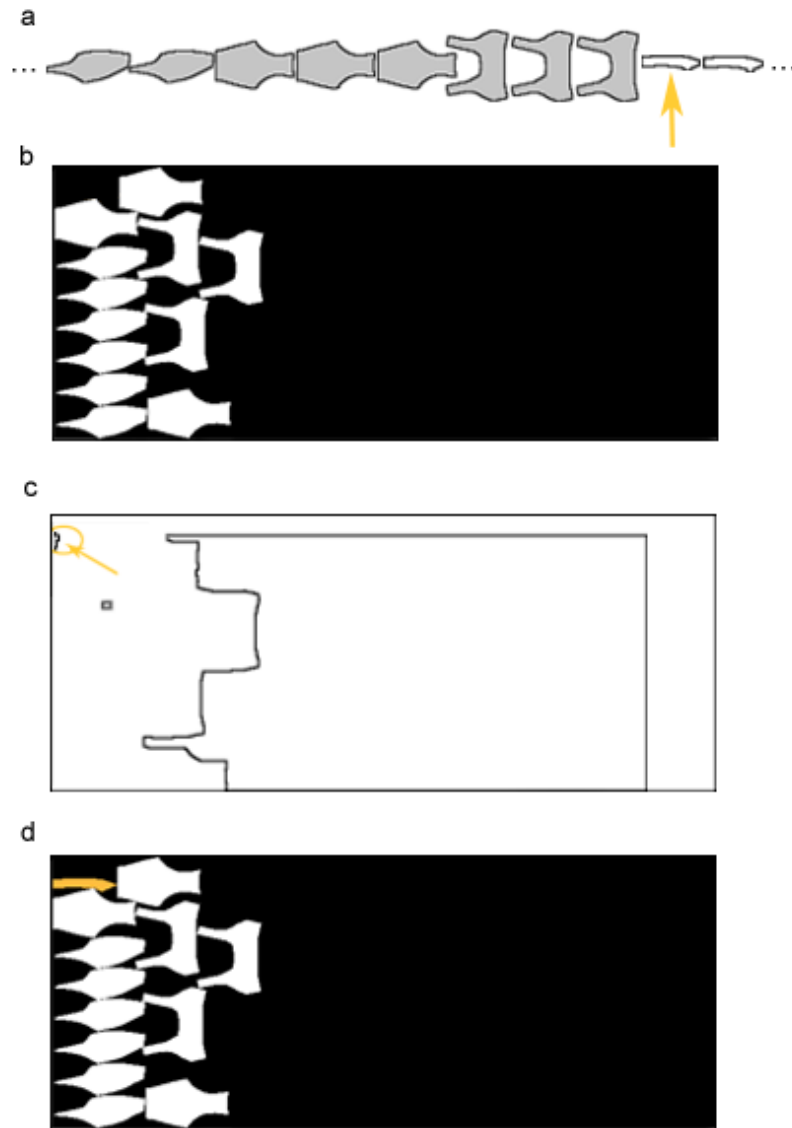


Figura 4.6: Posicionamento de uma peça do problema *swim* no respetivo *layout*, a partir de uma sequência dada pela heurística de corte WIDER.

a. Disposição resultante da iteração anterior. **b.** Sequência de peças (já colocadas a cinzento e por colocar a branco), indicando a peça p a posicionar na iteração ilustrada. **c.** Contornos da área disponível para colocar a peça p . O espaço escolhido para colocar p segundo a regra de posicionamento *bottom-left* encontra-se assinalado pelo círculo. **d.** Disposição com a nova peça (a laranja) já colocada.

região a , por outro identifica os posicionamentos b_1 e b_2 que são encaixes perfeitos.

A margem é definida em *pixels*, por exemplo, se a resolução definida estipular que um *pixel* corresponde a 0.10 unidades de comprimento, os espaçamentos possíveis são 0.0, 0.10, 0.20, etc.

Quanto maior a resolução, menor o espaço desperdiçado pelo espaçamento deixado entre as peças relativamente ao tamanho das mesmas.

A sequência de peças a colocar é definida por uma heurística, estática ou dinâmica.

4.2.2.1 Heurísticas estáticas

A heurística estática é definida previamente à execução por um dos critérios seguidamente enunciados.

- RANDOM - peças ordenadas aleatoriamente;
- HIGHER - peças ordenadas por alturas, as mais altas primeiro;
- MORE_IRREGULAR - peças ordenadas por irregularidade, as mais irregulares primeiro;
- LESS_RETANGULAR - peças ordenadas por retangularidade, as menos retangulares primeiro;
- WIDER - peças ordenadas por largura, as mais largas primeiro;
- LARGER - peças ordenadas por área, as maiores primeiro.

Os critérios MORE_IRREGULAR, LESS_RETANGULAR, LARGER e WIDER, no caso de serem permitidas rotações, adotam, por omissão, a rotação que resulta na menor largura, no caso de *strip packing*. O critério HIGHER dá preferência às rotações que resultam numa menor altura.

4.2.2.2 Heurística dinâmica

A heurística dinâmica implementada avalia todos os posicionamentos possíveis na iteração seguinte, escolhendo o que for mais vantajoso. Nesta variante, não existe uma lista de peças pré-definida. Numa dada iteração, a peça a colocar é aquela cujo posicionamento originar menos desperdício no espaço ocupado. Considera-se desperdício o espaço que não se encontra ocupado por peças e onde não é possível colocar mais nenhuma peça.

Inicialmente, num problema com T tipos de peças, existem T configurações possíveis, uma para cada tipo de peça (ver imagem 2.22, na secção 2.3.2). Se forem permitidas R rotações, o número de configurações possíveis passa a ser $T \times R$. Para decidir a peça a colocar, criam-se T (ou $T \times R$) *layouts* parciais distintos e avalia-se o desperdício de cada um, escolhendo aquele que tiver menor desperdício.

Para avaliar o desperdício de um *layout*, averigua-se quais as peças que poderiam ser colocadas na iteração seguinte, se esta fosse a disposição escolhida. O número de peças distintas que seria possível colocar na segunda iteração seria T (num problema com T tipos de peças que não admitisse rotações), ou $T-1$ no caso da primeira peça colocada ser a única do seu tipo. De seguida, desenham-se os NFP's de cada uma das peças possíveis com as peças do *layout* (obtendo T (ou $T-1$) imagens: imagem **a** da figura 4.9). Acumula-se, de seguida, as imagens dos NFP's (imagem

b da figura 4.9). Quanto maior for o valor de um *pixel* desta imagem, maior é o número de NFP's acumulados naquele ponto, ou seja, o número de peças que é possível colocar lá é mais reduzido. Se, num ponto, estiverem somados os NFP's de todas as T (ou T-1) peças a colocar, esse ponto é considerado um posicionamento inviável pois não existe forma de o ocupar. À imagem obtida, há que subtrair a área dos polígonos já colocados uma vez que estes, apesar de surgirem na imagem dos NFP's como posicionamentos inviáveis, são espaço aproveitado (ver imagem **c** da figura 4.9).

O desperdício resume-se, portanto, aos *pixels* correspondentes a posicionamentos inviáveis que não estão ocupados por nenhuma peça (imagem **d** da figura 4.9). Os restantes *pixels* em que estão presentes alguns NFP's, como já foi referido, não são contabilizados pois ainda podem vir a ser ocupados.

Note-se que, enquanto que nas heurísticas estáticas apenas era necessário fazer o *render* de uma imagem por peça colocada, a heurística dinâmica acima descrita implica, na pior das hipóteses, T*T desenhos por peça colocada que são T imagens dos NFP's por cada uma das T disposições possíveis, pelo que se trata de um método muito mais demorado.

4.3 Resumo

A plataforma desenvolvida recorre à computação gráfica para implementar a estratégia de posicionamento usada por [GO02]. Disponibiliza, para tal, certas funcionalidades base, tais como o carregamento de ficheiros, o pré-processamento de formas poligonais, o *render* destas formas em GPU e técnicas de processamento de imagem. Foram implementadas duas heurísticas de posicionamento construtivas e dois algoritmos para o cálculo dos NFP's baseados na decomposição em polígonos convexos.

A aplicação desenvolvida possui 3 módulos distintos: um dedicado à representação do problema (em que são guardadas as peças, a superfície de corte, entre outras variantes), outro que contém as representações intermédias dos elementos geométricos (tais como a decomposição convexa das peças e os NFP's) e um terceiro dedicado à representação gráfica do problema, incluindo as *display lists* das peças e as suas posições no *layout*. Este último módulo está diretamente associado ao algoritmo de posicionamento e guarda informações que lhe são relativas, como a lista de peças por colocar.

Um problema é fornecido, sob a forma de um ficheiro, à aplicação que o interpreta, extraindo a informação necessária. As peças são, de seguida, processadas de forma a determinar os seus NFP's em todas as rotações admissíveis e o *inner-fit-polygon* da superfície de corte. A biblioteca CGAL, dedicada ao processamento geométrico, é utilizada para executar algumas operações como, por exemplo, a rotação das peças. São finalmente geradas as *display lists* para a representação gráfica dos polígonos.

Segue-se a execução do algoritmo de posicionamento que realiza o *render* do *layout* (usando *OpenGL*) e processa a imagem obtida recorrendo à biblioteca *OpenCV* (dedicada ao processamento de imagens), identificando as posições disponíveis para colocar as novas peças, completando, assim, o *layout*.

Foram implementados dois algoritmos para o cálculo dos NFP's baseados na decomposição em polígonos convexos -soma de Minkowski e diagrama de declives que, posteriormente, tiram partido da representação gráfica. A principal vantagem é ser possível desenhar diretamente os componentes resultantes da decomposição sem existir a necessidade de os compôr, novamente, num só polígono, que é um processo muito demorado.

O algoritmo de posicionamento utiliza dois tipos de heurísticas: estáticas e dinâmicas. Enquanto que nas primeiras a sequência de peças a colocar é pré-definida, na heurística dinâmica a peça a colocar é determinada durante a execução, com base no desperdício gerado.

Uma vez que a implementação gráfica depende menos da geometria, espera-se que a implementação realizada produza soluções rapidamente para problemas complexos com resoluções muito aceitáveis. O seu desempenho dependerá, naturalmente, da resolução utilizada. No caso de resoluções baixas, o facto de se reservar um espaçamento mínimo entre as peças pode produzir um impacto negativo.

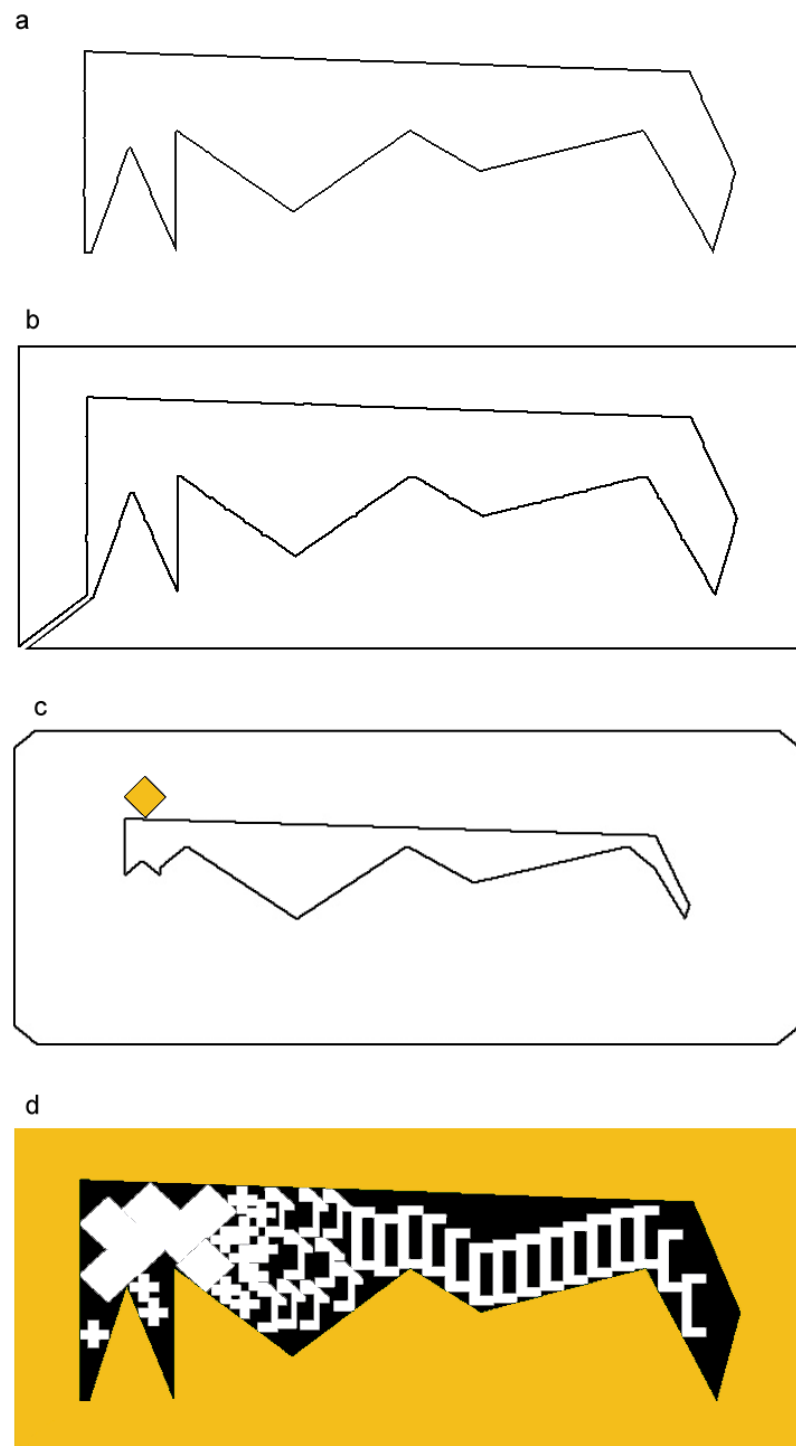


Figura 4.7: Definição do IFP de uma superfície.

a. Superfície irregular l . **b.** Inverso i da superfície original. **c.** NFP de i com uma das peças a colocar. **d.** Disposição do conjunto de peças *shapes* na superfície l .

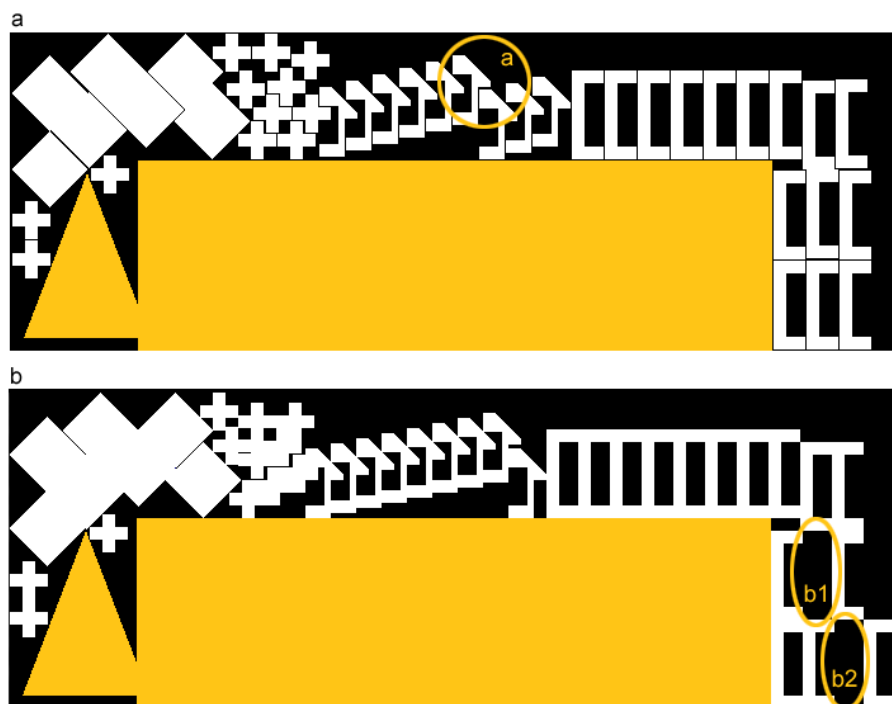


Figura 4.8: Comparação entre uma disposição com a margem de um *pixel* entre as peças (situação **a**) e sem nenhum espaçamento (situação **b**).

A área **a** assinalada na imagem **a** mostra que não é possível colocar tantas peças em altura como em **b**. As áreas b_1 e b_2 indicam posicionamentos possíveis que não são detetados em **b**.

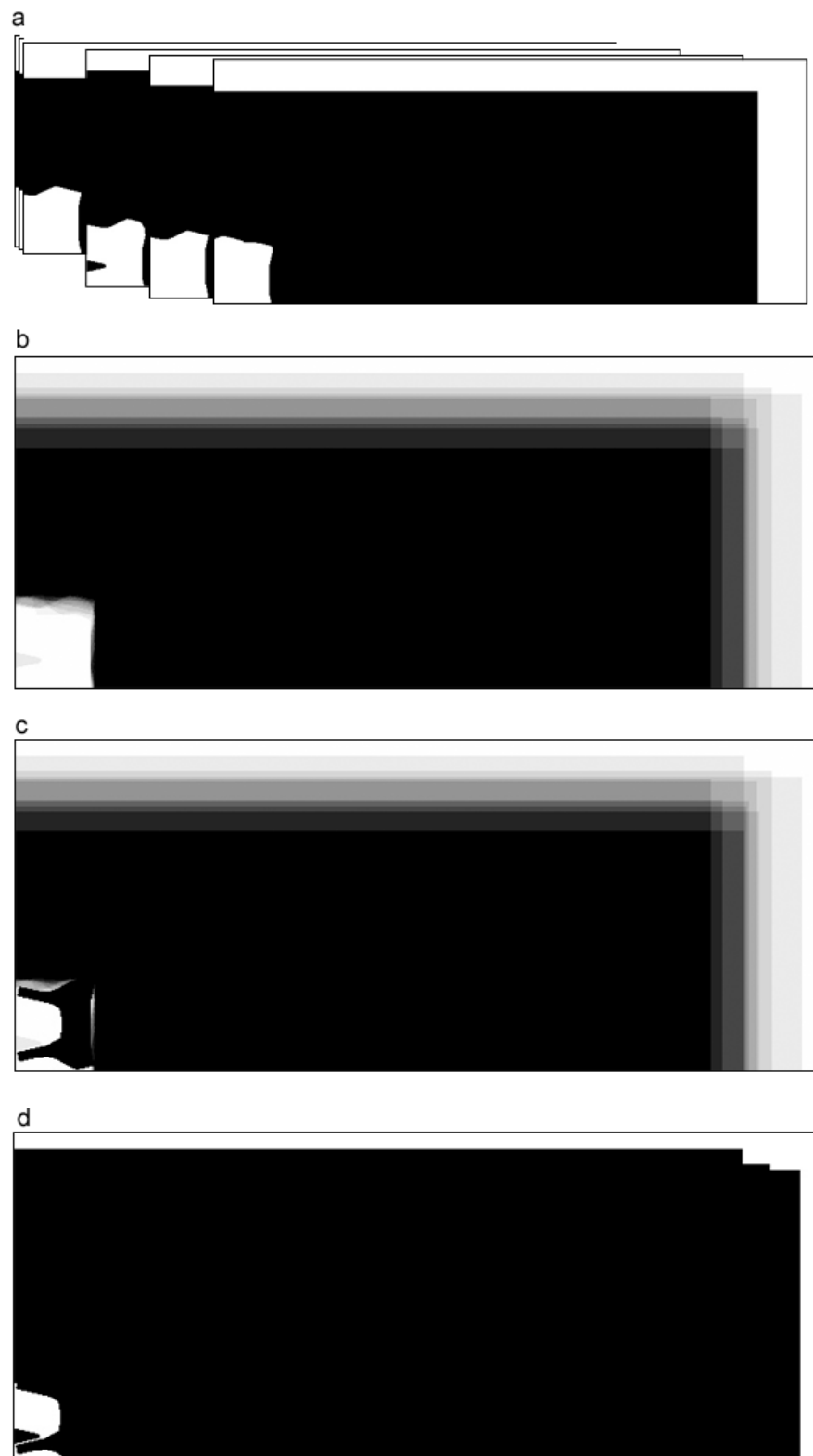


Figura 4.9: Avaliação do desperdício de uma disposição.

a. NFP's de todas as peças que é possível colocar de seguida. **b.** Acumulação de todos os NFP's. **c.** Subtração da área ocupada pela(s) peça(s) colocada(s). **d.** Desperdício: área onde não é possível colocar mais peças e não existe nenhuma peça colocada.

Capítulo 5

Testes e análise dos resultados

A fim de avaliar a implementação realizada e identificar os seus pontos fortes e as suas limitações realizou-se uma série de testes. Os principais objetivos dos testes realizados foram comparar as diferentes implementações realizadas entre elas e identificar as variantes que têm maior impacto no seu desempenho.

Foram também efetuados testes de desempenho de forma a estabelecer uma relação com resultados documentados de alguns algoritmos existentes. Apesar de a comparação efetiva não ser possível, dada a discrepância entre os sistemas reportados, é possível ter alguma sensibilidade relativamente ao desempenho.

São, de seguida, descritos os testes relativos aos algoritmos de cálculo dos NFP's (secção 5.1) e ao posicionamento das peças na superfície de corte (secção 5.2). A análise feita aos algoritmos de posicionamento distingue também as diferenças na eficiência das heurísticas estáticas e dinâmica.

5.1 Cálculo dos NFP's

Todos os algoritmos analisados, à excepção do algoritmo de deslizamento, recorrem à decomposição em polígonos convexos por isso o primeiro módulo de testes corresponde à avaliação das estratégias de decomposição.

De seguida, avaliou-se o desempenho da soma de Minkowski com decomposição [CGA], comparando os resultados com os da soma de Minkowski. Compararam-se ainda, entre elas, as duas implementações (soma de Minkowski e diagrama de declives) com representação gráfica. Finalmente avaliou-se o desempenho destes algoritmos face ao algoritmo de deslizamento.

É feito ainda um comentário às limitações de cada algoritmo (no que diz respeito à identificação de buracos e encaixes perfeitos) e é medido o tempo de *render* dos NFP's.

O tempo de execução foi medido executando 10 vezes o cálculo em questão e calculando o tempo médio dispendido em cada chamada. Os testes foram realizados num processador i5 com 2,27 GHz, salvo os resultados do algoritmo Burke [BHKW06], que foram extraídos da literatura.

Pretende-se avaliar o comportamento dos algoritmos implementados à medida que a complexidade das peças aumenta. Neste sentido são feitas referências a dois conjuntos de peças: *shapes* (figura 5.1), que define quatro peças de contornos simples, mas com algumas concavidades; e *swim* (figura 5.2), constituído por 10 polígonos complexos que pretendem modelar um problema real, representando peças de fatos de banho. Os gráficos apresentados, no entanto, referem-se ao conjunto de formas *swim*, uma vez que se crê ser mais importante visualizar a informação relativa aos problemas mais complexos.

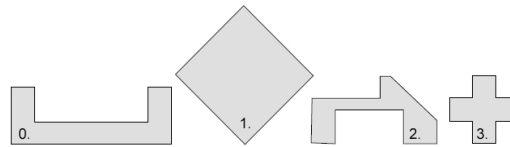


Figura 5.1: Conjunto de peças *shapes*.

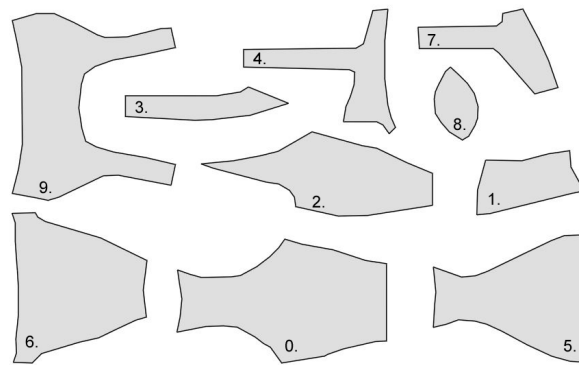


Figura 5.2: Conjunto de peças *swim*.

Avaliação das estratégias de decomposição Mediu-se quanto tempo demora cada um dos 4 algoritmos a decompôr cada polígono (tabelas B.7 e B.8) e o tempo da soma de Minkowski e da realização do diagrama de declives dos sub-polígonos resultantes de cada decomposição (tabelas B.9, B.11, B.10 e B.12), de forma a comparar a implementação realizada com o algoritmo de deslizamento. Verificou-se uma grande disparidade entre o tempo de execução da decomposição ótima e das restantes decomposições (ver descrições das decomposições na secção 4.2.1.3, consultar tempos no gráfico 5.4), pelo que terá demasiado impacto no tempo de execução dos algoritmos para o cálculo dos NFP's com decomposição.

No gráfico 5.3 identifica-se o número de sub-polígonos resultante de cada decomposição, verificando que o número de polígonos da decomposição ótima é sempre menor ou igual ao das restantes. Segue-se *small angle bissector* que é a decomposição não-ótima com menor número de sub-polígonos e finalmente Hertel e Melhorn e Greene, que apresentam resultados idênticos.

Testes e análise dos resultados

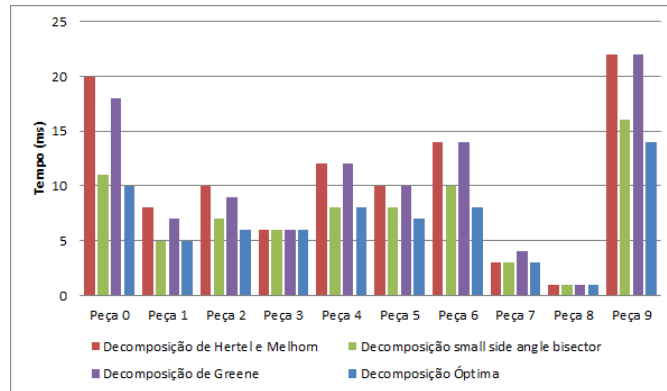


Figura 5.3: Número de polígonos por peça (do problema *swim*) originados para as decomposições ótima, de Hertel e Mehlhorn, de Greene e *small side angle bissector*.

Das decomposições não-ótimas, destaca-se *small side angle bissector* porque, para além de produzir o menor número de polígonos, é também a mais rápida.

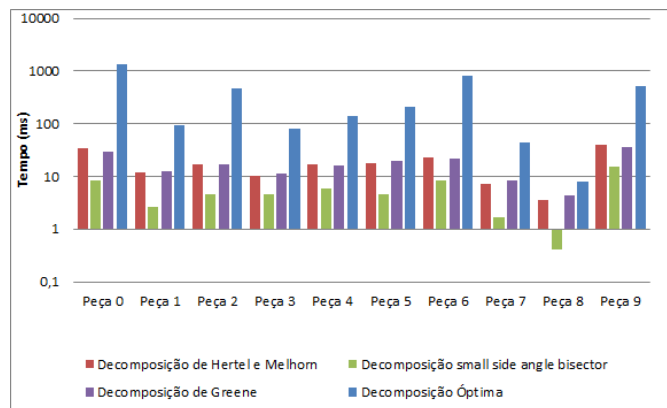


Figura 5.4: Tempo das decomposições ótima, de Hertel e Mehlhorn, de Greene e *small side angle bissector* para cada peça de *swim*.

O tempo médio das decomposições não-ótimas para o problema *swim* foi cerca de 3 a 4 vezes superior ao de *shapes*, o que pode ser explicado pela maior complexidade das peças do segundo problema. No caso da decomposição ótima, essa diferença é muito superior, passando de um tempo médio de decomposição por peça de 14,15 ms (em *shapes*) para 368,68 ms (em *swim*), provando que esta decomposição é muito mais sensível à complexidade das peças do que as restantes. Este fator também provoca uma maior dispersão dos tempos de cálculo dos NFP's para o problema *swim* o que se traduz num desvio padrão mais elevado, chegando a ser superior à média no caso da estratégia de decomposição ótima. Estas nuances nos desempenhos de decomposições distintas e entre problemas de diferentes graus de complexidade são visíveis nos algoritmos de cálculo dos NFP's que recorrem à decomposição. A implementação disponível na biblioteca CGAL é particularmente vulnerável a decomposições e a problemas complexos devido ao processo de

composição realizado no final.

A decomposição efetuada influencia negativamente os algoritmos implementados se o número de sub-polígonos resultantes for muito elevado ou se estes forem complexos em termos do número de vértices/arestas. Como tal, para validar a qualidade da decomposição *small side angle bissector*, mediu-se o tempo de execução da soma de Minkowski e do diagrama de declives para todas as decomposições efetuadas.

NFP_{AB} é o NFP a calcular de duas peças A e B. A decomposição do polígono A tem p_A sub-polígonos, que perfazem um total de v_A vértices; o mesmo sucede para B. A soma de Minkowski entre todos os pares de sub-polígonos de peças distintas ($p_A \times p_B$ pares) resulta, assim, em $v_A \times v_B$ somas de pares de vértices.

No gráfico 5.5 encontra-se o tempo da soma de Minkowski (que se comporta de forma idêntica ao diagrama de declives) sem contar com o da decomposição e em 5.6 podem-se observar esses mesmos tempos acrescidos do tempo médio investido na decomposição por NFP calculado (note-se que a decomposição de cada peça é realizada apenas uma vez; o tempo total é depois dividido pelo número de NFPs a calcular). Verifica-se facilmente que a decomposição ótima é a que resultaria num cálculo mais rápido se não se contabilizasse o tempo total. Das restantes alternativas, a melhor continua a ser *small side angle bissector*, que foi a estratégia eleita para utilizar no algoritmo.

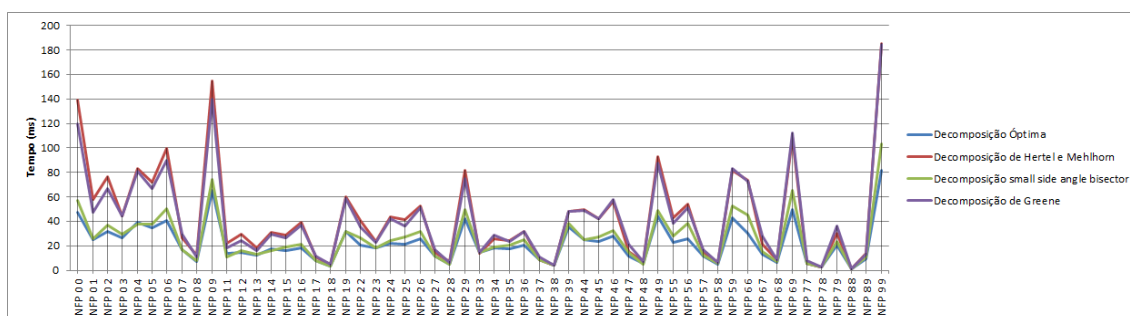


Figura 5.5: Tempo da soma de Minkowski dos sub-polígonos resultantes das 4 decomposições, sem contabilizar o tempo da decomposição. (swim)

Comparação das soluções propostas com a soma de Minkowski com decomposição Inicialmente, mediu-se o tempo que o processo tradicional demorava, contabilizando os três passos já referidos: a **decomposição de ambos os polígonos**, a **soma de Minkowski dos sub-polígonos resultantes** e a **composição dos NFP's** dos sub-polígonos.

Esta abordagem compõe os NFPs dos sub-polígonos num só, sendo que a decomposição usada influencia, de forma muito evidente, o processo de composição. Nos gráficos 5.7 (*shapes*) e 5.8 (*swim*) e nas tabelas B.1 (*shapes*) e B.2 (*swim*) estão indicados os tempos de execução da soma de Minkowski com decomposição disponível na biblioteca CGAL. É possível consultar, nas tabelas B.3 e B.5, o tempo consumido apenas pela **decomposição de ambas as peças e a respetiva soma**,

Testes e análise dos resultados

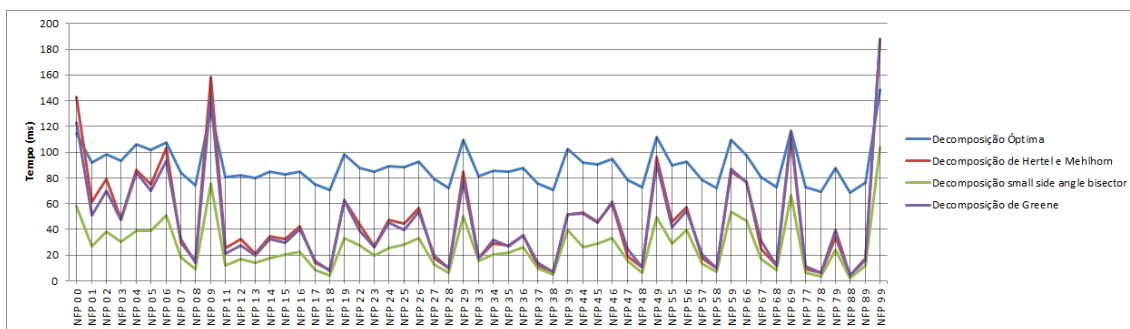


Figura 5.6: Tempo da soma de Minkowski dos sub-polígonos resultantes das 4 decomposições, contabilizando o tempo da decomposição.(*swim*)

que corresponde à implementação da biblioteca CGAL sem o processo de composição; e em B.4 e B.6 o tempo correspondente à decomposição e construção do diagrama de declives ¹.

A soma de Minkowski com decomposição com *Small side angle bisector* permite obter resultados rapidamente para conjuntos de peças simples (*shapes*, tabela B.1 e gráfico 5.7). No entanto, peças mais complexas (*swim*, tabela B.2 e gráfico 5.8) tiram partido do tempo investido na decomposição ótima, visto que esta simplifica significativamente a composição dos NFP's. Verificou-se ainda que, em *shapes*, a decomposição de Greene resultou num cálculo das somas de Minkowski particularmente demorado, nomeadamente para o *NFP*₀₂. Isto deve-se ao facto de a peça 2 ter sido decomposta num número de sub-polígonos muito mais elevado do que o necessário (como se pode constatar na tabela B.7), o que se traduz num aumento do número de vértices a somar e no número de NFP's parciais a compôr. Foi, também, a decomposição cujos tempos da soma de Minkowski resultaram num maior desvio padrão para este conjunto de peças. O mesmo sucede com a decomposição Hertel e Mehlhorn aplicada ao conjunto de formas *swim*, usada na soma de Minkowski. Esta é uma triangularização que produz muitos sub-polígonos no caso de peças mais complexas como as de *swim*, obtendo um pior desempenho e verificando um maior desvio.

Verificou-se que qualquer das estratégias de decomposição não-ótimas é significativamente mais rápida, mas que também resulta num processo de composição substancialmente mais lento, se for realizado analiticamente. Como tal, conclui-se que a decomposição que melhor se adequa às abordagens que recorrem à composição analítica (como é o caso da soma de Minkowski com decomposição, disponível na biblioteca CGAL) depende da complexidade do problema em questão: *small side angle bisector* adequa-se a problemas simples enquanto que a decomposição ótima é mais eficiente para problemas complexos.

No caso da decomposição ótima, o tempo da decomposição e da soma dos sub-polígonos pode ultrapassar os 50% do total enquanto que nas restantes não ultrapassa os 30%. Isto deve-se ao facto da decomposição ótima demorar mais tempo, tornando, simultaneamente, o processo de composição mais rápido.

¹ Este algoritmo não é referido na comparação porque o seu comportamento, em termos de desempenho, é idêntico ao da soma de Minkowski

Testes e análise dos resultados

É possível, no entanto, evitar o último passo - a composição do resultado das somas de Minkowski - através de uma representação gráfica: desenhando os NFP's dos sub-polígonos convexos nos seus locais respetivos obtém-se a representação em grelha do NFP final necessário ao algoritmo sem recorrer a uma composição analítica. Isto permite reduzir para menos de metade o tempo de processamento e obter bons resultados usando decomposições não-ótimas (que são as mais rápidas de efetuar), mesmo na resolução de problemas complexos (ver gráfico 5.6, para o exemplo de *swim*), reduzindo o impacto da complexidade da geometria no cálculo dos NFP's.

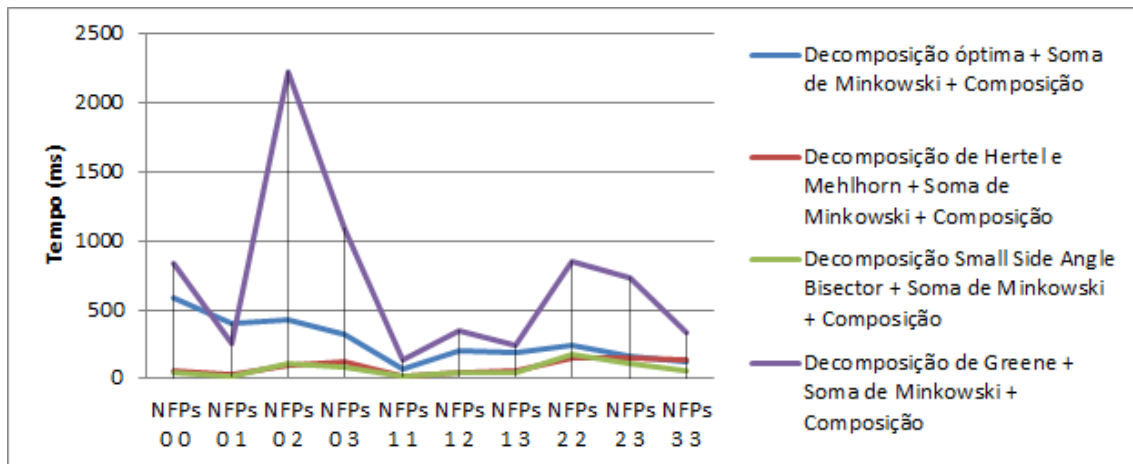


Figura 5.7: Tempo da soma de Minkowski de [CGA] dos sub-polígonos resultantes das 4 decomposições. (*shapes*)

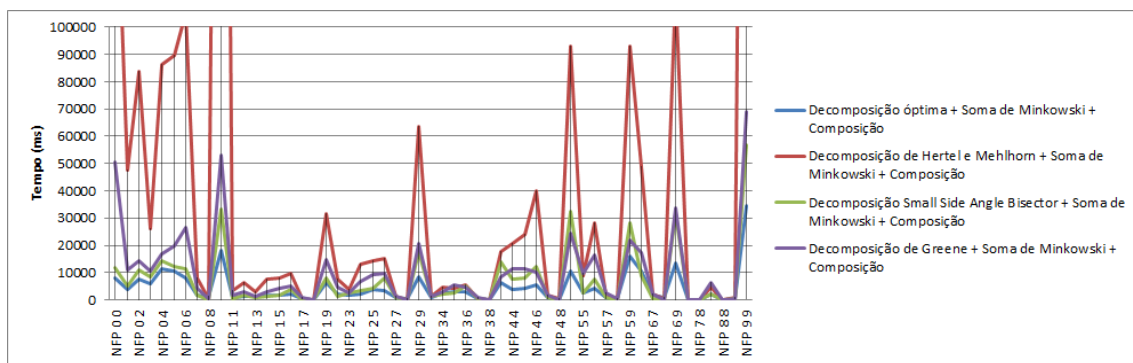


Figura 5.8: Tempo da soma de Minkowski de [CGA] dos sub-polígonos resultantes das 4 decomposições. (*swim*)

Análise das soluções implementadas Seguem-se os resultados dos testes das duas abordagens propostas - a soma de Minkowski e o diagrama de declives - para os problemas *shapes* (tabelas B.9 e B.10) e *swim* e (B.11 e B.12)).

O desempenho dos dois algoritmos foi equivalente, verificando-se uma ligeira melhoria nos tempos do diagrama de declives para decomposições com grande número de vértices. Para realizar esta avaliação usaram-se os resultados da decomposição *small side angle bissetor*, que foi a decomposição não-ótima com melhores resultados.

Avaliou-se o comportamento das soluções implementadas, nomeadamente no que diz respeito à variação do tempo de cálculo de um NFP, à medida que o número de sub-polígonos e o número de vértices resultantes da decomposição aumentavam, constatando que o tempo de cálculo aumentava em ambos os casos de forma proporcional. Tal deve-se à necessidade de, no cálculo de NFP_{AB} , executar o algoritmo em questão (soma de Minkowski ou diagrama de declives) para cada par de sub-polígonos de A e B. Consequentemente, quanto mais restrita for a decomposição em termos de número de sub-polígonos e de vértices, menos processamento existirá.

O gráfico 5.9 ilustra a variação do tempo em função do número de pares de vértices $v_A \times v_B$ enquanto que o gráfico 5.10 representa a dita variação em função do número de pares de polígonos ($p_A \times p_B$) envolvidos no cálculo. Pela observação dos gráficos, verifica-se que o tempo de execução aumenta em função do número de vértices de forma bastante regular² e idêntica para ambos os algoritmos. O aumento em função do número de sub-polígonos é um pouco menos constante.

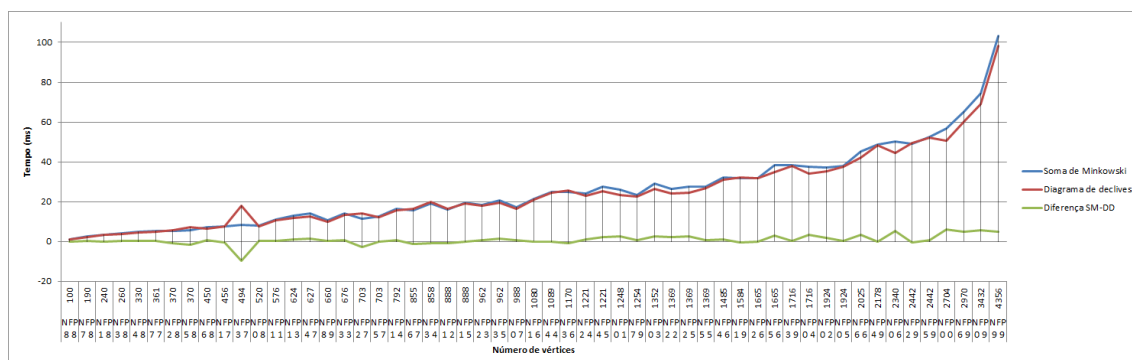


Figura 5.9: Tempo da soma de Minkowski e do diagrama de declives, em função do número de pares de vértices a somar para cada NFP.

Comparação das soluções propostas com o algoritmo de deslizamento Comparou-se o tempo de execução da solução proposta com o do algoritmo de deslizamento [GO02]. Para este efeito, realizou-se a decomposição das peças em pré-processamento e o cálculo das somas de Minkowski posteriormente, medindo quanto demorava cada etapa.

O algoritmo de deslizamento revelou-se bastante eficiente (ver tabelas B.13 e B.14), apesar das limitações no que diz respeito ao tratamento das concavidades e à detecção de buracos. Apenas as decomposições ótima e *small side angle bissetor* permitiram obter, no cálculo dos NFP's, tempos

²Existe uma maior variação no tempo de cálculo do NFP_{37} . Esta variação pode ser explicada, não só pelo processamento inerente à complexidade das peças, mas pela influência momentânea de uma utilização particularmente intensiva do CPU por parte de outro processo durante a execução.

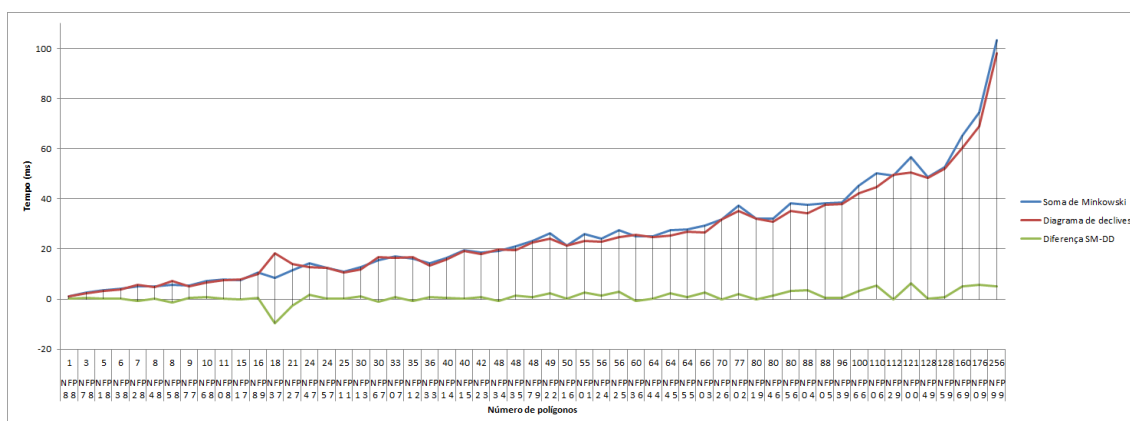


Figura 5.10: Tempo da soma de Minkowski e do diagrama de declives, em função do número de pares de sub-polígonos a processar para cada NFP.

comparáveis aos do algoritmo de deslizamento, todas as outras resultaram num processo mais lento. Porém, contabilizando o tempo da decomposição (gráfico 5.6), apenas *small side angle bissetor* permite que o processo completo seja mais ágil. Contabilizando o tempo médio investido na decomposição por NFP calculado, a diferença entre os tempos médios de cálculo dos NFP's de *swim* das implementações gráficas e do algoritmo de deslizamento foi de cerca 4 ms (de 26 ms para 30 ms, respetivamente). Para o conjunto de peças mais simples o desempenho das duas técnicas foi equivalente.

Naturalmente, é mais demorado processar os NFP's de polígonos mais complexos. Nos algoritmos que recorrem à decomposição, isso traduz-se numa decomposição menos eficiente. Os principais *bottlenecks* do algoritmo de deslizamento são determinar, para cada passo, a aresta sobre a qual deslizar e avaliar a distância que pode ser percorrida dada a existência de concavidades (numa ou em ambas as peças). Neste sentido, averiguou-se a variação dos tempos de cálculo do algoritmo de deslizamento em função do número total de vértices côncavos em ambos os polígonos. A relação observada não é linear porque não tem em conta a interação entre os 2 polígonos, no caso de ambos possuírem concavidades. O gráfico 5.11 ilustra esta situação.

Verificou-se que, para as formas de *shapes*, o algoritmo de deslizamento é equiparável aos outros dois, com algumas variações. A vantagem dos algoritmos implementados relativamente ao de deslizamento é subtil, sendo a diferença mais evidente no diagrama de declives. No caso de formas mais complexas como as de *swim*, a soma de Minkowski ou o diagrama de declives (primeira e última colunas da tabela B.11) resultam numa solução mais rápida do que o algoritmo de deslizamento em 70% a 80% dos casos, com a vantagem de detetarem buracos nos NFP's (gráfico 5.11).

Deve-se ainda considerar a versão do algoritmo de deslizamento de Burke et al [BHKW06] que identifica, para além de buracos, encaixes perfeitos e retas nos NFP's sem a necessidade de definir excepções para processar estes casos. Os resultados dos testes efetuados não são diretamente comparáveis pois foram obtidos num Pentium 4 com 2 GHz.

Testes e análise dos resultados

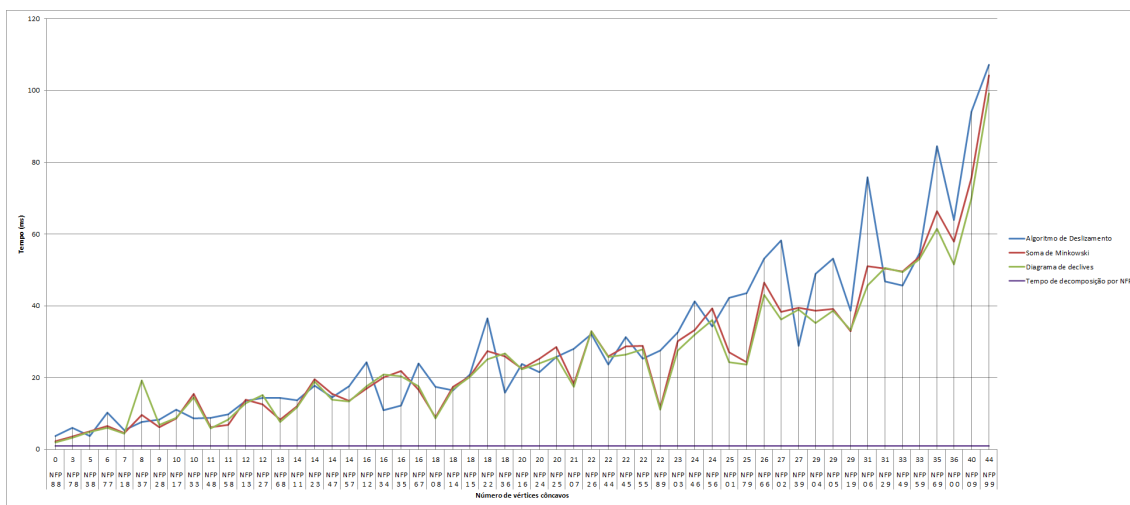


Figura 5.11: Tempos do algoritmo de deslizamento, da soma de Minkowski e do diagrama de declives em função do número de vértices côncavos de cada polígono.

Na tabela 5.1 encontram-se os resultados dos algoritmos implementados, juntamente com o referido em [BHKW06]. Note-se ainda que, nos testes efetuados, se executa a função em questão para cada par de polígonos e respetivas rotações. Por exemplo, para o problema *shapes1*, o número total de tipos de peças é 8 (4 peças com 2 rotações cada) o que se traduz em 64 NFP's; tal significa que o algoritmo a ser testado (soma de Minkowski ou diagrama de declives) é executado 64 vezes. Este pode ser um motivo pelo qual se obteve uma performance significativamente melhor à de [BHKW06] em *shapes0* e tempos mais elevados nos restantes problemas, onde se introduzem rotações.

De facto, é possível otimizar o número de NFP's que é necessário calcular para um dado problema. O NFP_{BA} é equivalente ao NFP_{AB} rodado 180° , o que nos permite calcular apenas metade dos NFP's, rodando-os para obter os restantes. Para além disso, quando se admitem rotações, a forma do NFP apenas diverge se a rotação relativa entre os dois polígonos variar. Por exemplo, o $NFP_{A'B'}$, representando A' e B' os polígonos A e B numa dada rotação admissível r , é equivalente ao NFP_{AB} rodado r^0 pois a rotação relativa entre os dois polígonos é nula. Por outro lado, para as duas formas irregulares A e B, o $NFP_{AB'}$ (com uma rotação relativa de r^0 , estando A rodado 0° e B' r^0) possui uma forma distinta do NFP_{AB} (com uma rotação relativa de 0°).

Estas otimizações reduziriam para menos de metade o número de NFP's a calcular em problemas com rotações, pelo que se estima que os resultados produzidos assemelhar-se-iam mais aos de Burke et al, nesses casos.

Deteção de buracos e encaixes perfeitos O algoritmo de deslizamento não deteta nem buracos nem encaixes perfeitos, mas obtém os contornos dos NFP's com um rigor considerável. Tanto a soma de Minkowski com decomposição (disponível na biblioteca CGAL) como a soma de Minkowski com representação gráfica implementada detetam buracos nos NFP's. Porém, a segunda

Testes e análise dos resultados

Conjunto de peças	Número de tipos de peças	Rotação	Número de rotações por peça	Número total de peças	Número total de NFP's	Tempo de cálculo de Burke [BHKW06]	Tempo de cálculo soma de Minkowski	Tempo de cálculo diagrama de declives
shapes	4	90	4	16	256	0,38	1,7008	1,0422
shapes0	4	0	1	4	16	0,11	0,0636	0,0599
shapes1	4	180	2	8	64	0,19	0,3873	0,266
swim	10	180	2	20	400	6,8	13,7901	10,0478

Tabela 5.1: Tempos (em segundos) de cálculo dos NFPs dos conjuntos de peças indicados do algoritmo de deslizamento de [BHKW06], soma de Minkowski e diagrama de declives.

solução apenas permite identificar os buracos na representação gráfica, visto que são desenhados os vários NFP's dos sub-polígonos resultantes da decomposição das duas peças iniciais, enquanto que na primeira se efetua a composição dos NFP's dos sub-polígonos, obtendo uma representação vetorial suficientemente rigorosa do NFP das peças em questão.

Desenho dos polígonos Avaliou-se, por último, o tempo dispendido no desenho (geração das *display lists*) dos NFP's em duas situações distintas: na primeira os NFP's são polígonos com buracos resultantes da soma de Minkowski com decomposição do CGAL e na segunda um NFP é constituído por um conjunto de polígonos convexos (os NFP's dos sub-polígonos resultantes da decomposição). Para desenhar os NFP's com concavidades e buracos realizou-se uma *tessellation* enquanto que para fazer o *render* dos polígonos convexos se utilizou o modo GL_POLYGON, em *OpenGL*.

Considera-se que o desenho dos NFP's diz respeito ao processo de atualização do estado do *layout* e, como tal, também se mediu o tempo da geração das *display lists* em ambos os casos, de forma a decidir qual das representações (um polígono com buracos ou um conjunto de polígonos convexos) é a mais conveniente.

Nas tabelas seguintes (B.15, B.16) ilustram-se os tempos de desenho de cada NFP. No desenho dos polígonos convexos, é indicado o tempo de desenho do conjunto de polígonos resultante de cada uma das decomposições, já que os sub-polígonos gerados por cada uma são distintos.

As tabelas B.17 e B.18 indicam o tempo da geração das *display lists* para a criação de *tessellations* dos NFP's.

Verificou-se que, para um exemplo com formas simples (*shapes*), é significativamente mais eficiente desenhar um conjunto de NFP's convexos do que realizar uma *tessellation* de um polígono irregular. Porém, quando o número de polígonos convexos a desenhar é elevado, como em *swim*, essa vantagem já não é evidente; chegando, em alguns casos a ser mais rápida a *tessellation*.

Testes *OpenCL*

A soma de Minkowski implica efetuar um grande número de somas, originando um conjunto de pontos do qual se obtém, posteriormente, a *convex hull*. A fim de reduzir o tempo das somas, considerou-se a hipótese de paralelizar, recorrendo a *OpenCL*, esta primeira parte do algoritmo.

Para tal, realizou-se um programa simples (*kernel*) que soma dois conjuntos de pontos. Por uma questão de simplicidade, considerou-se que os *arrays* de pontos eram da mesma dimensão. Desta forma, executaram-se 160 000 (400×400), 360 000 (600×600), 640 000 (800×800) e 1 000 000 (1000×1000) de somas, em blocos de trabalho de 10×10 *threads*, onde cada *thread* corresponde a uma soma.

Posteriormente, testaram-se blocos de 5000×5000 e 10000×10000 somas a fim de verificar o comportamento do programa paralelo cujo tempo de execução também aumenta, mas de forma muito menos evidente do que no processamento sequencial, que depende do aproveitamento dos recursos do *hardware*.

Na tabela seguinte (5.2), encontra-se a duração de cada etapa da implementação em *OpenCL*: invocação do *kernel* com os argumentos pretendidos e cópia dos resultados, do GPU para memória. O processo de inicialização do ambiente em *OpenCL* e compilação do programa não são contabilizados, pois apenas se executam uma vez.

Soma sequencial		Implementação em <i>OpenCL</i>		
		Execução do Kernel	Leitura dos resultados	Total
400*400	3,1	3	6	9
600*600	6,2	3	8	11
800*800	10,7	3	11	14
1000*1000	15,8	3	15	18
5000*5000	332	4	288	292
10000*10000	1359	8	-	-

Tabela 5.2: Tempo (em milissegundos) da execução de $n \times n$ somas em *OpenCl* (passagem dos argumentos, execução do *kernel* e leitura dos resultados), comparado com uma implementação sequencial.

Verificou-se que a execução do *kernel* ronda invariavelmente os 3 milissegundos independentemente do número de somas a efetuar. Por outro lado, a leitura dos resultados e a transferência dos mesmos da memória do GPU para a memória do computador aumenta proporcionalmente ao número de somas como se pode constatar na tabela 5.2. É mais rápido efetuar a soma sequencial do que executar o *kernel* para menos de 160 000 (400×400) somas para igualar o tempo de execução do *kernel* (3 segundos). A cópia dos resultados do GPU para memória é a parte mais demorada de todo o processo e faz com que esta abordagem não seja viável. No caso de se considerar um algoritmo que execute totalmente em GPU como [LM11], desenhar-se-ia diretamente o NFP, evitando-se a transferência de dados. Visto que o método utilizado calcula, seguidamente, a *convex hull* em CPU, não tira grande partido da soma.

Contabilizou-se ainda o número de somas efetuadas no cálculo dos NFP's dos problemas *swim* e *shapes* (não considerando rotações e contabilizando apenas os 10 e 55 casos, indicados nos gráficos acima, dos problemas *shapes* e *swim*, respetivamente). Constata-se que, para os dois problemas analisados, o número de somas é inferior ao mínimo que torna o recurso ao paralelismo considerável (160 000, como mencionado acima). Para problemas com um grande número de tipos de peças (e de vértices) uma implementação feita totalmente em GPU, que permitisse desenhar diretamente os resultados, seria vantajosa.

	Ótima	Hertel e Mehlhorn	Greene	<i>Small side angle bisetor</i>
swim	62715	94981	92159	68213
shapes	1553	1829	2423	1553

Tabela 5.3: Número de somas de pares de pontos efetuadas por cada decomposição (uma soma corresponde à adição das coordenadas x e y de dois pontos de sub-polígonos de peças distintas).

5.2 Algoritmo de posicionamento

Testou-se um conjunto de problemas, com peças de formatos e dimensões distintas. Verificou-se a necessidade de, para problemas de maiores dimensões, diminuir a resolução de forma a que o desenho do *layout* fosse comportável.

Foram feitos testes no sentido de avaliar a viabilidade da implementação para *strip packing*. Neste tipo de problemas o objetivo é minimizar o comprimento de tecido necessário para cortar um dado conjunto de formas. O espaço deixado entre as peças é considerado desperdício.

Realizaram-se testes para 7 problemas diferentes (extraídos do site do ESICUP - EURO Special Interest Group on Cutting and Packing³) com todas as heurísticas. Repetiram-se os testes das heurísticas estáticas com o dobro da resolução, esperando obter melhores aproveitamentos com uma representação mais pormenorizada.

Deve-se salientar que nos tempos de execução do algoritmo de posicionamento não é contabilizado o tempo de cálculo dos NFP's, sendo este referido na secção anterior.

Aproveitamento A heurística que obteve melhores resultados para este tipo de problemas foi a que prioriza as peças maiores (LARGER). O aproveitamento médio desta heurística foi de 69%, obtendo a disposição mais compacta, relativamente às originadas pelas restantes heurísticas, para 4 dos 7 problemas testados.

A heurística dinâmica foi a que resultou em *layouts* mais compactos a seguir a LARGER, ocupando uma média de 65% do espaço utilizado.

LESS_RETANGULAR, que coloca primeiro as peças que menos espaço ocupam dentro da sua moldura retangular (*bounding box*), e WIDER foram as seguintes melhores heurísticas estáticas, com um aproveitamento médio na ordem dos 64-65%. WIDER, para além de colocar primeiro os polígonos mais largos, escolhe sempre a rotação que minimiza a largura do polígono o que justifica os bons resultados obtidos.

As heurísticas que priorizam as peças mais irregulares (MORE_IRREGULAR) e as mais altas (HIGHER) geram mais desperdício do que as estratégias anteriores, obtendo um aproveitamento apenas de 62%. A estratégia de dar prioridade às peças mais altas prejudica o resultado porque, no caso de serem permitidas várias rotações, são escolhidas aquelas em que a dimensão mais pequena coincide com a altura e a maior com a largura, originando uma disposição mais longa.

No entanto, todas as regras definidas resultaram em disposições, em média, mais compactas que a aleatória (61%).

³<http://paginas.fe.up.pt/esicup>

Duplicando a resolução, o aproveitamento do espaço aumenta cerca de 1% ou 2%. Em alguns casos, a representação mais pormenorizada permite identificar posicionamentos novos, gerando disposições distintas. O problema *Marques*, por exemplo, apresenta uma redução do aproveitamento de 1% nas heurísticas LARGER e WIDER com a resolução superior. O mesmo sucede com *Mao* usando a regra HIGHER (-1%) e *trousers* com a regra MORE_IRREGULAR (-3%). Por outro lado, o aproveitamento de *shirts* com WIDER passa de 53% para 78%, graças à maior resolução.

Os resultados relativos ao aproveitamento para as diferentes heurísticas implementadas encontram-se na tabela B.19 do anexo B, assim como os tempos de processamento correspondentes, analisados de seguida.

Tempo de processamento No que diz respeito ao tempo dispendido, observou-se que, colocando primeiro as peças mais largas (WIDER), se reduz substancialmente o tempo de execução, demorando, em média, 0,011 segundos a posicionar cada peça enquanto que as restantes heurísticas estáticas que rondam os 0,015. Tal deve-se, essencialmente, ao facto das disposições originadas por WIDER tornarem o contorno do espaço disponível menos angular, facilitando, não só a detecção de contornos, mas sobretudo a pesquisa do vértice *bottom-left*.

A pesquisa dinâmica demora, em média, 1,3 segundos a colocar uma peça. Há que notar que esta pesquisa obtém melhores resultados para problemas com um grande número de peças, mas com pouca diversidade entre elas, isto é, com um número reduzido de tipos e de rotações admissíveis. Como testa a colocação de todos os tipos de peças disponíveis decidindo o mais vantajoso, quanto menos tipos existirem mais rápida é a decisão.

Em qualquer uma das heurísticas, para decidir a posição de uma peça é necessário detetar os contornos da região de posicionamentos admissíveis e aplicar a regra de posicionamento. Como tal, o tempo de posicionamento de uma peça depende tanto da resolução da imagem como da complexidade do contorno do espaço disponível: quantos mais pixels constituírem a imagem, mais tarda a detecção de contornos e quantos mais vértices formarem o contorno, mais tarda a pesquisa da posição *bottom-left*. Os testes realizados com uma maior resolução demoram, em média, mais 50% a 60% do tempo a posicionar uma peça, o que é positivo visto que se duplicou a resolução.

O tempo de posicionamento de uma peça varia, ainda, consoante o tipo de heurística utilizada. Nas heurísticas estáticas, o processo de colocação de uma peça engloba desenhar os seus NFP's e ler esta imagem para memória (a partir de qual se obtém o contorno acima referido). O tempo de desenho não é constante: aumenta à medida que o número de peças colocadas também aumenta. Isto deve-se, naturalmente, ao facto de o número de peças a desenhar ser maior. Esta situação é ilustrada pelo gráfico 5.12 que demonstra o aumento do tempo de desenho em função do número de peças colocadas, enquanto que o tempo de aplicação da regra *bottom-left* aos contornos se mantém relativamente constante (<10 milissegundos). Para as heurísticas estáticas, o número de tipos de peça disponíveis não é relevante dado que a sequência de peças é pré-definida e o tempo de atualização das disposições dos NFP's de cada tipo de peça é residual.

Testes e análise dos resultados

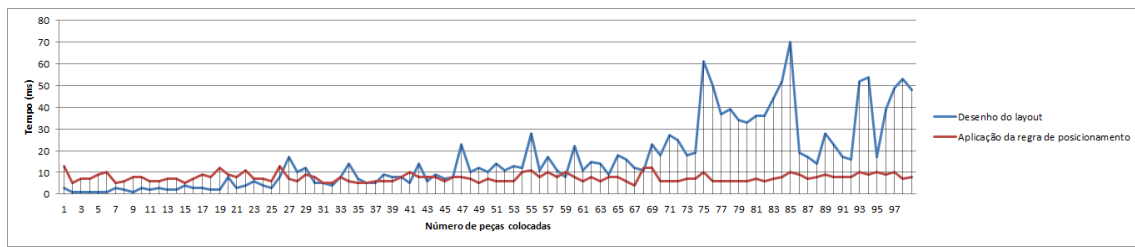


Figura 5.12: Tempos (em milissegundos) de desenho e da aplicação da regra de posicionamento em função do número de peças colocadas, na resolução do problema *shirts* usando uma heurística estática (WIDER).

Na heurística dinâmica o procedimento é distinto. Para posicionar uma peça experimentam-se todas as possibilidades e escolhe-se a que tiver menor desperdício. Para avaliar o desperdício de cada possibilidade desenham-se as disposições dos seus NFP's. Sendo T o número de tipos de peças disponíveis (não considerando rotações), são desenhadas T^2 imagens por peça colocada. Note-se que T vai diminuindo à medida que as peças de um determinado tipo de esgotam, o que se traduz na diminuição do tempo total de desenho uma vez que existem menos imagens a desenhar; passando, por exemplo, do *render* de T^2 imagens para $(T - 1)^2$ e assim sucessivamente. Existem dois fatores que influenciam o tempo médio de posicionamento de uma peça na heurística dinâmica: o número de tipos de peça a experimentar e o número total de peças a colocar (à semelhança do que sucede nas heurísticas estáticas, um *layout* com mais peças vai ter tempos de posicionamento maiores).

É possível visualizar esta situação no gráfico da figura 5.13. O tempo de avaliação e escolha entre *layouts* possíveis é desprezável comparado com o tempo de desenho. No entanto, o tempo de escolha vai diminuindo devido ao facto de o número de *layouts* possíveis ir decrementando na mesma medida que o número de tipos de peças. O tempo de desenho aumenta à medida que vão sendo colocadas novas peças, existindo um decréscimo repentino (assinalado pelas linhas a tracejado) quando se esgotam as peças de um tipo, visto que nas iterações seguintes se avalia menos um *layout*.

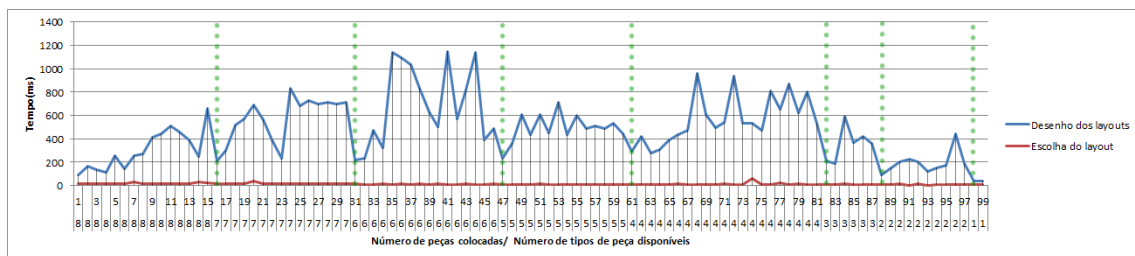


Figura 5.13: Tempos (em milissegundos) de desenho e de escolha do melhor *layout* em função do número de tipos de peça disponíveis (em cima) e do número de peças colocadas (em baixo), usando a heurística dinâmica. As linhas verticais sinalizam que um tipo de peça foi retirado da lista dos disponíveis.

Comparação com outras abordagens A maioria dos algoritmos disponíveis na literatura são de melhoramentos e realizam alterações numa solução fornecida inicialmente de forma a melhorá-la. Geram-se assim várias soluções, das quais se disponibiliza a melhor. A fim de comparar o algoritmo implementado, que é construtivo (gera apenas uma única solução), com estas abordagens considera-se apenas o tempo que tardam a realizar uma disposição. Compararam-se resoluções de problemas de *strip-packing*, com o algoritmo *2-exchange* [GO02] e a abordagem de Burke et al [BHKW06]⁴.

Na tabela B.20 pode ver-se o tempo médio necessário para gerar uma solução (T_{sol}) nos algoritmos de Gomes e Oliveira e Burke e em duas das heurísticas implementadas: a heurística estática capaz de gerar soluções mais compactas - LARGER - e a heurística dinâmica. De [BHKW06] é referido, ainda, o tempo dispendido até obter a melhor solução. Deve-se mencionar que estes resultados foram obtidos com um processador Pentium 4 de 2GHz. Para [GO02] disponibiliza-se o tempo total da execução de 20 iterações. O tempo médio de geração de uma solução foi obtido dividindo o tempo total por $20 \times 3 \times N$, em que N é o número de peças. Isto porque uma iteração deste algoritmo corresponde à geração de $3 \times N$ soluções, uma vez que o método utilizado gera, em cada iteração, todas as soluções resultantes da troca de cada peça da sequência com uma das 3 seguintes. Os testes foram realizados num Pentium III 450 MHz. O algoritmo de posicionamento com representação gráfica foi testado num processador i5 com 2,27 GHz.

O comprimento e aproveitamento destes dois algoritmos referem-se à melhor das soluções geradas, pelo que não são comparáveis com o algoritmo construtivo implementado no contexto da dissertação, que corre apenas uma vez.

No que diz respeito ao tempo, verifica-se que o algoritmo *2-exchange* gera soluções mais rapidamente para problemas de formas simples, como *shapes* e *shirts*. Porém esta tendência inverte-se quando a complexidade das formas aumenta: a abordagem estática implementada é a mais rápida a descodificar uma sequência num *layout* para *trousers*. O mesmo se observa no algoritmo de Burke et al, que obtém tempos idênticos a LARGER para os problemas mais simples, sendo significativamente mais lento a gerar disposições de *shirts*, *swim* e *trousers*. A heurística dinâmica, por incluir também o processo de decisão da peça a colocar de seguida, é a mais demorada.

Deve-se salientar que o tempo de execução é independente da complexidade das peças a serem colocadas. Nos algoritmos que utilizam representações poligonais, como é o caso de [BHKW06] e de [GO02], peças com um número muito elevado de vértices e com muitas irregularidades introduzem atrasos significativos no tempo de processamento visto existir a necessidade de gerir toda a geometria inerente (calcular intersecções entre os NFP's, identificar arestas exteriores, etc). Na implementação realizada, este processo é substituído pela leitura da imagem e detecção dos contornos da mesma (recorrendo à biblioteca OpenCV), que é independente da geometria e, por isso, se mantém sensivelmente constante. No caso de polígonos simples como os de *shapes* o processo utilizado é mais demorado, mas para problemas complexos (como *swim*) a vantagem deste método é significativa.

⁴Não foi possível realizar a comparação com problemas em que a superfície de corte tem defeitos [BBGM12] visto os dados dos problemas não estarem disponíveis

Detecção de buracos e encaixes perfeitos A detecção de buracos é realizada sem inconvenientes desde que a resolução definida o permita. A fim de evitar as dificuldades em identificar posicionamentos estreitos, como encaixes perfeitos e retas, deixa-se o espaçamento de um pixel entre as peças; esta margem permite detetar posicionamentos possíveis entre duas peças que estejam "encostadas" que passariam despercebidos se não existisse (ver imagem 4.8 do capítulo 4). É, no entanto, possível representar as imagens com resoluções suficientemente elevadas para que o desperdício correspondente a estes espaços não tenha um impacto de maior. A margem deixada entre as peças é configurável e encontra-se explicada em mais detalhe no capítulo relativo à implementação, secção 4.2.2.

5.3 Resumo

Estudou-se o comportamento de 4 soluções distintas para a obtenção dos NFP's: os algoritmos de deslizamento ([GO02] e [BHKW06]) e a soma de Minkowski com decomposição [CGA], disponíveis na bibliografia, e soma de Minkowski com representação gráfica e o diagrama de declives, implementadas no contexto da dissertação.

Analysaram-se quatro algoritmos para a decomposição: decomposição Ótima, de Hertel e Mehlhorn, de Greene e *Small side angle bissector*. Pôde-se concluir que *small side angle bissector* é a decomposição não-ótima mais eficiente e que melhor aproxima a ótima. Esta decomposição substitui perfeitamente a ótima na solução implementada e na soma de Minkowski [CGA] com decomposição de peças simples. Ao testar este último algoritmo com um conjunto de peças mais complexas, a decomposição ótima obteve melhores resultados.

Os métodos propostos para o cálculo e desenho dos NFP's, por não comporem os NFP's parciais num só, são mais eficientes do que a soma de Minkowski com decomposição disponível no CGAL. Os dois algoritmos permitem detetar buracos nos NFP's, uma vez efetuado o *render*, no entanto, encaixes perfeitos não são detetados. O algoritmo de deslizamento obteve resultados mais rapidamente do que a soma de Minkowski com representação gráfica para o problema mais simples; no entanto, o seu desempenho é pior do que o das duas soluções implementadas para formas complexas, com grande número de vértices, nomeadamente formas côncavas. O algoritmo de deslizamento de [GO02], ao contrário das suas outras soluções testadas, não deteta buracos nos NFP's, porém, a versão melhorada apresentada por [BHKW06] identifica simultaneamente buracos e encaixes perfeitos.

Avaliou-se ainda a possibilidade de paralelizar a soma dos pontos (realizada no cálculo da soma de Minkowski) em *OpenCL*, mas verificou-se que tal apenas é viável se tanto as somas como o cálculo da respetiva *convex hull* forem realizados em GPU de forma a poder desenhar os polígonos diretamente, sem ser necessário transferir novamente os dados para memória, que constitui o passo mais moroso de todo o processo.

A implementação original da soma de Minkowski devolve a representação de um polígono com concavidades e buracos (que não inclui encaixes perfeitos), enquanto que a versão implementada devolve um conjunto de polígonos convexos (que, juntos, definem o NFP). Visto que os

polígonos convexos são mais simples de desenhar em *OpenGL* do que os irregulares, mediram-se os tempos de desenho de ambas as representações, constatando-se que é mais eficiente gerar *display lists* de um número de polígonos convexos não muito elevado (como é o caso do *output* dos algoritmos implementados) do que fazer a *tessellation* de um polígono complexo.

A tabela 5.4 resume os resultados dos testes efetuados neste capítulo.

	Algoritmo de Deslizamento	Soma de Minkowski com decomposição	Soma de Minkowski com representação gráfica
Eficiência	4	2	3
Rigor	4	4	2
Representação	Vetorial	Vetorial	Gráfica
Deteção de buracos	Não	Sim	Sim
Deteção de encaixes perfeitos	Não	Não	Não

Tabela 5.4: Comparação das características dos 3 algoritmos testados. A avaliação da eficiência e do rigor é feita numa escala de 1 a 5.

No que diz respeito ao algoritmo de posicionamento, verificou-se a vantagem da heurística *LARGER* que, ao posicionar as peças maiores primeiro, obtém soluções mais compactas rapidamente. A heurística dinâmica maximiza a região de posicionamentos admissíveis mas é consideravelmente mais lenta. Nas heurísticas estáticas, o tempo de desenho aumenta, naturalmente, com o número de peças a desenhar; o tempo de posicionamento de uma peça depende, sobretudo, da resolução que determina o número de pontos candidatos à posição da peça seguinte. Na heurística dinâmica também se verificam estes contratempos. Esta heurística é mais demorada porque avalia todas as opções de posicionamento em cada iteração, o que depende do número de tipos de peças. Assim, quando os tipos de peças disponíveis decrementam, o posicionamento é feito mais rapidamente dado que há menos opções a avaliar. As heurísticas estáticas recorrendo à representação gráfica não se revelam mais eficientes do que as que utilizam representações poligonais referidas na literatura na solução de problemas com peças de contornos simples. Porém, a vantagem é significativa quando se compara o tempo de descodificação de uma sequência de peças complexas o que, nas representações poligonais, envolve muito processamento geométrico. Nesta implementação, o tempo de desenho de um *layout* é independente da complexidade da geometria.

Testes e análise dos resultados

Capítulo 6

Conclusões e trabalho futuro

6.1 Conclusões

Nos problemas de *nesting* procura-se encontrar uma disposição (*layout*) compacta (isto é, que minimize o desperdício) para um conjunto de peças irregulares numa superfície que tanto pode ser um retângulo de comprimento indefinido (problemas de *strip packing*) como um polígono complexo.

Nas abordagens disponíveis na literatura para a resolução deste tipo de problemas uma disposição pode ser obtida colocando uma sequência de peças ordenadamente e segundo uma certa regra (por exemplo *bottom-left*) na superfície de corte ou explorando as movimentações possíveis das peças no próprio *layout*.

Estes problemas possuem uma forte componente geométrica que se torna complexa de gerir, mesmo usando uma representação apropriada. Os dispositivos de processamento gráfico disponibilizam um grande conjunto de recursos para a manipulação da geometria e foram utilizados a fim de suavizar o impacto da complexidade das formas no tempo de geração de um *layout*.

No que diz respeito à programação em GPU, analisaram-se duas vertentes: gráfica (*render* de imagens, processamento da geometria, aplicação de texturas, iluminação...) e paralela (tirar partido do modelo SIMD do GPU para melhorar o rendimento de aplicações onde exista paralelismo ao nível dos dados) e estudaram-se as vantagens de ambas para a resolução do tipo de problemas de corte estudado [WHS07].

Na sequência desta análise, decidiu-se que se tiraria partido, principalmente, da vertente gráfica dos GPU's pois é a que melhor se adequa ao processamento geométrico e gráfico essencial à resolução dos problemas de *nesting*. Para tal, recorreu-se à API *OpenGL*, uma interface que disponibiliza os recursos do *hardware* gráfico.

Os algoritmos para a resolução deste tipo de problemas consideram dois tipos de representações: em grelha e poligonais. O algoritmo implementado utiliza uma representação em grelha que é equivalente à representação das imagens nos dispositivos de processamento gráfico. As representações poligonais são mais precisas pois não dependem da resolução da imagem e são mais simples de armazenar pois guardam-se apenas as coordenadas dos vértices dos polígonos. No

entanto, arriscam a ter que realizar cálculos complexos e repetitivos, pelo que verificam a necessidade de simplificar o tratamento da geometria recorrendo a conceitos de grande utilidade (função-D, NFP, função-Phi). No entanto, a memória dos dispositivos gráficos permite armazenar imagens com resoluções consideráveis, mantendo o nível de detalhe necessário para gerar disposições com qualidade. A representação gráfica evita cálculos trigonométricos demorados para determinar a região de posicionamentos admissíveis num *layout*, o que a torna adequada ao processamento de problemas com formas complexas.

Para determinar a região de posicionamentos admissíveis para uma peça recorre-se ao conceito de *no-fit-polygon*, que também foi objeto de estudo nesta dissertação. Os algoritmos para o cálculo dos NFP's podem distinguir-se em duas categorias: as abordagens baseadas no deslizamento de um polígono em torno de outro e aquelas que realizam a decomposição em polígonos convexos de forma a simplificar os cálculos. A principal dificuldade destes últimos é ser necessário tornar a compôr os sub-polígonos, o que é um processo demorado. Numa representação gráfica, porém, a composição está implícita no *render* uma vez que desenhando os sub-polígonos nas posições respetivas e com a mesma cor não se distinguem os seus contornos nem intersecções: a região colorida corresponde aos posicionamentos inviáveis. Por esta razão implementaram-se dois algoritmos baseados na decomposição: um que utiliza as somas de Minkowski e outro que constrói o diagrama de declives dos sub-polígonos. Os dois são equivalentes e o seu desempenho depende da decomposição efetuada, se bem que em decomposições com um grande número de vértices o diagrama de declives é ligeiramente mais rápido. Para realizar a decomposição em polígonos convexos elegeu-se a estratégia *small side angle bisector*, disponível na biblioteca CGAL pois revelou-se ser a decomposição mais rápida de gerar que melhor aproxima a ótima.

A implementação realizada obteve resultados mais rapidamente do que o algoritmo de deslizamento para peças complexas visto que este último algoritmo implica um maior número de testes no processamento de peças com um elevado número de vértices côncavos. A abordagem implementada, ao utilizar uma decomposição rápida e eficiente, minimiza o impacto da complexidade das peças no tempo de execução.

Avaliaram-se os algoritmos de posicionamento desenvolvidos para a resolução de problemas de *nesting*. Estes podem ser de dois tipos: construtivos ou de melhoramentos. Enquanto os primeiros constroem uma solução para o problema, os segundos partem de uma solução admissível, modificando-a iterativamente a fim de a melhorar. No contexto desta dissertação, foi dada particular ênfase à vertente construtiva destes algoritmos uma vez que o objetivo é minimizar o tempo de geração de um *layout*.

O algoritmo de posicionamento implementado baseia-se no algoritmo de [GO02] que aplica a regra de posicionamento *bottom-left* aos vértices dos contornos da região de posicionamentos admissíveis. A diferença entre a implementação de Gomes e Oliveira e a realizada no âmbito desta dissertação é a forma de obter esses contornos: enquanto que no primeiro algoritmo se

recorre a cálculos trigonométricos identificando as intersecções dos NFP's e do *inner-fit-polygon*, no segundo utiliza-se o processamento de uma imagem gerada com recurso a OpenGL.

Este algoritmo de posicionamento é construtivo e, na implementação realizada, definem-se heurísticas de posicionamento estático e dinâmico. As heurísticas de posicionamento estático descodificam uma sequência de peças pré-definida num padrão de corte enquanto que a heurística dinâmica, antes de efetuar um posicionamento, avalia todas as possibilidades para todos os tipos de peças distintos disponíveis, colocando a peça que gerar uma solução com menos desperdício. Este último processo é visivelmente mais lento.

Compararam-se os tempos de descodificação de uma sequência de peças num *layout* da abordagem gráfica com os dos algoritmos anteriores (poligonais), verificando-se que existe uma vantagem significativa no posicionamento de formas complexas (com muitos vértices e com contornos irregulares). Tal deve-se ao facto da representação gráfica ser praticamente independente da geometria.

6.2 Trabalho futuro

A implementação realizada resolve grande parte das dificuldades das implementações anteriores no que diz respeito ao processamento da geometria. Porém, o recurso aos dispositivos de processamento gráfico introduz outras dificuldades, nomeadamente no que diz respeito à transferência de dados do GPU para memória. Qualquer otimização que reduzisse a quantidade de dados a transferir ou a frequência com que é necessário transferi-los reduziria a latência deste processo, tornando o algoritmo mais eficiente. Efetivamente, quando se posicionam as primeiras peças, não é necessário ler do GPU todo o comprimento da superfície de corte uma vez que estas ocupam os posicionamentos mais à esquerda. Desta forma é possível reduzir o comprimento da imagem a ler para memória sem penalizar o conjunto de posicionamentos admissíveis a ser considerados na iteração seguinte.

Uma maior utilização dos recursos disponibilizados pela biblioteca OpenGL seria também benéfica se permitisse evitar o processo repetitivo de *render* do *layout* a ser construído. Na implementação realizada o tempo de *render* aumenta à medida que o número de peças colocadas cresce. Utilizar um *buffer object* para guardar as representações intermédias do *layout* tornaria o tempo de desenho constante pois, a cada iteração, seria apenas necessário realizar o *render* da peça nova sobre a imagem.

Outra limitação do método utilizado é a representação de encaixes perfeitos, nomeadamente nos NFP's. Efetivamente, se os contornos dos NFP's "encostarem" (definindo um encaixe perfeito), a representação gráfica não permite identificar a linha de separação entre ambos. Uma melhoria possível seria, ao representar os NFP's graficamente, distinguir sempre os seus buracos, mesmo que estes fossem encaixes perfeitos. O número de NFP's a calcular pode também ser otimizado. Para dois polígonos irregulares A e B, o NFP_{BA} é igual ao de NFP_{AB} , rodado 180°. Partindo deste princípio é possível reduzir para metade o cálculo dos NFP's. Para além disso, quando existem rotações, a forma do NFP entre dois polígonos apenas diverge se a rotação relativa entre eles

Conclusões e trabalho futuro

variado, o que permite uma redução ainda maior do número de NFP's a calcular para problemas com rotações.

Finalmente, seria ainda interessante utilizar a abordagem gráfica desenvolvida em algoritmos de melhoramentos que realizam uma pesquisa, gerando várias soluções. Isto permitir-nos-ia fazer uma melhor apreciação das vantagens de uma representação gráfica.

Anexo A

Características dos algoritmos de posicionamento

Aqui encontram-se resumidas as características dos algoritmos de *nesting* enunciados na revisão bibliográfica.

Partial Solutions: Placement Rules							
	BL			Floating Origin			Constrain Logic Programming
	Normal	w/ Hole Filling		Min Sol Area	Min Sol Len	Max Overlap	
		Grid	NFP				
Grid or NFP	Any	Grid	NFP	NFP	NFP	NFP	NFP
Hole Filling	No	Yes	Yes	No	No	No	
Efficient for Irregular Stock Sheets	No	No	No	Yes	Yes	Yes	
Computational Cost							Efficient for small problems

cas dos algoritmos de posicionamento

Tabela A.1: Heurísticas para o posicionamento das peças em soluções parciais.

Partial Solutions: Placement Order					
	Monte Carlo	Dynamic Selection			
		Normal	w/ Search Tree	w/ Search Tree and Pruning	Beam Search
Search Type	Random	Greedy	Depth-first	Depth-first	Breadth-first
Number of Levels Explored on the Tree	n/a	1	n	n	Breadth-first exploration
Pruning	n/a	No	No	Yes	Yes
Evaluation of the Partial Solution	n/a	Relative waste Overlap Relative distance Waste minus overlap Relative waste plus relative distance Distance minus overlap Waste minus overlap plus distance The amount of waste generated and an estimate of the future waste	One that selects a predefined number of successors for each node according to the evaluation of the partial solution generated by the successor node and A global evaluation found by greedily packing all the remaining pieces		

ento

Tabela A.2: Regras de posicionamento das peças na construção de soluções parciais.

Complete Solutions: Sequence						
	Tabu Search	Neighbourhood Search			Genetic	
		All Moves 1 Piece	1 Move all Pieces	Insert Moves Only	Normal	w/ Mult Sheets
Hole-filling						
Permitted Moves	Swap or Insert	Depends on the positioning rule Swap and Insert	Depends on the positioning rule Swap or Insert	Insert	Depends on the positioning rule Crossover of piece sequence Mutation: 90 rotation	Crossover of piece sequence Mutation: rotations of a piece Mutation: swap
						Yes
						Packing from the left end of the stock sheet (sequence determined by the X-coordinate of each piece in the previous operation)
States	One state	One state	One state	Normal/ Adaptive	One state	One state
Operation Choice	Random	No choice	Random	No choice	Random w/ Probability	Random w/ Probability
	Random	Random	No choice	Normal: Random Adaptive: Random w/ Probability (depends on piece's weight)		
Piece Choice						
					n/a	n/a
Neighb Size	5	n	n	n	n/a	n/a
Feasibility	Feasible Solutions	Feasible Solutions	Feasible Solutions	Feasible Solutions	Feasible Solutions	Feasible Solutions
Computational Cost	Decoding the Seq	Decoding the Seq	Decoding the Seq	Decoding the Sequence	Decoding the Sequence	Decoding the Sequence

Tabela A.3: Algoritmos para melhorar soluções completas movendo as peças de uma sequência.

Complete Solutions: Layout						
	Simulated Annealing			Local Search		
	Normal	w/ Placement Pos on the Edge	w/ppe and Point of Minimum Overlap	Tabu Search	Random Displacements	Allow Overlap
Solution Space	Discrete		Discrete in one direction, Continuous in the other		Continuous	Continuous
Placement Quality	Penalized by Grid		Good		Good	Penalized by overlapping
Computational Cost	Fast Search	Fast Search	Fast Search	Fast Search		Difficult to solve overlapping
Implementation	Simple	Simple	Simple	Simple		
Placement Rule	Random	Random	Random	BL w/ Hole Filling	Random direction and distance	
Placement Order	Random	Random	Random	Non-increasing area	Random piece	
Moves	Swap/Insert	Swap/Insert	Swap/Insert	Swap/Insert	Swap/Insert	Swap/Insert
Searched Points	Grid Points	Intersections of the edges of The stock sheet and other pieces' edges	Subset of points where the point of minimum Vertices and intersections of the NFP's and the IFP's	Hole list Avoids recently visited solutions	n/a	n/a
Feasibility	Feasible		Allows Overlap	Feasible	Feasible	Allows Overlap
Cost Function	n/a	n/a	n/a	n/a	n/a	The weighted sum of the total length of the layout The sum of the pair wise overlap Overlap across the horizontal and vertical dims of the sheet Horizontal translation and the direct sum of overlap and length

Tabela A.4: Algoritmos para melhorar soluções completas movendo as peças no espaço.

Anexo B

Tempo de execução dos algoritmos de posicionamento e para o cálculo dos NFP's

	Ótima	Hertel e Mehlhorn	<i>Small side angle bisector</i>	Greene
NFP 0 0	583,6	62,2	48,3	840,6
NFP 0 1	404	28,6	21,4	256,1
NFP 0 2	425,8	93,4	111,1	2231
NFP 0 3	319,9	121	88,6	1092,4
NFP 1 1	67,2	12,4	14,5	131,8
NFP 1 2	197,3	40,2	47	344,4
NFP 1 3	186,6	51	42,6	248
NFP 2 2	240,5	153,2	178	849
NFP 2 3	160,6	151,2	109,5	732,4
NFP 3 3	123,7	139,6	53,7	332,3
Média	270,92	85,28	71,47	705,8
Desvio Padrão	151,8773	50,51258853	47,64069794	593,8416

Tabela B.1: Tempo (em milissegundos) de cálculo dos NFPs de *shapes* recorrendo à soma de Minkowski com decomposição, disponível na biblioteca CGAL.

Tempo de execução dos algoritmos de posicionamento e para o cálculo dos NFP's

	Ótima	Hertel e Mehlhorn	<i>Small side angle bisector</i>	Greene
NFPs 0 0	7990,1	171710	12055,5	50335,9
NFPs 0 1	3771,1	47681	5274,3	11056,4
NFPs 0 2	7517,7	83725,8	11183,6	14178,1
NFPs 0 3	5937,1	26066,5	8544,6	10516,2
NFPs 0 4	11378,6	86305,2	14375	16899,5
NFPs 0 5	10636,9	89826,9	12429,1	19971,7
NFPs 0 6	8192,9	105033	11346,9	26622,1
NFPs 0 7	1937,2	8206,9	1663,9	4180,6
NFPs 0 8	1244,9	611	181,6	611,5
NFPs 0 9	18106,5	535557	33112,5	53175,7
NFPs 1 1	843	3405,8	537,1	1617,1
NFPs 1 2	1549,2	6571	1756	3045,4
NFPs 1 3	1312,6	2938,5	1076,9	1377,8
NFPs 1 4	1592,8	7728,8	1472,7	3214,4
NFPs 1 5	1675,9	8249,1	1953,5	4404,9
NFPs 1 6	2284,2	9843,6	3711,9	5216
NFPs 1 7	446,7	700,1	285,4	805,5
NFPs 1 8	154,5	143,8	56,8	183,1
NFPs 1 9	6280	31746,6	8209,9	14901,6
NFPs 2 2	1977,5	7541,9	1385,1	4908,9
NFPs 2 3	1766	3980,2	2486,1	2686,9
NFPs 2 4	2255,1	13225	3615,4	6786,5
NFPs 2 5	3866,7	14583,6	4174,2	9147,9
NFPs 2 6	3478,6	15192,6	8218	9735,6
NFPs 2 7	926,5	1223,3	820,8	1368
NFPs 2 8	519,2	207,7	162,4	237,9
NFPs 2 9	8504,9	63457,8	17921	20665
NFPs 3 3	960,1	1301,3	1500,8	1238,4
NFPs 3 4	2530,9	4651,7	2193,2	3133
NFPs 3 5	2881,7	4128,1	2523,6	5420,5
NFPs 3 6	2865,8	5659,4	5320,2	4602,6
NFPs 3 7	594,9	443,1	406,4	795,6
NFPs 3 8	140,9	110,3	73,8	197,2
NFPs 3 9	6344	17767,6	13928,3	8584,4
NFPs 4 4	3689,8	20718,5	7803,2	11535,3
NFPs 4 5	4367,9	24029,1	8040,1	11360,6
NFPs 4 6	5704,4	40046,1	12467,5	10348,8
NFPs 4 7	831,4	1586,7	934,2	1937,6
NFPs 4 8	220,4	310,1	166,1	234,8
NFPs 4 9	10544,7	92896,6	32550	24263,5
NFPs 5 5	2440,4	8853,1	2723,4	9664,4
NFPs 5 6	4205,6	28428,1	7473,2	16385
NFPs 5 7	847,5	1563,6	769,1	2526,8
NFPs 5 8	309,2	246,6	118,9	365
NFPs 5 9	16163,3	92789,9	28098,2	22067,1
NFPs 6 6	10681,1	48845,2	9001,4	17483,8
NFPs 6 7	1771,1	2277,5	661,3	2288,3
NFPs 6 8	858,5	399,7	140,1	192,6
NFPs 6 9	13676,1	113165	31475,8	33732,9
NFPs 7 7	198,8	182	109,4	294,4
NFPs 7 8	84	59,6	29,2	96,6
NFPs 7 9	2190,4	4893	2257,9	6391,9
NFPs 8 8	26,8	31,6	13,6	44,2
NFPs 8 9	696,8	745,3	379,1	673
NFPs 9 9	34633,4	475170	56935,6	69132,2
Média	4483,75091	42486,573	7201,887	10233,47
Desvio Padrão	5895,78652	97398,836	10767,98	13913,78

Tabela B.2: Tempo (em milissegundos) de cálculo dos NFPs de *swim* recorrendo à soma de Min-kowski com decomposição, disponível na biblioteca CGAL.

Tempo de execução dos algoritmos de posicionamento e para o cálculo dos NFP's

	Ótima	Hertel e Mehlhorn	<i>Small side angle bisector</i>	Greene
NFP 0 0	23,3	13,3	6,2	28,3
NFP 0 1	13,5	7,6	2,9	12,6
NFP 0 2	33,3	18,8	8,8	33,9
NFP 0 3	42,1	20,6	7,3	28,5
NFP 1 1	5,4	3,4	0,9	4,8
NFP 1 2	24,6	11,6	4,4	17,6
NFP 1 3	32,4	14,4	3,6	13,7
NFP 2 2	46,7	25,4	12,1	44,3
NFP 2 3	55,9	27,2	10,2	34
NFP 3 3	63,3	29,7	9	26,8
Média	34,05	17,2	6,54	24,45
Desvio Padrão	17,34671	8,213525	3,382957	11,40309

Tabela B.3: Tempo (em milissegundos) da decomposição e do cálculo do invólucro convexo da soma de Minkowski dos sub-polígonos de *shapes*.

	Ótima	Hertel e Mehlhorn	<i>Small side angle bisector</i>	Greene
NFPs 0 0	23,5	13,8	6,2	28,4
NFPs 0 1	13,4	7,7	3,6	12,5
NFPs 0 2	36,4	18,9	10,2	36,9
NFPs 0 3	44,8	20,6	7	27,5
NFPs 1 1	5,5	3,4	0,8	4,7
NFPs 1 2	25,9	11,3	4,5	16,5
NFPs 1 3	36,4	14	4	13,2
NFPs 2 2	51,7	28,2	13,2	46,3
NFPs 2 3	57,2	33,2	10,3	33,1
NFPs 3 3	63,8	32,2	9,9	26,3
Média	35,86	18,33	6,97	24,54
Desvio Padrão	18,01556	9,700418	3,658702	12,07528

Tabela B.4: Tempo (em milissegundos) da decomposição e da construção do diagrama de declives dos sub-polígonos de *shapes*.

Tempo de execução dos algoritmos de posicionamento e para o cálculo dos NFP's

	Ótima	Hertel e Mehlhorn	<i>Small side angle bisector</i>	Greene
NFPs 0 0	2778,1	216,2	74,4	189,5
NFPs 0 1	1509,2	100,9	36,6	89,3
NFPs 0 2	1886,3	129,2	51,8	112,7
NFPs 0 3	1516,2	87,7	41,8	105,5
NFPs 0 4	1684,2	137,6	52	134,4
NFPs 0 5	1658,2	129,6	51,6	122,1
NFPs 0 6	2217,6	164,8	64,2	148
NFPs 0 7	1478,6	72,4	28,9	73,5
NFPs 0 8	1425,7	51,3	18,1	46
NFPs 0 9	2070,5	248,2	101,1	214,8
NFPs 1 1	208,6	50,4	17,2	42,4
NFPs 1 2	577,2	65,8	24,8	57,3
NFPs 1 3	190,6	40,3	20,3	43,9
NFPs 1 4	244,9	66	25,4	67,6
NFPs 1 5	327,9	67	28,2	62,9
NFPs 1 6	897,5	76,3	34	72,2
NFPs 1 7	148,2	30,5	12,1	32,4
NFPs 1 8	106,9	22,7	6,4	21
NFPs 1 9	696	122,8	52,9	108
NFPs 2 2	951,9	81,3	33,9	75,7
NFPs 2 3	570,7	54,3	28,8	67,6
NFPs 2 4	643,2	86,8	35,4	74,6
NFPs 2 5	727,9	386,7	35,5	77,2
NFPs 2 6	1353	97,1	43,9	89,3
NFPs 2 7	546,4	40,8	18,5	44,1
NFPs 2 8	493,7	27,9	10,7	31
NFPs 2 9	1086,6	150,3	67,5	133,5
NFPs 3 3	176,2	39	22,9	39,5
NFPs 3 4	242,9	53,5	28,2	56,2
NFPs 3 5	313,6	58,6	29,9	54,7
NFPs 3 6	907,1	69,6	37,5	65,5
NFPs 3 7	138,8	27,3	14,8	32,4
NFPs 3 8	97	17,9	8,4	19,2
NFPs 3 9	700,3	108,4	57,7	99,1
NFPs 4 4	298,3	81,8	37,5	81,3
NFPs 4 5	367,7	82,6	36,4	80,9
NFPs 4 6	985,8	105,8	47,5	98,6
NFPs 4 7	197,5	43,1	18,8	44,2
NFPs 4 8	156,8	26,2	11	28,1
NFPs 4 9	788,2	158,9	74,6	150,4
NFPs 5 5	458,8	83,4	37,1	78,3
NFPs 5 6	1090,8	97,6	55	97,2
NFPs 5 7	278,9	39,3	20,5	43,9
NFPs 5 8	227,2	27,5	13,4	28,8
NFPs 5 9	851,2	145,9	82,5	141
NFPs 6 6	1691,2	125,9	59,5	115
NFPs 6 7	976,6	52,4	24,8	54,1
NFPs 6 8	842,5	34,5	15,1	35
NFPs 6 9	1490,7	184,1	91,1	175,2
NFPs 7 7	98,5	20,5	9,6	27,4
NFPs 7 8	60,7	12,9	4,6	17,3
NFPs 7 9	674,6	81	40,9	82,5
NFPs 8 8	16,7	8,5	2,2	10,9
NFPs 8 9	592,3	62,9	26,3	54,8
NFPs 9 9	1324,2	273,7	139,9	257,3
Média	800,7436	89,59454545	37,52181818	80,09636
Desvio Padrão	635,5239	70,16469565	26,24769213	50,95104

Tabela B.5: Tempo(em milissegundos) da decomposição e do cálculo do invólucro convexo da soma de Minkowski dos sub-polígonos de *swim*.

Tempo de execução dos algoritmos de posicionamento e para o cálculo dos NFP's

	Ótima	Hertel e Mehlhorn	<i>Small side angle bisector</i>	Greene
NFPs 0 0	3006,7	208,7	72,1	177,4
NFPs 0 1	1605,3	103	36,8	87,7
NFPs 0 2	2020,1	128	60	107,6
NFPs 0 3	1593,5	91,5	43,1	80,4
NFPs 0 4	1669	133,2	57,1	120,9
NFPs 0 5	1785,6	124,5	61,9	118,2
NFPs 0 6	2336,8	157,1	77,8	143,8
NFPs 0 7	1523,9	69,3	30,7	66,5
NFPs 0 8	1511,7	54,2	19,3	43,8
NFPs 0 9	2059,2	227,9	104,2	227,4
NFPs 1 1	208,5	46	19	67,9
NFPs 1 2	589,3	60	25	69,3
NFPs 1 3	191	42,3	20,5	46,7
NFPs 1 4	247,9	64,3	27,1	69,3
NFPs 1 5	337,7	59,9	28	64,4
NFPs 1 6	919,3	75,7	40,4	77,7
NFPs 1 7	148,6	30,8	11,8	34,7
NFPs 1 8	107,1	20,2	7	22,1
NFPs 1 9	723,3	113,4	59,6	105,5
NFPs 2 2	948,2	78,2	36,3	68
NFPs 2 3	565,5	53,6	28,3	52,6
NFPs 2 4	643	78,8	37,8	71,4
NFPs 2 5	724	78,8	42,8	71,3
NFPs 2 6	1301,8	100,7	55,4	93,7
NFPs 2 7	524,9	39,9	19,5	40,4
NFPs 2 8	498,2	30,9	12,1	26,3
NFPs 2 9	1097,2	138,4	84,8	134,7
NFPs 3 3	173,4	34,7	26,5	37,7
NFPs 3 4	234,4	56,3	36,1	54,5
NFPs 3 5	308,8	55,1	33,1	59,8
NFPs 3 6	883,3	75,8	39,7	68,1
NFPs 3 7	133,4	28,2	16	30,5
NFPs 3 8	91,1	20,1	9,4	20,2
NFPs 3 9	705,2	108,8	60	100,4
NFPs 4 4	296,8	87,7	48,2	75,4
NFPs 4 5	366,7	83,8	40,9	77
NFPs 4 6	942,2	115,8	54,5	105,6
NFPs 4 7	197,1	45,1	20,5	41,8
NFPs 4 8	158,8	30,7	10,6	27,2
NFPs 4 9	846,1	169,6	87,6	141,9
NFPs 5 5	529,4	86,3	44	77,3
NFPs 5 6	1071,5	118,6	62,2	94,6
NFPs 5 7	266,1	48,4	24,4	43,8
NFPs 5 8	222,1	32,1	12,6	29,3
NFPs 5 9	859,6	168,9	97,4	134,5
NFPs 6 6	1629,5	141,3	78,7	113,5
NFPs 6 7	858,9	58,5	28,1	53,8
NFPs 6 8	794,5	40,6	18,8	34,8
NFPs 6 9	1464,4	202,8	105,3	193,1
NFPs 7 7	104,3	24,5	9,3	25,4
NFPs 7 8	55,4	17,6	5,6	20,7
NFPs 7 9	676,9	94,9	44,8	87,6
NFPs 8 8	20	9	1,8	9,9
NFPs 8 9	570,4	68,8	32,3	57,3
NFPs 9 9	1189,2	267,1	141,8	246,5
Média	809,76	85,46181818	41,97454545	79,12545
Desvio Padrão	661,6959	56,399244	29,05544679	50,23026

Tabela B.6: Tempo (em milissegundos) da decomposição e da construção do diagrama de declives dos sub-polígonos de *swim*.

Tempo de execução dos algoritmos de posicionamento e para o cálculo dos NFP's

	Ótima			Hertel e Mehlhorn			Small side angle bisector			Greene		
	Tempo	Polígono	Vértices/	Tempo	Polígono	Vértices/	Tempo	Polígono	Vértices/	Tempo	Polígono	Vértices/
			Polí-gono			Polí-gono			Polí-gono			Polí-gono
Peça 0	12,4	3	4	4,6	3	4	1,2	3	4	8,3	6	3
Peça 1	2,9	1	4	1,4	1	4	0,2	1	4	2,1	1	4
Peça 2	16,2	4	4,25	7,2	4	4,25	2,2	4	4,25	11,2	7	3,28571
Peça 3	25,1	3	5,33333	10,4	5	4	1,7	3	5,33	9,2	3	5,33333
Média	14,15			5,9			1,325			7,7		
Desvio	7,964452272			3,3121			0,739509973			3,399		
Padrão												

Tabela B.7: Tempo (em milissegundos), número de sub-polígonos e número médio de vértices por subpolígono das decomposições ótima, de Hertel e Mehlhorn e *Small side angle bisector* dos polígonos de *shapes*.

	Ótima			Hertel e Mehlhorn			Small side angle bisector			Greene		
	Tempo	Polígono	Vértices/	Tempo	Polígono	Vértices/	Tempo	Polígono	Vértices/	Tempo	Polígono	Vértices/
			Polí-gono			Polí-gono			Polí-gono			Polí-gono
Peça 0	1320,5	10	5	33,4	20	3,5	8,5	11	4,72727	29,5	18	3,66667
Peça 1	95,9	5	4,8	11,9	8	3,75	2,6	5	4,8	12,4	7	4
Peça 2	469,6	6	5,83333	17,3	10	4,3	4,6	7	5,28571	17,1	9	4,55556
Peça 3	81,9	6	4,33333	10,2	6	4,33333	4,5	6	4,33333	11,5	6	4,33333
Peça 4	137,3	8	4,125	16,8	12	3,41667	6	8	4,125	15,8	12	3,41667
Peça 5	206,9	7	5	18	10	4,1	4,6	8	4,625	20	10	4,1
Peça 6	809,6	8	5,125	22,8	14	3,78571	8,3	10	4,5	22	14	3,78571
Peça 7	44,7	3	6,33333	7,1	3	6,33333	1,7	3	6,33333	8,5	4	5,25
Peça 8	8,1	1	10	3,5	1	10	0,4	1	10	4,3	1	10
Peça 9	512,3	14	4,42857	40,3	22	3,54545	15,2	16	4,125	35,4	22	3,54545
Média	368,68			18,13			5,64			17,65		
Desvio	400,3155575			10,89202001			4,030682324			9,00547056		
Padrão												

Tabela B.8: Tempo (em milissegundos), número de sub-polígonos e número médio de vértices por subpolígono das decomposições ótima, de Hertel e Mehlhorn e *Small side angle bisector* dos polígonos de *swim*.

	Ótima	Hertel e Mehlhorn	Small side angle bisector	Greene
NFP 0 0	3,7	3,7	3,6	10,7
NFP 0 1	1,2	1,3	1,2	2,4
NFP 0 2	4,9	5,1	5	13,2
NFP 0 3	4,1	5,9	4,1	7
NFP 1 1	0,4	0,4	0,4	0,4
NFP 1 2	1,7	1,7	1,8	2,6
NFP 1 3	1,6	2,1	1,4	1,8
NFP 2 2	7,2	7,1	7,2	16,3
NFP 2 3	6,1	8,5	5,8	8,7
NFP 3 3	4,9	10,8	5,3	5
Média	3,58	4,66	3,58	6,81
Desvio Padrão	2,1554	3,25275268	2,167394749	5,062301848

Tabela B.9: Tempo (em milissegundos) da soma de Minkowski (sem incluir o tempo das decomposições) dos sub-polígonos de *shapes* resultantes das decomposições ótima, de Hertel e Mehlhorn.

	Ótima	Hertel e Mehlhorn	<i>Small side angle bisector</i>	Greene
NFP 0 0	3,5	4,3	3,4	11
NFP 0 1	1,2	1,2	1,1	2
NFP 0 2	4,6	4,9	4,7	12,7
NFP 0 3	3,7	6,4	3,8	6,2
NFP 1 1	0,4	0,4	0,3	0,4
NFP 1 2	1,6	1,6	1,5	2,4
NFP 1 3	1,1	1,9	1,2	1,1
NFP 2 2	6,7	7,8	6,6	15,6
NFP 2 3	5,9	7,5	4,8	8,3
NFP 3 3	3,8	8,3	3,5	3,6
Média	3,25	4,43	3,09	6,33
Desvio Padrão	2,024475241	2,8453647	1,9092	5,0634

Tabela B.10: Tempo (em milissegundos) da construção do diagrama de declives dos sub-polígonos de *shapes* resultantes das decomposições ótima, de Hertel e Mehlhorn.

Tempo de execução dos algoritmos de posicionamento e para o cálculo dos NFP's

	Ótima	Hertel e Mehlhorn	Small side angle bisector	Greene
NFP 0 0	47,7	139,3	56,8	120,1
NFP 0 1	24,8	58,1	26	47,7
NFP 0 2	31,6	76,2	37,3	67
NFP 0 3	26,5	44,7	29,2	44,4
NFP 0 4	39,1	83,1	37,7	80,9
NFP 0 5	34,9	72	38,1	67,1
NFP 0 6	40,4	99,7	50,1	90,1
NFP 0 7	16,9	26,7	17,2	29,2
NFP 0 8	7,4	12,5	7,9	11,2
NFP 0 9	65,2	154,9	74,6	139,9
NFP 1 1	13,8	21,8	11	18,2
NFP 1 2	14,7	29,2	16	24,6
NFP 1 3	12,6	18,2	12,9	16,2
NFP 1 4	17,9	31,4	16,4	29,4
NFP 1 5	15,9	29	19,4	26,8
NFP 1 6	18,1	39,3	21,5	37,1
NFP 1 7	7,7	10,6	7,6	12
NFP 1 8	3,5	4,9	3,5	4,7
NFP 1 9	31,6	59,9	32	58,7
NFP 2 2	20,6	40,6	26,4	35,7
NFP 2 3	18	23,8	18,6	22,7
NFP 2 4	22	43,9	24,2	42,3
NFP 2 5	21,1	41,1	27,5	36,1
NFP 2 6	25,6	53	31,9	50,8
NFP 2 7	12	14,6	11,5	16,9
NFP 2 8	4,9	6,9	5,2	6,3
NFP 2 9	42,4	81,6	49,3	74,9
NFP 3 3	14,4	14	14,4	14,6
NFP 3 4	18,5	25,5	19,1	28,5
NFP 3 5	17,8	24	20,9	23,3
NFP 3 6	20,8	31,9	24,9	32,1
NFP 3 7	8,6	8,9	8,6	11
NFP 3 8	4	3,9	4,1	4
NFP 3 9	35,8	48	38,4	48,1
NFP 4 4	25,1	49,5	24,9	48,9
NFP 4 5	23,8	42,5	27,6	42,3
NFP 4 6	27,8	56,6	32,2	58
NFP 4 7	11,4	15,5	14,4	21,5
NFP 4 8	5,5	7,1	5,1	7,1
NFP 4 9	44,7	92,6	48,6	87,7
NFP 5 5	22,6	42,7	27,8	38,7
NFP 5 6	25,5	54,2	38,3	51,4
NFP 5 7	11,4	14,5	12,5	17,1
NFP 5 8	5,1	6,5	5,8	6,5
NFP 5 9	42,8	81,9	52,7	83,5
NFP 6 6	30,5	73,6	45,4	73,1
NFP 6 7	13,5	21,3	15,6	27,8
NFP 6 8	6,1	9	7,3	9
NFP 6 9	49,9	108,3	65,4	112,2
NFP 7 7	5,8	5,5	5,5	8,1
NFP 7 8	2,5	2,7	2,6	3
NFP 7 9	20,8	30,4	23,3	36,1
NFP 8 8	1,3	1,2	1,2	1,2
NFP 8 9	9,3	14	10,6	13,2
NFP 9 9	81,4	185	103,2	184,3
Média	22,2473	43,4145	25,64	41,8782
Desvio Padrão	15,9249	39,0981	19,7375	36,714

Tabela B.11: Tempo (em milissegundos) do cálculo da soma de Minkowski (sem incluir o tempo da decomposição) dos sub-polígonos de *swim* resultantes das decomposições ótima, de Hertel e Mehlhorn.

Tempo de execução dos algoritmos de posicionamento e para o cálculo dos NFP's

	Ótima	Hertel e Mehlhorn	Small side angle bisector	Greene
NFP 0 0	47,7	129	50,5	112,9
NFP 0 1	21,4	57,7	23,3	42,3
NFP 0 2	34,4	77	35,2	63,2
NFP 0 3	25,3	42,7	26,6	40,6
NFP 0 4	34	77,9	34,2	70,6
NFP 0 5	32,9	69,7	37,7	63,3
NFP 0 6	36,5	95,1	44,6	86,2
NFP 0 7	14,8	26,2	16,4	27,7
NFP 0 8	6,7	11,7	7,6	10,8
NFP 0 9	57,7	160,7	69	130,5
NFP 1 1	11	21,7	10,7	17,6
NFP 1 2	14,6	29,1	16,6	23,6
NFP 1 3	12,9	17,3	11,9	15,4
NFP 1 4	15,5	31,6	15,8	27,6
NFP 1 5	15,3	29,2	19,2	26
NFP 1 6	18	38,3	21,3	35,2
NFP 1 7	7,2	10,6	7,8	11,3
NFP 1 8	3,4	4,8	3,3	4,3
NFP 1 9	30,2	59,2	32,2	55,1
NFP 2 2	19,4	40,2	24,1	34,1
NFP 2 3	16,2	22,6	17,9	20,8
NFP 2 4	20,7	41	22,9	38,9
NFP 2 5	24,9	38,2	24,7	38,9
NFP 2 6	27,2	50,6	31,9	48,4
NFP 2 7	9,8	14,7	14,1	15,4
NFP 2 8	4,2	6,4	5,8	5,9
NFP 2 9	40,2	78,3	49,5	71,3
NFP 3 3	14,5	14,2	13,5	13,9
NFP 3 4	18,2	25,6	19,8	24,3
NFP 3 5	18	23,8	19,4	24
NFP 3 6	20,6	30,8	25,7	30,4
NFP 3 7	8,5	10,3	18,2	9,9
NFP 3 8	6,3	3,9	3,8	5,4
NFP 3 9	32,8	46,3	37,9	45,2
NFP 4 4	23,1	46,8	24,7	44,3
NFP 4 5	24,9	43,8	25,4	42,6
NFP 4 6	27,5	57,3	30,9	54,9
NFP 4 7	11,2	15,4	12,8	18,1
NFP 4 8	4,8	6,9	4,8	6,9
NFP 4 9	45,8	81,6	48,4	88,7
NFP 5 5	23,7	40,8	26,9	40,8
NFP 5 6	25,7	52,1	35,1	52,4
NFP 5 7	10,8	14	12,4	16,9
NFP 5 8	4,9	8,3	7,3	6,6
NFP 5 9	43,2	85,9	52	81,6
NFP 6 6	30,8	73,7	42,1	69,7
NFP 6 7	14	20	16,6	22,9
NFP 6 8	5,4	8,8	6,6	8,8
NFP 6 9	50,3	105,2	60,4	104,8
NFP 7 7	4,7	4,8	5	7,1
NFP 7 8	2,1	2	2,2	2,7
NFP 7 9	19	30,1	22,6	36,2
NFP 8 8	1,3	0,9	1	1
NFP 8 9	8,5	13,5	10,1	13,6
NFP 9 9	76,6	166,8	98,1	168,1
Média	21,4418	42,0927	24,7	39,6309
Desvio Padrão	15,2322	37,3173	18,3932	34,2596

Tabela B.12: Tempo (em milissegundos) da construção do diagrama de declives (sem incluir o tempo da decomposição) dos sub-polígonos de *swim* resultantes das decomposições ótima, de Hertel e Mehlhorn.

Tempo de execução dos algoritmos de posicionamento e para o cálculo dos NFP's

NFP 0 0	2,2
NFP 0 1	1,1
NFP 0 2	4,5
NFP 0 3	4,9
NFP 1 1	0,6
NFP 1 2	1,5
NFP 1 3	1,3
NFP 2 2	5,7
NFP 2 3	6,4
NFP 3 3	6,2
Média	3,44
Desvio Padrão	2,1946

Tabela B.13: Tempo (em milissegundos) da geração dos NFP's de *shapes* com o algoritmo de deslizamento.

Tempo de execução dos algoritmos de posicionamento e para o cálculo dos NFP's

NFP 0 0	64
NFP 0 1	42,3
NFP 0 2	58,3
NFP 0 3	32,6
NFP 0 4	49
NFP 0 5	53,1
NFP 0 6	75,9
NFP 0 7	28
NFP 0 8	17,4
NFP 0 9	94,1
NFP 1 1	13,7
NFP 1 2	24,3
NFP 1 3	13,5
NFP 1 4	16,4
NFP 1 5	20,9
NFP 1 6	23,8
NFP 1 7	11,1
NFP 1 8	5,4
NFP 1 9	38,7
NFP 2 2	36,5
NFP 2 3	17,8
NFP 2 4	21,5
NFP 2 5	25,7
NFP 2 6	32,1
NFP 2 7	14,4
NFP 2 8	8,3
NFP 2 9	46,8
NFP 3 3	8,6
NFP 3 4	11
NFP 3 5	12,3
NFP 3 6	15,8
NFP 3 7	7,7
NFP 3 8	3,8
NFP 3 9	28,8
NFP 4 4	23,7
NFP 4 5	31,3
NFP 4 6	41,3
NFP 4 7	14,5
NFP 4 8	8,8
NFP 4 9	45,7
NFP 5 5	25,2
NFP 5 6	34,2
NFP 5 7	17,6
NFP 5 8	9,8
NFP 5 9	54,6
NFP 6 6	53,1
NFP 6 7	24
NFP 6 8	14,4
NFP 6 9	84,5
NFP 7 7	10,3
NFP 7 8	6,1
NFP 7 9	43,5
NFP 8 8	3,8
NFP 8 9	27,6
NFP 9 9	107,1
Média	30,0855
Desvio Padrão	22,9538

Tabela B.14: Tempo (em milissegundos) da geração dos NFP's de *swim* com o algoritmo de deslizamento.

Tempo de execução dos algoritmos de posicionamento e para o cálculo dos NFP's

	Ótima	Hertel e Mehlhorn	<i>Small side angle bisector</i>	
NFP 0 0	0,3	0,4	0,3	2,2
NFP 0 1	0,1	0,1	0,1	0,3
NFP 0 2	0,6	0,6	0,5	3,4
NFP 0 3	0,5	0,7	0,5	1,4
NFP 1 1	0	0	0	0,1
NFP 1 2	0,2	0,2	0,3	0,5
NFP 1 3	0,1	0,3	0,2	0,3
NFP 2 2	0,7	0,8	0,8	3,5
NFP 2 3	0,5	0,8	0,7	1,4
NFP 3 3	0,5	1	0,3	0,5

Tabela B.15: Tempo (em milissegundos) da criação da *display list* correspondente aos NFP's de *shapes* calculados na experiência anterior (tabela B.9).

Tempo de execução dos algoritmos de posicionamento e para o cálculo dos NFP's

	Ótima	Hertel e Mehlhorn	Small side angle bisector	Greene
NFP 0 0	19	21,8	7,3	30,7
NFP 0 1	3,1	9,5	4,4	11,9
NFP 0 2	5	11,3	8,1	15,9
NFP 0 3	5,2	8,5	4,5	12,4
NFP 0 4	13	15,9	5,1	14,4
NFP 0 5	4,7	17,8	7	17,3
NFP 0 6	8,3	15,1	9,5	23,9
NFP 0 7	2,7	5	3,1	10,2
NFP 0 8	1,3	2,6	1,5	5
NFP 0 9	12,6	24	13,1	43,5
NFP 1 1	1,9	2,2	2,1	6,3
NFP 1 2	2,1	4,9	3,1	9,1
NFP 1 3	1,7	3,4	2,3	5
NFP 1 4	2,7	7,4	3	9,6
NFP 1 5	2,7	4,4	3,4	9,2
NFP 1 6	4	6,7	3,7	12
NFP 1 7	1,5	2,6	1,6	5
NFP 1 8	0,6	1,1	0,8	1,4
NFP 1 9	4,4	11,2	5,8	18,5
NFP 2 2	3	8,6	4,8	9
NFP 2 3	2,9	4,3	3,3	6,5
NFP 2 4	3,9	7,8	4,8	10,4
NFP 2 5	3,4	7	5	10
NFP 2 6	4,1	9,1	5,5	13,7
NFP 2 7	1,5	2,6	2,2	4,5
NFP 2 8	0,7	1,5	1,1	1,6
NFP 2 9	6,6	14,7	8,9	16,7
NFP 3 3	3,3	3,3	1,9	2,6
NFP 3 4	4,4	5,2	2,8	5
NFP 3 5	3,9	4,1	3,4	4,3
NFP 3 6	4,7	5,9	5,1	7,2
NFP 3 7	1,8	1,7	4,2	2,7
NFP 3 8	0,8	0,8	2,3	1
NFP 3 9	7,6	17,3	7,7	11,8
NFP 4 4	4,6	6,3	8,2	10,6
NFP 4 5	5,9	7,5	3,7	8,9
NFP 4 6	9,4	11,7	5	12,4
NFP 4 7	4,7	5,9	1,6	4,8
NFP 4 8	2,8	3,8	0,9	1,9
NFP 4 9	15,5	22	8,5	18,3
NFP 5 5	5,8	6,7	5,4	7,6
NFP 5 6	4,2	11,6	12,5	10,2
NFP 5 7	1,9	2,6	4,5	4
NFP 5 8	0,9	1,5	2,4	1,5
NFP 5 9	7,6	13,4	12,3	14,2
NFP 6 6	5,2	15,9	6,2	12
NFP 6 7	2,3	4	3	4,7
NFP 6 8	0,9	2,1	1,3	1,8
NFP 6 9	8,2	19	10,7	18,6
NFP 7 7	0,9	0,9	1	1,5
NFP 7 8	0,4	0,4	0,4	0,6
NFP 7 9	3,6	6,1	4,4	7,2
NFP 8 8	0,1	0,1	0	0
NFP 8 9	1,7	3,1	2,4	3,1
NFP 9 9	13,3	27	14,7	25,8

Tabela B.16: Tempo (em milissegundos) da criação das *display list* correspondente aos NFP's de *swim* calculados na experiência anterior (tabela B.11).

Tempo de execução dos algoritmos de posicionamento e para o cálculo dos NFP's

NFP 0 0	4
NFP 0 1	1,4
NFP 0 2	2,4
NFP 0 3	2
NFP 1 1	0,1
NFP 1 2	1,7
NFP 1 3	0,7
NFP 2 2	1,6
NFP 2 3	2,3
NFP 3 3	1,3

Tabela B.17: Tempo (em milissegundos) da criação da *display list* correspondente aos NFP's de *shapes* resultantes da soma de Minkowski com decomposição (tabela B.1).

Tempo de execução dos algoritmos de posicionamento e para o cálculo dos NFP's

NFP 0 0	9,2
NFP 0 1	4,8
NFP 0 2	6,4
NFP 0 3	4,6
NFP 0 4	4,9
NFP 0 5	4,2
NFP 0 6	5
NFP 0 7	3,9
NFP 0 8	2,7
NFP 0 9	8,1
NFP 1 1	1,8
NFP 1 2	2,3
NFP 1 3	1,9
NFP 1 4	6,2
NFP 1 5	2,7
NFP 1 6	1,9
NFP 1 7	1,9
NFP 1 8	1,5
NFP 1 9	4
NFP 2 2	4,1
NFP 2 3	2,5
NFP 2 4	3,9
NFP 2 5	3,3
NFP 2 6	2,8
NFP 2 7	5,6
NFP 2 8	2,6
NFP 2 9	5
NFP 3 3	2
NFP 3 4	3,9
NFP 3 5	2,9
NFP 3 6	3,1
NFP 3 7	1,7
NFP 3 8	1,1
NFP 3 9	6,8
NFP 4 4	5
NFP 4 5	3,7
NFP 4 6	3,7
NFP 4 7	2,9
NFP 4 8	3,4
NFP 4 9	5,6
NFP 5 5	3,8
NFP 5 6	3,3
NFP 5 7	2,6
NFP 5 8	5
NFP 5 9	39,8
NFP 6 6	4,9
NFP 6 7	14,4
NFP 6 8	4,1
NFP 6 9	5,9
NFP 7 7	3,8
NFP 7 8	2,9
NFP 7 9	5,5
NFP 8 8	2,6
NFP 8 9	3,9
NFP 9 9	7,5

Tabela B.18: Tempo (em milissegundos) da criação da *display list* correspondente aos NFP's de *swim* resultantes da soma de Minkowski com decomposição (tabela B.2).

Tempo de execução dos algoritmos de posicionamento e para o cálculo dos NFP's

Fu		Mao	Marques	Shapes	Shirts	Swim	Trousers	Média							
Número de tipos de peças		36	32	8	16	20	34								
Total de peças		20	24	42	99	48	64								
Rotações		90	90	180	180	180	180								
Resolução		5	5	5	5	10	2	4							
Largura		2550	104	40	40	5772	79								
RANDOM															
Comprimento		45,4	2977	2364	101,8	104	76,2	71,4	75,6	70,3	8519	7311	304	300	
Aproveitamento		0,62775	0,49512	0,62351	0,6795	0,66513	0,52362	0,55882	0,71429	0,76814	0,5192	0,60498	0,71646	0,726013	0,6137
Tempo(s)		0,165	0,235	0,372	0,487	0,703	0,389	0,572	2,402	2,43	0,687	1,304	1,305	1,396	
Tempo/posicionamento(s)		0,01375	0,01175	0,0186	0,02029	0,02929	0,00926	0,01362	0,02426	0,02455	0,01431	0,02717	0,02039	0,0218125	0,01562
		1,51376	1,58298	1,44353	1,47044	1,01166	1,89811	1,0697318							
LESS_RECTANGULAR															
Comprimento		44,5	2102	2057	95,8	93,2	76,6	72,1	72,6	68,8	8131	7771	332,5	314	
Aproveitamento		0,64045	0,70122	0,71656	0,72206	0,7422	0,52089	0,5534	0,7438	0,78488	0,54397	0,56917	0,65505	0,693643	0,64576
Tempo(s)		0,163	0,26	0,388	0,419	0,676	0,45	0,665	2,133	2,92	1,012	1,329	1,085	1,569	
Tempo/posicionamento(s)		0,01358	0,013	0,0194	0,01746	0,02817	0,01071	0,01583	0,02155	0,02949	0,02108	0,02769	0,01695	0,0245156	0,01581
		1,36975	1,49231	1,61337	1,47778	1,36896	1,31324	1,4460829							
MORE_IRREGULAR															
Comprimento		45	2302	2267	111,4	110,5	76	75,8	72,8	71	8161	8061	303,5	315,25	
Aproveitamento		0,63333	0,6403	0,65018	0,62094	0,626	0,525	0,52639	0,74176	0,76056	0,54197	0,5487	0,71764	0,690892	0,62808
Tempo(s)		0,146	0,207	0,368	0,291	0,656	0,471	0,655	2,513	3,337	0,713	1,357	1,125	1,46	
Tempo/posicionamento(s)		0,01217	0,01035	0,0184	0,01213	0,02733	0,01121	0,0156	0,02538	0,03371	0,01485	0,02827	0,01758	0,0228125	0,01427
		1,44554	1,77778	2,2543	1,39066	1,32789	1,90323	1,2977778							
LARGER															
Comprimento		38,4	2172	2107	85,8	86,5	75,4	74	72,6	71,9	7771	7471	286	285	
Aproveitamento		0,74219	0,67862	0,69956	0,80621	0,79969	0,52918	0,53919	0,7438	0,75104	0,56917	0,59203	0,76155	0,764224	0,68741
Tempo(s)		0,172	0,257	0,349	0,318	0,63	0,39	0,705	2,167	3,268	0,661	1,32	1,185	1,538	
Tempo/posicionamento(s)		0,01433	0,01285	0,01745	0,01325	0,02625	0,00929	0,01679	0,02189	0,03301	0,01377	0,0275	0,01852	0,0240313	0,01397
		1,73737	1,35798	1,98113	1,80769	1,50808	1,99697	1,2978903							
HIGHER															
Comprimento		45,2	2555	2525	107	106,5	81,6	80,3	74,6	72,2	7851	7671	299	293	
Aproveitamento		0,63053	0,5769	0,58375	0,64648	0,64951	0,48897	0,49689	0,72386	0,74792	0,56337	0,57659	0,72844	0,743358	0,62186
Tempo(s)		0,154	0,301	0,382	0,329	0,738	0,405	0,689	2,187	2,829	0,697	1,386	1,029	1,53	
Tempo/posicionamento(s)		0,01283	0,01505	0,0191	0,01371	0,03075	0,00964	0,0164	0,02209	0,02858	0,01452	0,02888	0,01608	0,0239063	0,01464
		1,12409	1,2691	2,24316	1,70123	1,29355	1,98852	1,4868805							
WIDER															
Comprimento		42,6	2102	2057	89	89,6	75,8	72,4	75,8	68,9	7861	7801	287	286	
Aproveitamento		0,66901	0,70122	0,71656	0,77723	0,77202	0,52639	0,55111	0,52639	0,78375	0,56266	0,56698	0,7589	0,761552	0,64378
Tempo(s)		0,212	0,196	0,344	0,329	0,678	0,417	0,698	0,417	2,503	0,775	1,317	1,128	1,563	
Tempo/posicionamento(s)		0,01767	0,0098	0,0172	0,01371	0,02825	0,00993	0,01662	0,00421	0,02528	0,01615	0,02744	0,01763	0,0244219	0,01154
		1,89286	1,7551	2,06079	1,67386	6,0024	1,69935	1,3856383							
DYNAMIC															
Comprimento		42,4	2067		112		69,6		78,6		7373		312,5		
Aproveitamento		0,67217	0,7131		0,61762		0,57328		0,68702		0,5999		0,69697		0,65144
Tempo(s)		18,91	22,936		25,478		5,404		110,809		34,551		137,052		
Tempo/posicionamento(s)		1,57583	1,1468		1,06158		0,12867		1,11928		0,71981		2,14144		1,12763

Tabela B.19

Gomes e Oliveira [GO02]				Burke et al [BHKW06]				Static (LARGER)				Dynamic			
Total	Temposolução	Comprimento	Aproveitamento	Tmelhor	Temposolução	Comprimento	Aproveitamento	Temposolução	Comprimento	Aproveitamento	Temposolução	Comprimento	Aproveitamento	Temposolução	Comprimento
Fu	-	-	-	20,78	0,24	32,8	0,889	0,099	39,4	0,72335	18,91	42,4	0,67217		
Marques	-	-	-	4,87	0,25	80	0,865	0,257	2172	0,678622	22,936	2067	0,713095		
Mao	-	-	-	29,74	0,38	1854,3	0,795	0,318	85,8	0,806213	25,478	112	0,617617		
Shapes	551,73	27,3	-	2,19	0,82	60	0,665	0,39	75,4	0,529178	5,404	69,6	0,573276		
Shirts	6367,57	63,13	-	58,36	4,99	63,8	0,846	2,167	72,6	0,743802	110,809	78,6	0,687023		
Swim	-	-	-	607,37	12,39	6462,4	0,684	0,661	7771	0,569172	34,551	7373	0,599896		
Trousers	13613,67	245,75	-	736,15	7,89	246,57	0,885	1,185	286	0,761552	137,052	312,5	0,696972		

Tabela B.20: Tempo (em segundos) da representação de uma solução para os algoritmos estudados e os implementados.

Tempo de execução dos algoritmos de posicionamento e para o cálculo dos NFP's

Referências

- [AFH00] Pankaj K. Agarwal, Eyal Flato e Dan Halperin. Polygon decomposition for efficient construction of minkowski sums. 2000.
- [AFH02] Pankaj K Agarwal, Eyal Flato e Dan Halperin. Polygon decomposition for efficient construction of minkowski sums. *Computational Geometry*, 21(1-2):39–61, 2002.
- [Art96] Art, Jr., R. C. An approach to the two dimensional irregular cutting stock problem. Technical Report 36. Y08, IBM Cambridge Scientific Center, Cambridge, Massachusetts, USA, 1996.
- [AS80] A. Albano e G. Sapuppo. Optimal allocation of two-dimensional irregular shapes using heuristic search methods. *IEEE Trans. Syst., Man, Cybern., SMC-10*, 5:242–248, 1980.
- [BBGM12] Roberto Baldacci, Marco a. Boschetti, Maurizio Ganovelli e Vittorio Maniezzo. Algorithms for nesting with defects. *Discrete Applied Mathematics*, April 2012.
- [BD99] Julia A. Bennell e Kathryn A. Dowsland. A tabu thresholding implementation for the irregular stock cutting problem. *International Journal of Production Research*, 37(18):4259–4275, 1999.
- [BDD01] Julia A Bennell, Kathryn A Dowsland e William B Dowsland. The irregular cutting-stock problem * a new procedure for deriving the no- "t polygon. *Computers & Operations Research*, 28, 2001.
- [BHKW06] Edmund Burke, Robert Hellier, Graham Kendall e Glenn Whitwell. A New Bottom-Left-Fill Heuristic Algorithm for the Two-Dimensional Irregular Packing Problem. *Operations Research*, 54(3):587–601, May 2006.
- [BHW93] J. Błażewicz, P. Hawryluk e R. Walkowiak. Using a tabu search approach for solving the two-dimensional irregular cutting problem. *Annals of Operations Research*, 41:313–325, 1993. 10.1007/BF02022998.
- [BO06] Julia A Bennell e José F. Oliveira. The geometry of nesting problems: A tutorial. *European Journal of Operational Research*, 184(2):397–415, January 2006.
- [BO09] Julia A Bennell e José F. Oliveira. A tutorial in irregular shape packing problems. *Journal of the Operational Research Society*, 60:S93–S105, February 2009.
- [Bra00] G. Bradski. The OpenCV Library. *Dr. Dobbs's Journal of Software Tools*, 2000.
- [BS08] Julia a Bennell e Xiang Song. A comprehensive and robust procedure for obtaining the nofit polygon using Minkowski sums. *Computers & Operations Research*, 35(1):267–281, January 2008.

REFERÊNCIAS

- [BS10] Julia Bennell e Xiang Song. A beam search implementation for the irregular shape packing problem. *Journal of Heuristics*, 16:167–188, 2010. 10.1007/s10732-008-9095-x.
- [CD85] Bernard Chazelle e D. P. Dobkin. Optimal convex decompositions. *Computational Geometry*, pages 63–133, 1985.
- [CG89] R. Cuninghame-Green. Geometry, shoemaking and the milk tray problem. *New Scientist*, 1677:50–53, August 1989.
- [CGA] CGAL, Computational Geometry Algorithms Library. <http://www.cgal.org>.
- [CR03] MA Carravilla e C Ribeiro. Solving nesting problems with non-convex polygons by constraint logic programming. in *Operational Research*, 2000:1–15, 2003.
- [DVD02] Kathryn A Dowsland, Subodh Vaid e William B Dowsland. An algorithm for polygon placement using a bottom-left strategy. *European Journal of Operational Research*, 141(2):371–381, September 2002.
- [ENO07] J Egeblad, B Nielsen e A Odgaard. Fast neighborhood search for two- and three-dimensional nesting problems. *European Journal Of Operational Research*, 183(3):1249–1266, 2007.
- [GA99] Oliveira JF Gomes AM. Nesting irregular shapes with simulated annealing. *Extended Abstracts of MIC1999—III Metaheuristics International Conference*, pages 19–22, July 1999.
- [Gho91] Pijush K. Ghosh. An algebra of polygons through the notion of negative shapes. *CVGIP: Image Underst.*, 54:119–144, June 1991.
- [GO02] A.Miguel Gomes e José F. Oliveira. A 2-exchange heuristic for nesting problems. *European Journal of Operational Research*, 141(2):359–370, September 2002.
- [GO06] A. Miguel Gomes e José F. Oliveira. Solving irregular strip packing problems by hybridising simulated annealing and linear programming. *European Journal of Operational Research*, 171(3):811 – 829, 2006.
- [Gre83a] D. H. Greene. Fast triangulation of simple polygons. *Lecture Notes Comput. Sci.*, 158:207–218, 1983.
- [Gre83b] D. H. Greene. The decomposition of polygons into convex parts. *Computational Geometry, Advances in Computing Research*, 1:235–259, 1983.
- [HL95] Ralf Heckmann e Thomas Lengauer. A simulated annealing approach to the nesting problem in the textile manufacturing industry. *Annals of Operations Research*, 57:103–133, 1995. 10.1007/BF02099693.
- [IYN07] Takashi Imamichi, Mutsunori Yagiura e Hiroshi Nagamochi. An iterated local search algorithm based on nonlinear programming for the irregular strip packing problem. Technical Report 2007-009, Department of Applied Mathematics and Physics, Graduate School of Informatics, Kyoto University, 2007.
- [Jak96] Stefan Jakobs. On genetic algorithms for the packing of polygons. *European Journal Of Operational Research*, 88(1):165–181, 1996.

REFERÊNCIAS

- [KH10] David B. Kirk e Wen-mei W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach (Applications of GPU Computing Series)*. Morgan Kaufmann, 1 edition, February 2010.
- [LM11] Wei Li e Sara McMains. Voxelized Minkowski sum computation on the GPU with robust culling. *Computer-Aided Design*, 43(10):1270–1283, October 2011.
- [Mah84] Anantharam Mahadevan. *Optimization in computer-aided pattern packing (marking, envelopes)*. PhD thesis, 1984. AAI8507009.
- [Mar09] Clodéric Mars. The Minkowski sum, 2009.
- [MDL92] Victor J. Milenkovic, Karen Daniels e Zhenyu Li. Placement and compaction of nonconvex polygons for clothing manufacture. In *In Proceedings of the 4th Canadian Conference on Computational Geometry*, pages 236–243, 1992.
- [OF93a] J. E. Oliveira e J. S. Ferreira. Algorithms for nesting problems, in: *Applied Simulated Annealing*, ed. pages 255–274. LNEMS 396 Springer, 1993.
- [OF93b] J. F. Oliveira e J. S. Ferreira. Algorithms for Nesting Problems. In *Applied Simulated Annealing, Lecture Notes in Economics and Mathematical Systems*, pages 255–273. Springer-Verlag, 1993.
- [OGF00] José F. Oliveira, A. Miguel Gomes e J. Soeiro Ferreira. Topos – a new constructive algorithm for nesting problems. *OR Spectrum*, 22:263–284, 2000. 10.1007/s002910050105.
- [OLG⁺05] John D Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E Lefohn, Timothy J Purcell, E Aaron, A Survey e General-purpose Computation. A Survey of General-Purpose Computation on Graphics Hardware. *Computer Graphics Forum*, (August):21–51, 2005.
- [OSW⁺05] Opengl, D. Shreiner, M. Woo, J. Neider e T. Davis. *OpenGL(R) Programming Guide : The Official Guide to Learning OpenGL(R), Version 2 (5th Edition)*. Addison-Wesley Professional, August 2005.
- [RR01] A. Ramesh Babu e N. Ramesh Babu. A generic approach for nesting of 2-D parts in 2-D sheets using genetic and heuristic algorithms. *Computer-Aided Design*, 33(12):879–891, October 2001.
- [Seg86] Braga L.M. Segenreich, S.A. Optimal nesting of general plane figures: aMonte Carlo heuristical approach. *Computers & Graphics*, 10:229–237, 1986.
- [SNK96] Yu G Stoyan, M V Novozhilova e A V Kartashov. Mathematical model and method of searching for a local extremum for the non-convex oriented polygons allocation problem. *European Journal Of Operational Research*, 92(1):193–210, 1996.
- [SSGR02] Y. Stoyan, G. Scheithauer, N. Gil e T. Romanova. Phi-functions for complex 2d-objects, 2002.
- [Sto77] L.D. Stoyan, Y.G. Ponomarenko. Minkowski sum and hodograph of the dense placement vector function. *Reports of the SSR Academy of Science, SER.A*, 10, 1977.

REFERÊNCIAS

- [STS⁺01] Y. Stoyan, J. Terno, G. Scheithauer, N. Gil e T. Romanova. Phi-functions for primary 2d-objects, 2001.
- [TKM03] S. Takahara, Y. Kusumoto e S. Miyamoto. Solution for textile nesting problems using adaptive meta-heuristics and grouping. *Soft Computing - A Fusion of Foundations, Methodologies and Applications*, 7(3):154–159, January 2003.
- [WHS07] G Wascher, H Hausner e H Schumann. An improved typology of cutting and packing problems. *European Journal of Operational Research*, 183(3):1109–1130, December 2007.
- [Wik12a] Wikipedia. Minkowski addition — wikipedia, the free encyclopedia, 2012. [Online; accessed 9-February-2012].
- [Wik12b] Wikipedia. Ramer–douglas–peucker algorithm — wikipedia, the free encyclopedia, 2012. [Online; accessed 16-June-2012].
- [WT99] P D Watson e A M Tobias. An efficient algorithm for the regular w1 packing of polygons in the infinite plane. *Journal of the Operational Research Society*, 50(10):1054–1062, 1999.