Faculdade de Engenharia da Universidade do Porto

Biblioteca Geométrica para a Resolução de Problemas de Corte de Formas Irregulares

Nuno Miguel Neves de Abreu

Dissertação realizada no âmbito do
Mestrado Integrado em Engenharia Electrotécnica e de Computadores
Major Automação

Orientador: Prof. Dr. António Miguel da Fonseca Fernandes Gomes

Julho de 2008

U.PORTO

FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

MIEEC - MESTRADO INTEGRADO EM ENGENHARIA ELECTROTÉCNICA E DE COMPUTADORES 2007/2008

A Dissertação intitulada

"Biblioteca Geométrica para a Resolução de Problemas de Corte de Formas Irregulares"

foi aprovada em provas realizadas em 24/Julho/2008

o júri

Presidente Professor Doutor Adriano da Silva Carvalho
Professor Associado da Faculdade de Engenharia da Universidade do Porto

Professora Doutora Maria Teresa do Valle Moura da Costa
Equiparada a Professora Adjunta do Instituto Superior de Engenharia do Porto

Professor Doutor António Miguel da Fonseca Fernandes Gomes
Professor Auxiliar da Faculdade de Engenharia da Universidade do Porto

O autor declara que a presente dissertação (ou relatório de projecto) é da sua exclusiva autoria e foi escrita sem qualquer apoio externo não explicitamente autorizado. Os resultados, ideias, parágrafos, ou outros extractos tomados de ou inspirados em trabalhos de outros autores, e demais referências bibliográficas usadas, são correctamente citados.

Autor - Nuno Miguel Neves de Abreu

Faculdade de Engenharia da Universidade do Porto

Resumo

Nesta dissertação realizou-se uma pesquisa bibliográfica sobre o trabalho existente acerca de decomposição de polígonos e sobre o cálculo do *NFP* de dois polígonos. Acrescentaram-se novas funcionalidades a uma biblioteca geométrica desenvolvida por investigadores da FEUP. Pretende-se implementar um método alternativo ao de deslizamento para calcular o *NFP* de polígonos simples. Implementou-se um algoritmo que efectua a triangulação de polígonos simples, criando um conjunto de triângulos cujos vértices estão sobre os vértices do polígono. Adicionámos um algoritmo de partição que, partindo dos triângulos formados anteriormente, cria o menor número de subpolígonos convexos, em que a reunião desses elementos dá origem ao polígono original e esses elementos têm de ser disjuntos. Através de uma simples alteração no algoritmo anterior, obtemos um algoritmo que permite fazer uma cobertura de um polígono simples, em que a reunião dos elementos dessa cobertura dá origem ao polígono original, não tendo estes de ser disjuntos. De forma a que a biblioteca consiga lidar com polígonos com "buracos" acrescentamos um algoritmo que divide o polígono conectando o contorno exterior ao interior. Na pesquisa bibliográfica consideramos diferentes estratégias para o cálculo do *NFP* e implementamos uma que lida com polígonos convexos, mais simples e eficiente que a implementada anteriormente na biblioteca geométrica, através da ordenação angular das arestas. Foi desenvolvido também um algoritmo que efectua a união de polígonos convexos sobrepostos, que permite detectar contornos interiores resultantes da sobreposição. A introdução destes algoritmos na biblioteca acrescenta um novo método de cálculo para o *NFP* global, alternativo ao de deslizamento, sendo obtido através da sobreposição dos *NFPs* convexos dos polígonos resultantes da decomposição convexa. São realizadas várias experiências computacionais usando várias instâncias descritas na literatura e outras geradas aleatoriamente. Os resultados são apresentados e discutidos ao longo da dissertação.

Para acrescentar as novas funcionalidades é modificada a estrutura de dados, concretamente de C para C++, com a introdução da *STL*. Para a apresentação do trabalho realizado é desenvolvida também uma interface *web* que permite ao utilizador realizar um conjunto de operações sobre problemas descritos em ficheiros em um formato de dados *standard*, o NestingXML.

Abstract

In this dissertation we do a bibliographic review on polygon decomposition and NFP calculation. We add new functionalities to the geometric library that was originally developed by researchers from FEUP. We implement an alternative method from the sliding approach in order to calculate the NFP. A simple polygon triangulation algorithm is implemented, followed by an algorithm that creates a partition from the triangles determined before. With minor changes to the algorithm we can also cover simple polygons. In order to deal with holes an algorithm will be implemented that divides the original polygon, establishing a connection between the outer and inner contours from the polygon. In the bibliographic research we considered different procedures for obtaining the NFP and we implemented one that deals with convex polygons, simpler and more efficient. We also developed an algorithm that merges overlapped convex polygons and detects inner contours, if available. A new method for calculating the NFP from simple polygons with holes is obtained by merging convex NFPs generated with the polygons that resulted from the polygon decomposition. A series of computational tests are done, using typical polygons gathered from the bibliographic research.

Agradecimentos

Agradeço aos meus pais pelo apoio que sempre prestaram.

Índice

Lista de figuras

Lista de tabelas

Abreviaturas e Símbolos

Lista de abreviaturas (ordenadas por ordem alfabética)

CAD	*Computer Aided Design (texto não português em itálico)*
CFI	Corte de Figuras Irregulares
ESICUP	*EURO Special Interest Group on Cutting and Packing*
FEUP	Faculdade de Engenharia da Universidade do Porto
GIS	*Geographic Information System*
HTML	*HyperText Markup Language*
IP	Invólucro de Posicionamento
NFP	*No Fit Polygon*
PHP	*Hypertext Preprocessor*
PNG	*Portable Network Graphics*
STL	*Standard Template Library*
SVG	*Scalable Vectorial Graphics*
XML	*Extensible Markup Language*

Capítulo 1

Introdução

1.1 - Âmbito da dissertação

Foi nos anos 70 que surgiu o campo da geometria computacional, preocupando-se com o estudo sistemático de algoritmos e estruturas de dados para resolver problemas geométricos. Hoje existe um grande conjunto de algoritmos que são eficientes e relativamente fáceis de perceber e implementar.

Para ilustrar a vasta área de aplicação da geometria computacional são apresentadas de seguida algumas dessas áreas e os seus problemas:

- Computação gráfica: refere-se à criação de imagens de objectos modelados para representação no monitor ou em qualquer dispositivo de saída. Podem ser desde linhas, polígonos até fontes de luz, texturas, etc. Como estas imagens representam objectos geométricos, os algoritmos geométricos têm um papel importante.
- Robótica: este campo estuda o design e a utilização dos robôs. Um dos problemas frequentes é o planeamento do movimento de um robô, em que se tenta definir um percurso num espaço com obstáculos.
- Sistemas de informação geográficos (GIS): este tipo de sistema armazena dados geográficos como o formato de países, altura de montanhas, percurso de rios, densidade populacional, ruas, linhas eléctricas, condutas de gás, etc. Podem ser usados para extrair informação sobre determinada região ou para estabelecer uma relação sobre diferentes tipos de dados. Por vezes a quantidade de dados é tanta que torna imperativo a utilização de algoritmos eficientes. Por exemplo, relativamente ao desenvolvimento de um sistema de navegação que informa o utilizador da sua posição, a cada momento é preciso determinar a posição do carro no mapa e seleccionar uma porção do mapa para representar no monitor, o que requer estruturas de dados eficientes.
- Desenho orientado por computador (CAD): preocupa-se com o *design* de produtos no computador. Entre os produtos encontram-se placas de circuito integrado, maquinas, mobília ou edifícios. Como em qualquer caso o produto é uma entidade geométrica

podem surgir muitos problemas geométricos: intersecções, união e decomposição de objectos, etc.

- Corte e empacotamento: este tipo de problemas é encontrado em muitas indústrias, divergindo em restrições e objectivos. Na indústria da madeira, vidro e papel preocupam-se essencialmente com o corte de figuras regulares enquanto na indústria naval, têxtil e da pele é com figuras irregulares. Os problemas de corte e empacotamento são problemas de optimização que se preocupam em encontrar um bom arranjo de múltiplas formas geométricas de forma a minimizar o desperdício de matéria-prima.

Este pequeno conjunto de problemas geométricos evidencia o papel que a geometria computacional tem em diferentes áreas da ciência computacional.

Os algoritmos investigados nesta dissertação estão relacionados com um tipo específico de problemas de corte e empacotamento: o posicionamento de formas irregulares (*nesting*). Os problemas de *nesting* em geral referem-se a posicionamento de um número de formas geométricas no interior de um determinado material, tipicamente rectangular (ver Figura 1), de forma a que não se sobreponham. O objectivo passa por maximizar a utilização desse material, ou seja minimizar o desperdício. Este tipo de problemas são também conhecidos como problemas de Posicionamento de Figuras Irregulares. Podem ser encontrados por exemplo no sector industrial aquando do corte de matéria-prima (tecido, pele, metal vidro, etc.).

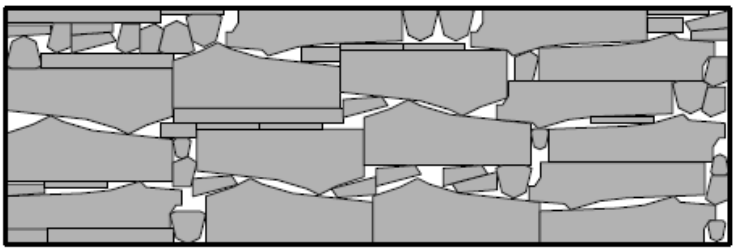

Figura 1 - Exemplo de um *layout* de um problema de posicionamento de formas irregulares.

É uma área relativamente inexplorada na medida em que não há muitas publicações científicas, o que se compreende dado o elevado custo temporal de desenvolvimento de soluções para problemas da área. Este tipo de problemas ocorre quando um qualquer processo de produção envolve o corte de peças a partir de determinada matéria-prima. Quando estão envolvidos componentes irregulares é introduzido um novo grau de complexidade através da geometria. É em torno destes cálculos geométricos que vai ser desenvolvida esta dissertação. As abordagens existentes variam de indústria para indústria, dependendo dos materiais cortados ou das máquinas que executam essas operações. Segundo Dowsland e Dowsland [1] todas estas abordagens têm um requisito comum, que consiste na capacidade de responder a questões de índole geométrica, como por exemplo se duas peças se sobrepõem ou qual é a distância que se pode mover uma peça até tocar outra. O *NFP* é de momento a principal abordagem relativamente à manipulação da geometria nos problemas de *nesting*. A grande vantagem dos *NFPs* é transformar comparações de polígono contra polígono em comparações de um polígono contra um ponto. Se todas as peças envolvidas forem convexas o *NFP* é fácil de calcular, mas se forem não convexas há um aumento na complexidade. Nestes problemas de posicionamento habitualmente apenas se consideram duas dimensões da matéria-prima, sendo formulados de duas maneiras diferentes. Na

primeira, a placa (ou o item maior) é um rectângulo de comprimento infinito e o que se pretende é posicionar os itens mais pequenos de forma a minimizar o comprimento do maior. Na segunda o item grande é fechado e limitado, como por exemplo a placa representada na Figura 1. Pretende-se agora maximizar a utilização do item grande através da maximização do número de itens pequenos posicionados. A restrição principal comum às duas formulações passa pela não sobreposição dos itens pequenos entre si e com a fronteira do item grande.

1.2 - Trabalho realizado

Nesta dissertação pretende-se acrescentar novas funcionalidades à biblioteca geométrica existente, eficiente e robusta, de geração de *NFPs* entre dois polígonos simples, para a resolução de problemas de Corte de Formas Irregulares (CFI).

Para a introdução de novas funcionalidades é essencial uma adaptação da estrutura de dados existente, o que obriga à conversão do código em C para C++ e a utilização da *Standard Template Library*.

Pretende-se implementar uma nova maneira de gerar os *NFPs* que permita eliminar as limitações da implementação corrente. Actualmente a biblioteca só consegue lidar com polígonos simples (ver Figura 2 b), isto é, se o polígono tiver "buracos"[1] (ver Figura 2 c) estes são ignorados. De forma a conseguir tratar polígonos com "buracos", uma das abordagens possíveis consiste na sua decomposição em polígonos simples. Para poder então utilizar a nova forma de criar *NFPs*, terá de ser possível introduzir novas funcionalidades que permitam a decomposição de polígonos simples em subpolígonos convexos (ver Figura 2 a) e a junção desses *NFPs* resultantes de forma a obter o *NFP* final.

Para a visualização dos resultados e dos dados que constituem o problema de posicionamento, guardados num ficheiro de formato NestingXML, foi desenvolvida uma interface *web*. Foi disponibilizado ao utilizador um conjunto de funções a aplicar sobre as peças seleccionadas, sendo estes dados passados ao programa que contém a biblioteca geométrica.

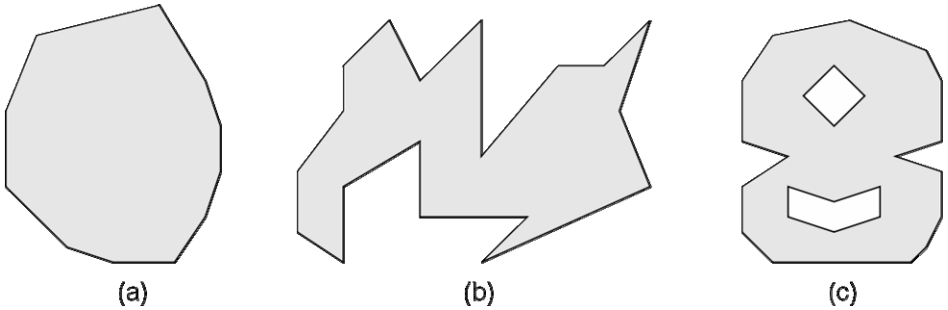

Figura 2 - Tipos de polígonos: (a) polígono convexo; (b) polígono simples; (c) polígono com "buracos".

1.3 - Organização da dissertação

Esta dissertação está organizada em seis capítulos. Neste capítulo é apresentado o tema em estudo e no seguinte é realizada uma pesquisa bibliográfica em busca de soluções que

[1] Um polígono com "buracos" é um polígono com pelo menos um contorno interior.

permitam abordar o problema proposto. No terceiro capítulo é descrita a interface *web* que serve de plataforma de demonstração deste trabalho, além de permitir a introdução de um novo formato de dados. No capítulo quatro são descritos os algoritmos implementados, justificando as opções seguidas, e apresentadas pequenas demonstrações. No capítulo cinco são realizadas algumas experiências computacionais com o objectivo de validar os algoritmos implementados e é realizado um teste comparativo entre duas formas distintas de calcular o *NFP*. Finalmente no capítulo seis é apresentado um resumo das conclusões mais relevantes desta dissertação, assim como possíveis funcionalidades a desenvolver no futuro.

Capítulo 2

Bibliotecas Geométricas para Problemas de Posicionamento de Figuras Irregulares

2.1 - Introdução

A biblioteca geométrica desenvolvida por investigadores da FEUP [2][3] contém um conjunto de algoritmos que permitem a manipulação de polígonos. Estão implementadas funções ao nível de arestas e pontos, permitindo por um lado calcular a posição relativa entre segmentos ou entre um segmento e um ponto. Existem também funções de baixo nível para cálculo de distâncias e áreas de polígonos. A partir das funções anteriores foram elaboradas funções ao nível dos polígonos das quais as principais são:

- A operação "fusão" que é aplicada sobre dois polígonos numa determinada posição relativa fixa com pelo menos um ponto de contacto (Figura 3);
- A criação do invólucro convexo de um polígono simples *P* que é definido como o invólucro convexo de área mínima que contém todos os pontos de *P* (Figura 4);
- A determinação do *NFP* de um polígono face a um segundo, que consiste no lugar geométrico dos pontos onde o ponto de referência[2] do segundo polígono (a sua origem) pode ser colocado de modo a que os dois polígonos fiquem em contacto. Se o ponto de referência do segundo polígono estiver dentro do NFP (Figura 5 a) então os dois polígonos sobrepõem-se, se estiver no exterior (Figura 5 b) estes não se sobrepõem nem se tocam e se estiver sobre o NFP (Figura 5 c), os dois polígonos estão em contacto.

[2] Na Figura 5 o ponto de referência do triângulo é o vértice superior.

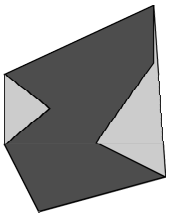

Figura 4 - Invólucro convexo de um polígono.

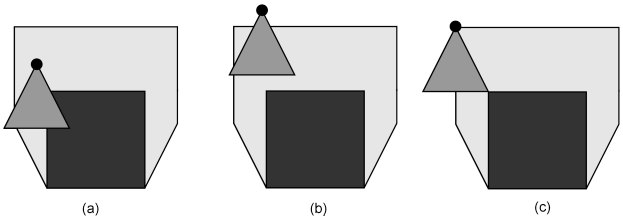

Figura 5 - Diferentes posicionamentos de um polígono.

Algumas das funções referidas têm limitações, muitas das quais já foram actualizadas por investigadores nesta área. O algoritmo utilizado para o cálculo da fusão entre polígonos ignora por exemplo a área interior livre entre os contactos de dois polígonos.

Quanto ao algoritmo de cálculo do *NFP* implementado, foi baseado na abordagem proposta por Mahadevan [4] e é conhecido como de deslizamento. A principal limitação é que ignora polígonos com "buracos" (Figura 2 c) e concavidades de polígonos simples cuja entrada da concavidade seja mais pequena que o polígono orbital (Figura 6). Ou seja, uma suposta área interior existente é ignorada.

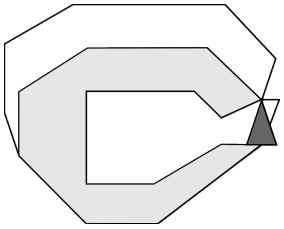

Figura 6 – Limitação do algoritmo de deslizamento.

Algumas destas limitações são impostas pela estrutura de dados implementada, que não contempla a existência de contornos interiores. Assim, será necessário repensar a estrutura de dados de forma a acomodar as novas funcionalidades.

2.2 - Outras Bibliotecas Geométricas

Nos últimos anos o campo da geometria computacional tem sido bastante dinâmico, produtivo e criativo, conquistando um papel importante em áreas como a ciência e a engenharia. Destaca-se um projecto que disponibiliza uma biblioteca geométrica: a CGAL [5].

A CGAL é uma biblioteca de algoritmos de geometria computacional. Foi desenvolvida com o objectivo de facilitar o acesso a algoritmos fiáveis, eficientes, reutilizáveis e robustos tanto a nível académico como industrial. Actualmente é desenvolvida por muitas instituições entre as quais se destacam o Instituto Max Planck para a Ciência Computacional (Alemanha), INRIA Sophia-Antipolis (França) e a Universidade de Tel-Aviv (Israel). Um aspecto que demonstra a flexibilidade destes algoritmos é o facto de poderem ser facilmente adaptados para utilizarem estruturas de dados de aplicações já existentes. Uma das principais

características da CGAL é a possibilidade de recorrer à computação exacta, favorecendo a robustez em detrimento da rapidez. O seu carácter genérico contribuiu para a sua adopção em grande escala, com clientes em várias partes do mundo e em diferentes domínios de aplicação.

2.3 - Revisão bibliográfica

É apresentada de seguida uma revisão bibliográfica dos principais trabalhos sobre a decomposição de polígonos e também sobre formas alternativas à existente na biblioteca geométrica, que serve de base a este trabalho, para o cálculo do *NFP*.

2.3.1 – Decomposição de Polígonos

Estruturas geométricas complexas são mais fáceis de tratar quando decompostas em estruturas mais simples. Por isso é que a decomposição dos polígonos é uma ferramenta importante para muitos algoritmos geométricos mais complexos.

O problema a considerar é:

Dado um polígono simples P, qual é o número mínimo de polígonos convexos que formam uma partição de P?

Este problema é denominado de "decomposição óptima convexa de polígonos".

Um problema clássico na área da decomposição de polígonos é a triangulação, onde o interior do polígono é completamente dividido em triângulos (figura geométrica convexa mais simples de tratar). Têm muitas aplicações entre as quais a resolução de problemas acerca do ponto mais próximo, cálculo de áreas, visibilidades e caminhos internos.

Um dos objectivos associados à decomposição é a criação do menor número de subpolígonos, portanto a solução não pode passar só pela triangulação, uma vez que todas as triangulações de um polígono produzem o mesmo número de triângulos (a triangulação de um polígono *P* com *n* vértices produz *n-2* triângulos). Outro dos objectivos é a rapidez do algoritmo, o que está em conflito com o primeiro. É então necessário arranjar um compromisso entre os dois.

A heurística proposta por Hertel e Mehlhorn [6] para a decomposição convexa é bastante simples e eficiente. Consiste em traçar diagonais, que não se intersectem, entre os vértices de um polígono, triangulando-o (Figura 7). Depois são eliminadas as diagonais (não essenciais) que permitem a obtenção de subpolígonos convexos, ou seja, diagonais que não produzem ângulos internos maiores que 180º. A verificação da convexidade é realizada localmente sobre as arestas que constituem o elemento da partição (subpolígono convexo) analisado. Neste caso diferentes triangulações e diferentes ordens de remoção de diagonais podem produzir melhores decomposições.

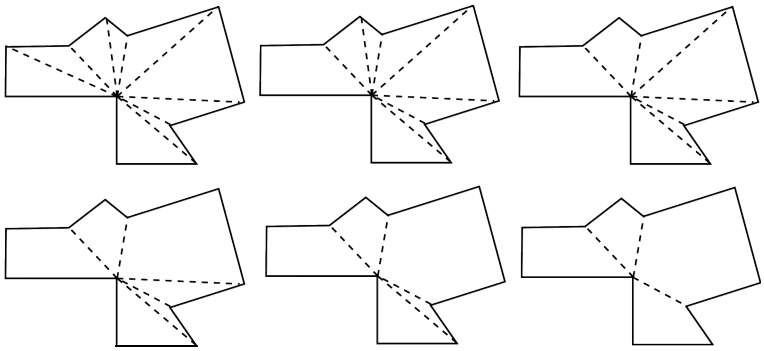

Figura 7 - Remoção de diagonais não essenciais

Há três teoremas associados com o algoritmo de Hertel-Mehlhorn:

Teorema 1: *Podem existir no máximo duas diagonais essenciais para qualquer vértice côncavo.*

Na Figura 8 estão representadas duas arestas de um polígono ((*v-*,*v*) e (*v*,*v+*)) e três diagonais (*a*, *b* e *c*). Podemos verificar que os dois semiplanos (*H-*, definido pela aresta (*v-*,*v*) e pela sua projecção, e *H+*, definido pela aresta (*v*,*v+*) e pela sua projecção) partilham a área delimitada pelas linhas a tracejado (projecções das arestas) e que existem diagonais em cada uma das áreas. Prova-se ao observar-se que só pode existir uma diagonal essencial em cada semiplano em torno do vértice côncavo. A diagonal *a* não é essencial porque *b* também está em H+. De forma semelhante *c* também não é essencial.

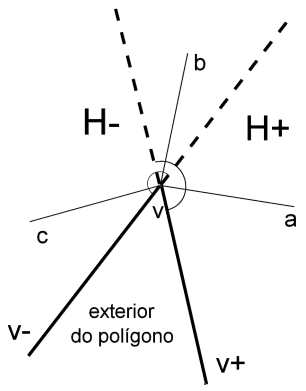

Figura 8 - Diagonais essenciais [7].

Teorema 2 (Chazelle e Dobkin [8])*: Seja Φ o número de polígonos convexos em que um polígono P pode ser decomposto. Para um polígono com r vértices côncavos [r/2]+1<= Φ<=r+1.*

A inserção de segmentos que intersectam cada ângulo côncavo torna-os convexos, resultando numa partição convexa (Figura 9).

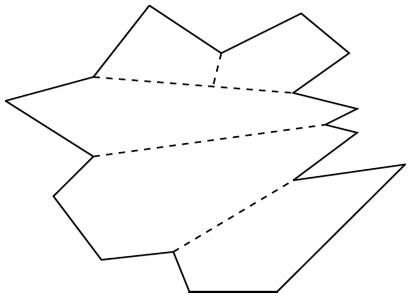

Figura 9 - Polígonos convexos resultantes de uma decomposição:[r/2]+1 polígonos convexos; r=7; 5 polígonos [6].

Teorema 3: *Seja P um polígono simples e U a triangulação interior de P. Então uma decomposição convexa de P produz no máximo quatro vezes o número óptimo de polígonos convexos.*

Quando o algoritmo acaba, cada diagonal existente é essencial para os ângulos côncavos. Segundo o teorema 1, cada vértice deste tipo tem no máximo duas diagonais essenciais associadas. Então, o número de diagonais essenciais é no máximo *2r* em que *r* é o número de ângulos côncavos. Sendo assim, o número de polígonos convexos *M* produzidos pelo algoritmo satisfaz $2r+1 \geq M$. Segundo o teorema 2 de Chazelle $\Phi \geq [r/2]+1$, portanto $4\Phi \geq 2r+4 > 2r+1 > M$.

Este algoritmo decompõe polígonos simples em convexos de forma rápida através da inclusão de diagonais, com um número máximo de subpolígonos limitado.

Outra questão prende-se com o uso de pontos adicionais (de Steiner, ou novos vértices). A decomposição pode ser feita exclusivamente unindo vértices existentes ou então criando vértices adicionais de forma a que esta seja optimizada. Este último caso pode resultar num número inferior de peças mas envolve algoritmos mais complexos. A desvantagem principal da sua introdução é o facto de tornarem as representações mais complicadas e causarem erros de arredondamento. Permitir pontos de Steiner torna o problema bastante diferente, uma vez que há infinitos locais possíveis para colocar estes pontos. Chazelle e Dobkin [8] elaboraram um algoritmo capaz de resolver este problema. Introduziram o conceito de X_k - *pattern* (ver Figura 10) que consiste em conectar *k* vértices côncavos removendo-os (deixam de ser côncavos) sem criar novos deste tipo. Este tipo de ligação é estabelecido criando um nó central que por sua vez se conecta a *k* vértices. Uma decomposição utilizando este método, com *p* nós em conjunto com o método de decomposição Ingénua para remover vértices côncavos pode criar até *r+1-p* subpolígonos convexos. Desta forma os autores demonstram que, com a introdução de mais nós, é minimizado o número de polígonos resultante.

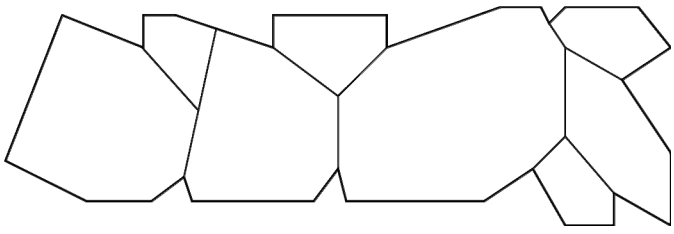

Figura 10- Decomposição-x com nós conectando 3 e 4 vértices côncavos [8].

Considera-se como decomposição Ingénua a decomposição onde os polígonos são obtidos através da inserção de segmentos entre as projecções das arestas que produzem os ângulos côncavos (ver Figura 11). Cada segmento tem sempre de conter um vértice côncavo.

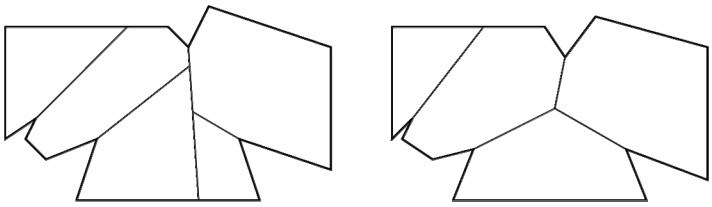

Figura 11 - Decomposição Ingénua e uma melhoria [8].

Teorema 1: *Qualquer decomposição Ingénua de um polígono P produz exactamente r+1 subpolígonos convexos.*

Teorema 2: *A classe de decomposições-X contém sempre uma decomposição convexa óptima.*

Teorema 3: *Uma decomposição-X com p nós tem pelo menos r+1-p subpolígonos convexos.*

Os autores concluem que o algoritmo é linear no número de vértices não côncavos e cúbico no número de vértices côncavos. A complexidade do algoritmo é aceitável mas infelizmente é demasiado elaborado e admitem que a sua implementação é "uma tarefa formidável". Propõem então sacrificar a eficiência favorecendo a simplicidade. Mesmo a decomposição Ingénua é um método válido se for aceitável não obter a solução óptima, que neste caso seria no máximo o dobro dos polígonos convexos.

Keil [9] também apresentou um algoritmo não só capaz de decompor um polígono em trapézios e estes em triângulos, mas permite também a existência de um contorno interior. Durante a decomposição é feita uma classificação dos vértices, sendo verificado se alguma aresta adjacente é horizontal:

- Se nenhuma das arestas é horizontal classifica o vértice como de início/separação, contínuo ou de fim/conecção.
- Se pelo menos uma das arestas adjacentes ao vértice é horizontal classificá-lo como de início/fim, contínuo ou de fim/conecção, examinando os trapézios criados até ao momento.

O vértice de inicio/separação é usado para formar um novo trapézio ou para dividir um trapézio existente. Os contínuos são usados para completar um trapézio existente ou formar um novo. Os de fim/conecção são usados para completar um trapézio ou para combinar um par destes.

Quando um polígono contém contornos interiores, este método insere uma diagonal entre o contorno exterior e o contorno interior de forma a criar um só contorno. As arestas do contorno interior são ordenadas de forma a terem o sentido oposto às do contorno exterior. Os "buracos" são removidos duplicando os vértices e reconectando diagonais que entram e saem desses vértices (ver Figura 12 e Figura 13). Se o polígono tiver arestas que se intersectem, criando um ponto múltiplo, a análise do polígono é interrompida.

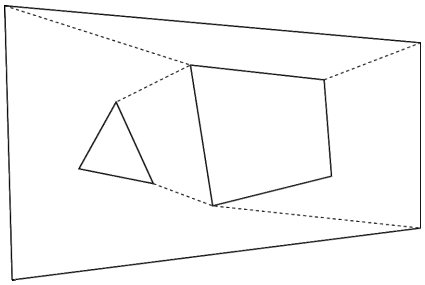

Figura 12 - Polígono com "buracos" [9].

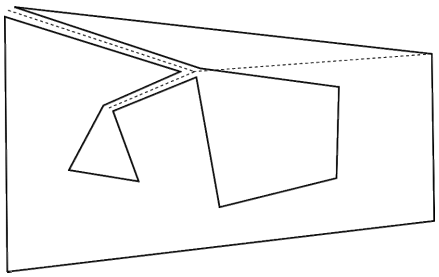

Figura 13 - Polígono anterior com os "buracos" removidos [9].

Relativamente à decomposição de polígonos ainda existe outra opção, a cobertura. Enquanto que a partição de um polígono envolve dividir o interior deste em subpolígonos não sobrepostos, a cobertura permite a existência de tal sobreposição. O número de polígonos gerados pela cobertura pode ser igual ou inferior ao da partição.

Cohen-Or et al [10] desenvolveram um algoritmo que, para um polígono simples *P*, determina a cobertura interna por polígonos convexos grandes. O algoritmo baseia-se numa partição inicial de *P*, criando um conjunto *C* de subpolígonos convexos não sobrepostos. É então construído um conjunto de subpolígonos convexos contidos em *P* através da construção do invólucro convexo de elementos do conjunto *C*. O conjunto de subpolígonos gerados por este algoritmo satisfaz os seguintes requisitos:

- O conjunto é pequeno;
- Cada elemento do conjunto é convexo;
- Cada polígono na cobertura deve ocupar a maior área de *P* possível;
- Cada elemento está contido em *P*.

Adicionalmente pode ser gerada uma cobertura parcial de uma percentagem pré-especificada do polígono. Como já foi referido, o algoritmo começa por simplificar o polígono simples de forma a facilitar a geração de coberturas parciais. Em seguida é feita a sua decomposição em pequenos subpolígonos formando os blocos elementares da cobertura. Unindo alguns destes elementos (enquanto o polígono resultante da união se mantiver convexo) dá origem a uma cobertura parcial. Os autores fornecem um conjunto de definições formais importantes:

Partição: Um conjunto $\{P_1, P_2, \ldots, P_k\}$ de polígonos é uma partição de P se:
$\forall\, i = 1\ldots k,\ P_i \subset P,\ \cup^{k}_{i=1} P_i = P$ **e** $P_i \cap P_j = \varnothing$.

Cobertura: Um conjunto $\{P_1, P_2, \ldots, P_k\}$ de polígonos é uma cobertura de *P* se:
$\forall\, i = 1\ldots k,\ P_i \subset P,\ \cup^{k}_{i=1} P_i = P.$ Aqui os polígonos não necessitam de ser disjuntos.

Cobertura interna: Dado $0 \leq p \leq 1$, um conjunto $\{P_1, P_2, \ldots, P_k\}$ de polígonos é uma p-cobertura interna de P se: $\forall i = 1 \ldots k, P_i \subset P, area(P) \geq area(\cup^{k}_{i=1} P_i) \geq p * area(P)$. Além dos polígonos não necessitarem de ser disjuntos, a sua união pode só cobrir P parcialmente.

Partição/Cobertura/Cobertura interna convexas: Um conjunto $\{P_1, P_2, \ldots, P_k\}$ de polígonos, sejam partições, coberturas ou coberturas internas, são denominadas convexas se: $\forall i = 1 \ldots k$, P_i é convexo.

O algoritmo desenvolvido por Cohen-Or et al [10] é baseado em três ideias principais (ver Figura 14):

- A recta definida pela aresta que contém um vértice côncavo divide o ângulo correspondente em dois ângulos convexos;
- O maior polígono convexo contido em P tem de ser adjacente a pelo menos um vértice côncavo.
- Normalmente as arestas de um subpolígono convexo grande contido em P coincidem com as arestas de P. Os Cohen-Or et al demonstraram mais tarde que este caso nem sempre se verifica.

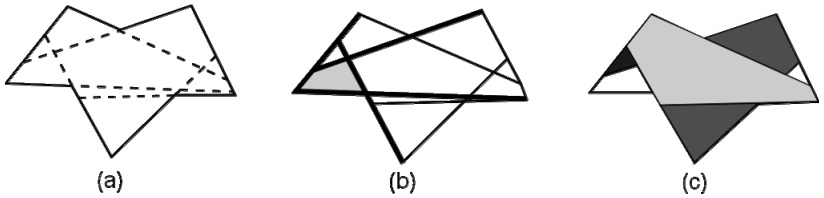

Figura 14 - Cobertura de um polígono simples [10]: (a) polígono é decomposto pelo método de projecção das arestas nos vértices côncavos; (b) A partição com maior potencial é definida pela soma dos comprimentos das arestas adjacentes; (c) Uma cobertura interna.

O algoritmo consiste em cinco fases distintas:

- Simplificação: O polígono de entrada é simplificado de forma a diminuir o número de vértices (côncavos) resultando numa diminuição da complexidade. Todos os vértices têm um triângulo associado com os seus vizinhos adjacentes e quando a área deste é menor que um valor predefinido o vértice corrente é removido.
- Geração dos segmentos de divisão do polígono: Através do método de prolongamento das arestas associadas com os ângulos côncavos até intersectarem um ponto de P (ponto adicional, de Steiner) é iniciada a sua partição (só introduz as diagonais).
- Construção das partições: Através das diagonais criadas na fase anterior, são criadas as partições convexas. Estes são os blocos elementares da cobertura.
- Produção da cobertura: Começando com uma cobertura nula, o algoritmo procura uma face (partição), numa lista ordenada, capaz de combinar com a face actual a analisar. Constrói-se o invólucro convexo deste arranjo e verifica-se se está inteiramente contido em P. Se sim é actualizada a cobertura, se não é formada uma cobertura só com a face actual.
- Alargamento: Na fase anterior só são guardadas as faces que constituem cada cobertura, portanto é necessário formar o subpolígono que constituem. Devido à construção do algoritmo é realizado novo ciclo de teste de expansão da cada cobertura dado que nem todas as faces são testadas anteriormente (as faces são

ordenadas pela área logo as mais pequenas podem não ter sido testadas contra faces adjacentes).

Os autores usaram os algoritmos geométricos da biblioteca CGAL. Demonstraram que geralmente são formadas coberturas interiores eficazes. Contudo existem casos em que o resultado não é óptimo relativamente ao número de subpolígonos da cobertura (ver Figura 15).

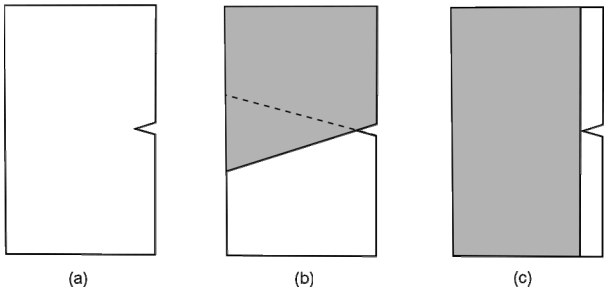

Figura 15 – Diferentes coberturas de um polígono: (a) o polígono; (b) a maior cobertura interna obtida pelo algoritmo; (c) a maior cobertura interna que é possível obter.

No exemplo da Figura 15, o algoritmo desenvolvido por Cohen-Or et al [10] não obtém a maior cobertura interna. No entanto a maior cobertura não cobre o polígono inteiro.

2.3.2 - Determinação do *NFP*

Mahadevan [4] propôs um algoritmo conhecido como de deslizamento que modeliza o movimento de um polígono orbital em torno de um polígono estacionário (ver Figura 16). Denomina-se o primeiro polígono como orbital dado que descreve uma trajectória orbital em torno do segundo, fixo. De forma a que não exista sobreposição de polígonos, no inicio do algoritmo, o ponto com maior ordenada do polígono orbital é posicionado de forma a estar em contacto com o ponto do polígono estacionário com menor ordenada. Os vértices seguintes são definidos na direcção anti-horária identificando qual a combinação vértice-aresta em que ocorre o deslizamento e a distância que é possível deslizar. Estes são determinados através da função-D. Esta função, proposta por Konopasek [11], devolve a posição relativa de um ponto em relação a uma aresta orientada. Consiste num rearranjo da equação de distância de um ponto a uma recta. Devolve -1 quando o ponto está à esquerda da recta de suporte da aresta, 1 quando está à direita e 0 quando está sobre a recta.

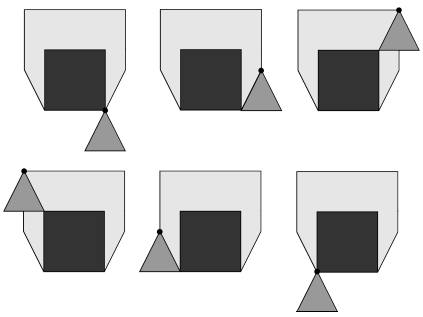

Figura 16 - Determinação do NFP segundo o método de deslizamento orbital.

Quanto ao método alternativo para o cálculo do *NFP*, Cuninghame-Green [12] propôs um novo método. Apesar de ser limitado a figuras convexas tem a vantagem de ser muito simples

e rápido de calcular. Na Figura 17 está ilustrado o funcionamento deste algoritmo. Vamos assumir que no cálculo do *NFP* o polígono *A* é o estacionário e o *B* é o orbital, movendo-se no sentido anti-horário. As arestas do *NFP* serão as de *A* no sentido anti-horário e as de *B* no sentido horário. Portanto, o passo inicial será ordenar os pontos de cada polígono nesse sentido. Todas as arestas são colocadas na origem (0,0) mantendo a sua direcção original e magnitude. Começando num ponto arbitrário, as arestas (orientadas) são ordenadas segundo o seu ângulo. O *NFP* constrói-se inserindo aresta a aresta segundo a ordem determinada, mantendo a sua magnitude e direcção. Isto só é válido para polígonos convexos.

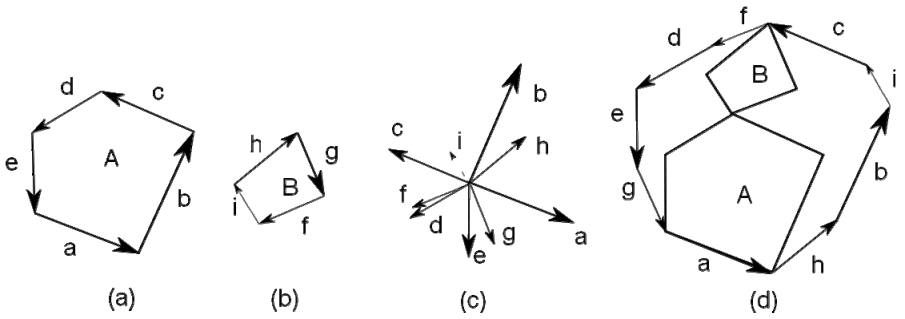

Figura 17 - Geração do *NFP* entre polígonos convexos: (a) polígono estacionário; (b) polígono orbital; (c) arestas ordenadas pelo seu ângulo; (d) *NFP* [13].

Agarwal et al [14] propõem um algoritmo semelhante (uma evolução do último) para construir *NFPs* para polígonos simples. Como entrada para o seu algoritmo têm dois polígonos simples *P* e *Q*, com m e n vértices respectivamente. O método consiste em três fases:

- Decompor P nos seus subpolígonos convexos P_1, P_2, …, P_m *e* *Q* nos seus subpolígonos convexos Q_1, Q_2, …, Q_n;
- Calcular a diferença de Minkowski[3] ($P_i \oplus -Q_j$, $\forall\, i = 1…m$, $\forall\, i = 1…n$) entre os subpolígonos de *P* e os de *Q* (este ponto corresponde ao método de ordenação das arestas referido anteriormente, são criados *NFPs* entre os vários subpolígonos convexos);
- União de todos os polígonos construídos no ponto anterior.

Apesar da decomposição simplificar a criação do *NFP* também gera dois problemas: a decomposição eficiente e a recombinação robusta dos *NFPs* resultantes. Agarwal et al [14] concluíram que a operação de recombinação dos polígonos é demasiado pesada computacionalmente, com tempos de execução elevados. Apesar deste facto, a decomposição continua atractiva porque não é necessário detectar contornos interiores.

Bennell e Song [15] apresentaram um algoritmo que mantém as concavidades (não decompõe o polígono) mas divide um dos polígonos em grupos de arestas sequenciais, distinguindo as côncavas das convexas. Cada grupo pode então ser individualmente representado no diagrama angular e depois ligado. Este algoritmo permite a detecção de contornos interiores, através de um simples teste que determina se dois polígonos simples se

[3] A soma de Minkowski é um método de adição vectorial. Sendo *A* e *B* conjuntos vectoriais, a soma de Minkowski dos dois é definida como: $A \oplus B = \{a + b : a \in A, b \in B\}$. Alguns investigadores [20][21] mostraram que $A \oplus -B$, conhecido como a diferença de Minkowski de A e B , é equivalente ao *NFP*.

sobrepõem quando o ponto de referência do polígono orbital é colocado sobre um vértice do contorno interior do polígono estacionário.

Capítulo 3

Utilização da Biblioteca Geométrica

3.1 – Introdução

Neste capítulo iremos apresentar um formato *standard* para a representação de problemas de posicionamento de formas irregulares e troca de dados entre investigadores desta área. Será descrita a interface da biblioteca geométrica com o utilizador e a estrutura de dados utilizada no programa. A interface *web* foi implementada como plataforma de demonstração das funcionalidades da biblioteca, através da representação gráfica das soluções obtidas, sendo também um meio de acesso e representação da informação contida nos ficheiros do referido formato *standard*.

3.2 – O Formato Nesting.XML

A estrutura essencial onde está reunida toda a informação utilizada pela interface e pela biblioteca geométrica é um ficheiro denominado por NestingXML. É um ficheiro XML que contém toda a informação necessária para os problemas de posicionamento de formas irregulares, desde as estruturas básicas como as placas e peças até às soluções (ver Figura 18).

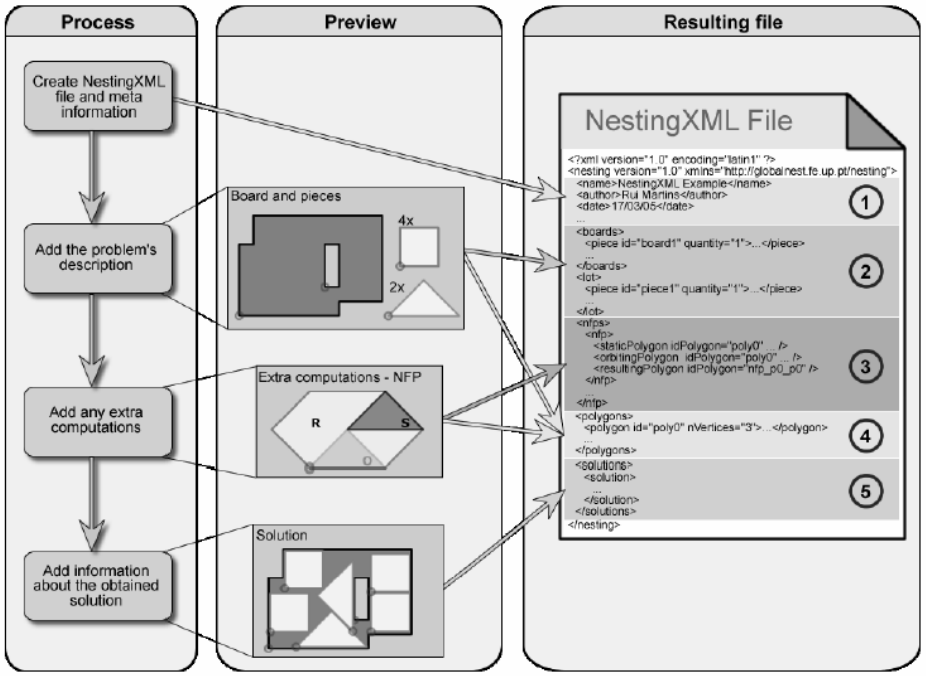

Figura 18 - Formato Nesting.XML [16].

Martins et al [16] desenvolveram o formato para a representação dos dados relacionados com problemas de *nesting*, tendo este sido aceite pelo ESICUP [17]. Foi escolhido um formato standard (XML) de forma a permitir a utilização de uma estrutura de dados única entre a comunidade de investigadores nesta área. Sendo assim, este formato permite:

- Representar tanto instâncias de problemas simples como complexas num formato comum;
- Armazenar tanto o problema de *nesting*, como as soluções alcançadas, e operações intermédias no mesmo ficheiro;
- É um formato textual simples e flexível, tornando-o fácil de ler;
- Como é escrito em XML é facilmente extensível.

O formato NestingXML, especificado em [18], está estruturado em cinco secções:

1. Informação sobre o autor e breve descrição do problema;

```
<name>Albano</name>
<author>ALBANO/SAPPUPO</author>
<date>2006/06/13</date>
<description>
Data set from the textile industry, scanned by E. Hopper from sample layout in Albano A. and Sappupo G., 1980, "Optimal Allocation of two-dimensional irregular shapes using heuristic search methods", IEEE Transactions on Systems, Man and Cybernetics, SMC-10, 242-248.
</description>
<verticesOrientation>clockwise</verticesOrientation>
<coordinatesOrigin>up-left</coordinatesOrigin>
```

2. Descrição da instância do problema;

```
<problem>
    <boards>
        <piece id="board0" quantity="1">
            <component idPolygon="polygon0" type="0" xOffset="0" yOffset="0" />
        </piece>
    </boards>
    <lot>
        <piece id="piece0" quantity="2">
            <orientation>
                <enumeration angle="0" />
                <enumeration angle="180" />
            </orientation>
            <component idPolygon="polygon1" type="0" xOffset="0" yOffset="0" />
        </piece>
        <piece id="piece1" quantity="2">
            …
        </piece>
</lot>
</problem>
```

3. Heurísticas, dados criados por operações intermédias e informação extra;

```
<NFPs>
    <NFP>
<staticPolygon angle="0" idPolygon="polygon1" mirror="none" />
        <orbitingPolygon angle="0" idPolygon="polygon1" mirror="none" />
        <resultingPolygon idPolygon="NFPPolygon0" />
    </NFP>
    …
</NFPs>
```

4. Descrição geométrica de todos os polígonos usados;

```
<polygons>
    <polygon id="polygon0" nVertices="4">
        <lines>
            <segment n="1" x0="  0.0" x1="29000.0" y0="  0.0" y1="  0.0" />
            <segment n="2" x0="29000.0" x1="29000.0" y0="  0.0" y1="4900.0" />
            <segment n="3" x0="29000.0" x1="  0.0" y0="4900.0" y1="4900.0" />
            <segment n="4" x0="  0.0" x1="  0.0" y0="4900.0" y1="  0.0" />
        </lines>
        <xMin>0</xMin>
        <xMax>29000</xMax>
        <yMin>0</yMin>
        <yMax>4900</yMax>
    </polygon>
    …
</polygons>
```

5. Soluções para o problema;

```
<solutions>
    <solution>
        <placement angle="180.0" boardNumber="1" idBoard="board0" idPiece="piece0"
mirror="none" x="9957.406" y="3613.154" />
        <placement angle="0.0" boardNumber="1" idBoard="board0" idPiece="piece1" mirror="none"
x="0.0" y="261.0" />
        <placement angle="0.0" boardNumber="1" idBoard="board0" idPiece="piece2" mirror="none"
x="0.0" y="2392.725" />
        …
<extraInfo>
            <Algorithm>SAHA</Algorithm>
            <Reference>
Solving Irregular Strip Packing problems by hybridising simulated annealing and linear programming,
European Journal of Operational Research, Volume 171 (3), 2006, Pages 811-829
</Reference>
            <Authors>A. Miguel Gomes, Jose F. Oliveira</Authors>
            <solutionWidth>9957.406</solutionWidth>
            <CPU>Pentium IV 2.4 Ghz and 512 Mb RAM</CPU>
                <solutionTime>2257.0</solutionTime>
                <firstSolutionTime>N/A</firstSolutionTime>
        </extraInfo>
    …
    </solution>
</solutions>
```

Desta forma é possível adicionar novos problemas, manter uma listagem de soluções obtidas para esses problemas e adicionar novos campos ao formato.

3.3 – Interface *Web*

A biblioteca que serviu de base a este trabalho dispunha de uma interface gráfica para a representação dos resultados via linha de comandos, sendo esta também utilizada para a introdução dos dados de entrada. Desde então foram desenvolvidas novas linguagens de programação que permitem facilitar o desenvolvimento e a utilização de determinado programa. Estas permitem construir novas interfaces onde tipicamente o utilizador interage com controlos gráficos de forma a realizar um conjunto de acções predefinidas sobre o tipo de dados utilizados. A grande vantagem é a utilização de um *browser* dado que é uma ferramenta que o utilizador já conhece e é disponibilizada na maioria dos computadores. O desenvolvimento desta plataforma tem o objectivo principal de facilitar a visualização e interpretação dos dados que queremos manipular, assim como dos dados criados durante a execução do programa que contém a biblioteca geométrica a testar.

Basicamente, o que se pretende é que seja disponibilizado ao utilizador um conjunto de dados cabendo a este seleccionar o conjunto sobre os quais quer executar determinada função. São depois representados os dados produzidos pelo programa, sendo possível ao utilizador guardar ou eliminar tais dados. A interface está estruturada em duas camadas: a de processamento de dados e a de visualização.

A camada de processamento de dados tem a função de transformar os dados de entrada, que neste caso são ficheiros *XML*, em dados que possam ser manipulados pelo utilizador e também de criar uma representação gráfica desses dados. Esta transformação é realizada

pelo *Expat XML parser*[4] escrito em *PHP*. Este "vê" o código *XML* como uma série de eventos e quando um determinado evento ocorre, chama a função adequada. Esses eventos correspondem às *tags*[5] que definem tipos ou categorias de informação e a própria informação em si. É um *parser* rápido e pequeno que não valida o ficheiro *XML* a analisar, isto é, ignora o documento que contem a definição do tipo de dados contidos no *XML*. Posteriormente, parte dos dados são convertidos em *SVG (Scalable Vectorial Graphics)* para facilitar a visualização dos dados ao utilizador. O *SVG* é uma linguagem baseada em *XML* usada para descrever gráficos de forma vectorial. Uma das principais vantagens deste tipo de gráficos é que não perdem qualidade ao serem ampliados. Para a criação das imagens de pré-visualização, estes *SVG's* são convertidos em *PNG's* chamando o *ImageMagick*[6]. De notar que para criar estas imagens de pré-visualização foi necessário ajustar os *SVG's* de forma a que ficassem todos à escala do maior, caso contrário, durante a conversão do formato, seria ignorado o tamanho relativo entre as figuras. Dado que o *SVG* é baseado em *XML*, ou seja, é uma linguagem textual, esse redimensionamento foi realizado lendo cada polígono representado, determinando o maior (excluindo a placa onde estão posicionadas as peças mais pequenas), e, consoante a relação da resolução de cada polígono face ao maior determinado, é passada, ao *ImageMagick* a resolução adequada. Toda esta camada foi escrita em *PHP*. Todas as imagens neste documento foram originalmente *SVG's*, sendo convertidas posteriormente em *PNG's* com mínima perda de qualidade, sendo óbvia a elevada qualidade visual.

A camada de visualização consiste no resultado final enviado para o *web browser*. É constituída por código *HTML* encarregue da representação do texto e formatação da interface e vários *scripts*[7] que produzem o resultado final visto pelo utilizador. Foi utilizada a *framework* de apresentação *Smarty* (ferramenta de compilação de *templates* para *PHP*), que tem como objectivo principal automatizar determinadas tarefas relacionadas com a camada de visualização da aplicação (criação de tabelas, introdução de dados), permitindo o uso de *templates* que separam o código *PHP* do *HTML*. Para o seccionamento do conteúdo foi utilizado um elemento do *HTML*, conhecido por *IFrame*, que possibilita a introdução em cada uma de código *HTML* específico. Também se recorreu ao *Javascript*, encarregue de orientar o conteúdo a ser visualizado para cada *IFrame*. São utilizadas *CSS's* (*Cascading Style Sheets*) que é uma linguagem de estilo utilizada para definir a apresentação de documentos escritos em *HTML*, isto é, aplica determinada formatação a um elemento específico do *HTML*.

De forma a poder representar os resultados das operações seleccionadas pelo utilizador, o programa vai comunicar com a interface dando a conhecer a referência do ficheiro temporário onde estão guardados os resultados. Estes dados vão ser interpretados pelo *parser* e serão apresentados ao utilizador.

Esta interface web para a biblioteca geométrica está disponível em [19].

---

[4] Um *parser* divide um conjunto de dados em elementos mais pequenos para que possam ser interpretados.

[5] Em *XML* uma *tag* é o que está escrito dentro de parênteses angulares (< >).

[6] O *ImageMagick* é uma aplicação que permite a manipulação e representação de imagens, suportando perto de 100 formatos diferentes.

[7] Um *script* é um conjunto de instruções que podem ser executadas pelo computador.

3.4 – Estrutura de dados

Um dos objectivos desta dissertação é acrescentar novas funcionalidades a partir de novas estruturas de dados. Para isso vai ser estudado o recurso à *STL* (*Standard Template Library*). A *STL* é uma colecção de classes, funções e algoritmos que disponibiliza uma *framework*[8] eficiente, simples e extensível para o desenvolvimento de aplicações. Oferece um sofisticado nível de abstracção que promove o uso de estruturas de dados e algoritmos genéricos sem perda de eficiência. O uso de *STL* pretende oferecer os seguintes benefícios:

- É um standard: Os programadores que modificarem o código não perdem muito tempo a decifrá-lo;
- Reutilização de código: Como é baseado em *templates*, as classes da *STL* podem ser adaptadas a tipos distintos sem mudança de funcionalidade;
- Portabilidade e flexibilidade: Como partilham a mesma convenção pode-se optar pelo que apresenta melhor performance com pouca alteração de código;
- Facilidade de uso.

A *STL* contém *containers*, um objecto que guarda uma colecção de outros objectos. Gere dinamicamente a memória necessária para os seus elementos e disponibiliza funções que permitem aceder a estes, directamente ou através de iteradores (objectos com propriedades semelhantes aos apontadores). Muitos *containers* partilham algumas das referidas funções. A decisão acerca do tipo a ser utilizado não depende unicamente das funcionalidades disponibilizadas mas também da eficiência (ou complexidade) das operações a realizar. Isto verifica-se especialmente nos *sequence containers*, oferecendo diferentes compromissos de eficiência entre inserir/remover elementos e o acesso aos mesmos:

- *vector*: elementos organizados na forma de um *array* que pode crescer dinamicamente;
- *list*: elementos organizados na forma de uma lista duplamente encadeada;
- *deque*: elementos organizados em sequência, permitindo inserção ou remoção no início ou no fim sem necessidade de movimentação de outros elementos.

Tabela 1 - Comparação de eficiência entre as estruturas representadas.

Operações	C array	Vector	Deque	List
inserir/remover no inicio	-	linear	constante	constante
inserir/remover no fim	-	constante	constante	constante
inserir/remover no meio	-	linear	linear	constante
Aceder ao primeiro elemento	constante	constante	constante	constante
Aceder ao ultimo elemento	constante	constante	constante	constante
Aceder a elemento do meio	constante	constante	constante	linear
Custo	nenhum	baixo	médio	alto

[8] Uma *framework* é uma estrutura de suporte definida em que outro projecto de software pode ser organizado e desenvolvido.

Na Tabela 1 comparam-se as eficiências das principais operações sobre elementos de *containers* em função do número de elementos do *container*, em que "constante" significa que não depende do número de elementos e "linear" que cresce linearmente com o número de elementos. O "custo" refere-se ao custo adicional em processamento.

Tabela 2 - Teste comparativo de tempos de execução de inserção de elementos no inicio e no fim de containers.

Instrução	Vector	Deque	List
push_back integer	0,0020s	0,0015s	0,0043s
push_back 100 bytes	0,025s	0,016s	0,020s
push_back 1000 bytes	0,41s	0,16s	0,19s
push_back 1000 bytes, with reserve	0,11s	-	-
push_front integer	0,86s	0,0019s	0,0043s
push_front 100 bytes	42,5s	0,015s	0,021s

As *lists* são *containers* criados especificamente para serem eficientes a inserir e remover elementos em qualquer posição. *Vector* e *deque* são lineares a inserir/apagar um elemento no meio mas a *list* é constante. Enquanto que com o *vector* e *deque* o acesso a qualquer elemento é constante, com as listas é linear para um elemento do meio (o acesso aos elementos é iterativo). Por causa deste aspecto foi decidido eliminar a utilização da *list*. A opção entre *vector* e *deque* recai nesta última devido à maior eficiência na gestão da memória: os seus elementos não estão armazenados contiguamente (estão distribuídos em diversos segmentos). Podem "crescer" mais eficientemente que os vectores, com a capacidade gerida automaticamente, evitando grandes realocações de memória. Na Tabela 2 é realizada uma comparação de tempos de execução relativos a inserções de elementos no início e no fim de *containers*. Está demonstrada a vantagem da *deque* em relação ao *vector* neste conjunto de operações dado que mesmo sem reservar memória antecipadamente a *deque* consegue tempos na mesma ordem de grandeza que o *vector* (em que foi reservada memória). Devido à flexibilidade da *STL*, podemos facilmente trocar entre *vector* e *deque* (funções membro muito semelhantes), portanto esta escolha não é irreversível.

Concluindo, a *STL* pode obrigar a uma mudança no estilo de programação, permitindo que o código seja mais reutilizável, sólido e robusto. A sua implementação permite maior simplicidade sem perda de eficiência

.

Capítulo 4

Algoritmos Geométricos

4.1 – Introdução

O algoritmo seleccionado para fazer a operação de decomposição de polígonos é o de Hertel-Mehlhorn [6] pela sua rapidez e eficiência, mas também pela sua simplicidade. Dado que o método proposto é do tipo *divide and conquer* (dividir o polígono em triângulos e formar conjuntos a partir destes) consegue-se facilmente diminuir o número de subpolígonos convexos agrupando vários triângulos (polígono convexo mais simples), facto comum entre os algoritmos de partição e cobertura. Adicionalmente esta escolha permite obter algoritmos de partição e cobertura apenas pela modificação de uma restrição: o facto de poder conquistar triângulos já utilizados anteriormente (condição imposta pela cobertura mas inválida na partição).

Como forma de lidar com contornos interiores optou-se por uma abordagem semelhante à utilizada por Keil [9] que consiste na introdução de diagonais entre o contorno exterior e o interior. Foi introduzida uma alteração relativamente ao algoritmo original: as diagonais utilizam vértices adicionais, de forma a permitir a criação de diagonais horizontais. Esta modificação deve-se apenas a razões relacionadas com a implementação, dado ser menos complexo de executar computacionalmente e obter o mesmo resultado. Outra modificação introduzida foi a de dividir completamente os polígonos pelas diagonais inseridas entre o contorno exterior e cada contorno interior. O método de duplicação dos pontos que pode ser visualizado na Figura 13 foi considerado desnecessário quando a divisão do polígono composto pelo método a apresentar é mais simples e igualmente eficaz. As diferenças no resultado (número de polígonos simples) são irrelevantes dado que o objectivo é decompor o polígono composto em subpolígonos convexos, portanto a aplicação do algoritmo de Hertel-Mehlhorn a qualquer um deles vai na teoria obter resultados muito semelhantes (os triângulos obtidos da triangulação de cada polígono simples são guardados no mesmo conjunto).

O método apresentado por Cuninghame-Green [12] para a criação do *NFP* para figuras convexas é a alternativa conhecida ao algoritmo de deslizamento orbital, quando aliado aos algoritmos escolhidos de decomposição de polígonos com “buracos” em polígonos simples e

estes em convexos. Desta forma é garantido o tratamento de contornos interiores se existente, facto que era ignorado no algoritmo de posicionamento na biblioteca existente.

Neste capítulo vão ser detalhados os algoritmos implementados, nomeadamente de:

- Triangulação de polígonos simples;
- Partição e Cobertura de polígonos simples;
- Remoção de contornos interiores de polígonos simples;
- Cálculo de NFP de polígonos convexos;
- União de polígonos convexos.

4.2 – Triangulação

Neste algoritmo pretende-se dividir completamente um polígono simples num conjunto de triângulos. Como já foi referido, o algoritmo implementado foi baseado no de Hertel-Mehlhorn, que consiste na triangulação através da introdução de diagonais. Facilmente podemos concluir que este algoritmo vai devolver um conjunto de diagonais e triângulos, pelo que é importante definir adequadamente as suas estruturas. Uma diagonal é caracterizada por unir sempre dois vértices não consecutivos de um polígono, nunca intersectar outra diagonal ou aresta e está sempre associada a dois triângulos. Um triângulo é definido por três vértices e as suas arestas podem ser arestas do polígono ou diagonais, herdando as suas propriedades. Todas estas informações têm de ser guardadas em estruturas adequadas porque vão ser necessárias ao algoritmo seguinte: o de junção de triângulos. É importante referir que, por questões de eficiência, foi decidido utilizar, sempre que possível, referências para o índice do vértice no polígono (*int*[9]) e não as coordenadas deste (*double*[10]) sempre que seja necessário associar os pontos das estruturas.

O primeiro passo é a determinação de um ponto inicial válido. Uma diagonal é traçada sempre entre duas arestas adjacentes e tem de pertencer sempre ao interior do polígono, pelo que são estas que teremos que analisar. Seja *i* o índice de um dos vértices do polígono simples *P*, em que os pontos estejam distribuídos no sentido horário. Através das *funções D* (explicadas em 2.3.2 e implementadas anteriormente na biblioteca geométrica) testamos os três vértices que definem as duas arestas adjacentes (*i, i+1, i+2*) relativamente a uma "viragem" à direita (é uma maneira simples de verificar se o ângulo formado pelas duas arestas é convexo ou côncavo).

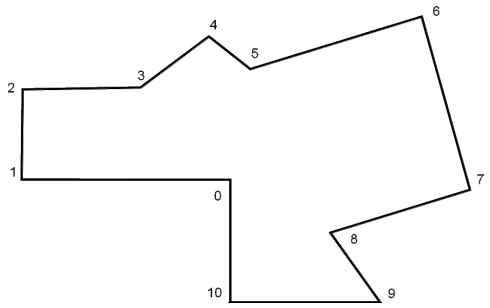

Figura 19- Polígono simples a triangular.

[9] É usado para definir números inteiros.

[10] É usado para definir números grandes em vírgula flutuante.

Se tal se não se verificar, avança-se para o próximo vértice e repete-se este passo. Se se verificar, passa-se para a próxima condição, que exige que uma diagonal pertença exclusivamente ao interior do polígono.

São novamente utilizadas as *funções D* para testar a existência de algum ponto do polígono dentro do triângulo a analisar. Caso exista, procura outro vértice. No caso contrário, a diagonal é válida, sendo esta e o triângulo que forma adicionados aos respectivos conjuntos. Dado que neste algoritmo não são permitidos pontos adicionais, existe a particularidade de estarmos sempre a analisar arestas, portanto podemos facilmente remover o triângulo formado através da remoção de um ponto exterior (ponto que une as duas arestas analisadas).

Este pormenor facilita a análise do polígono mas exige uma adaptação cuidada dos índices deste último (temporário), devido ao facto já referido de as diagonais e os triângulos estarem definidos através dos índices do polígono original.

Para concluir este algoritmo, são associadas as diagonais aos triângulos e vice-versa. Não esquecer que uma diagonal está sempre associada a dois triângulos e um triângulo está associado a uma, duas ou três diagonais.

Consideremos o polígono representado na Figura 19. Sendo *i=0*, em que *i* é o índice de um vértice do polígono, verificamos que o ponto *2 (i+2)* está à direita da aresta (0,1) e que não se encontra nenhum ponto do polígono dentro da área delimitada pelas arestas (0, 1), (1, 2) e (2, 0). A diagonal (2, 0) é válida logo é criada. Depois de criarmos o triângulo (0,1,2) podemos excluí-lo eliminando o vértice 1 da lista de vértices. Este procedimento elimina a hipótese de haver qualquer intersecção entre diagonais. Neste momento, o ponto inicial está no vértice 0 e como foi eliminado o 1, os pontos seguintes serão os vértices 2 e 3. Como as condições referidas anteriormente também se verificam neste caso, adicionamos a diagonal (3,0). Agora é eliminado o ponto 2 e continua a análise do polígono até restar um único triângulo. Se, por exemplo, começarmos a analisar o polígono no ponto *2 (i)*, verificamos que a diagonal traçada (4,2) está fora, logo é inválida, portanto são procurados os pontos seguintes que verificam as condições. O polígono resultante da triangulação está apresentado na Figura 20.

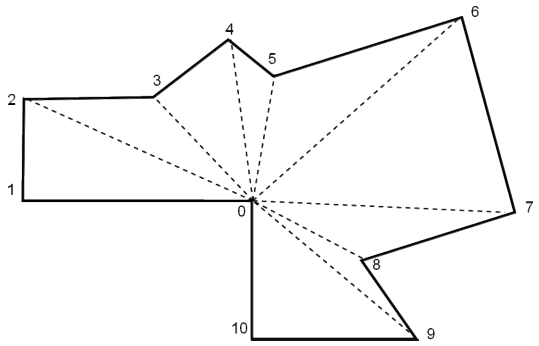

Figura 20 - Polígono triangulado.

4.2.1 – Estrutura de dados

- polígono: polígono simples a decompor;
- triângulo: cada triângulo é definido pelos índices de três pontos, tem de uma a três diagonais associadas e possui uma variável booleana para saber se já foi conquistado;
- diagonal: cada diagonal é definida pelos índices de dois pontos, tem dois triângulos associados;

- vertices_activos: estrutura booleana com mesmo número de elementos que o polígono, que permite calcular o índice real de um vértice. À medida que vamos eliminando vértices no polígono temporário vamos "desactivando-os" (0) neste vector, sendo a posição real igual ao índice onde o somatório dos elementos até a esse ponto seja igual à posição_alterada;

4.2.2 – Descrição das funções

- obtem_posição_inicial(): procura vértice onde as arestas adjacentes façam um ângulo convexo, devolve o índice;
- verifica_diagonal(): testa se pode traçar diagonal, sem intersectar nenhuma aresta e sem que o triângulo formado por *i*, *i+1* e *i+2* contenha algum ponto adicional no interior;
- adiciona(): adiciona primeiro elemento ao segundo.

4.2.3 – Algoritmo

```
algoritmo triangula(polígono, diagonais, triângulos)
{
SE testa_se_é_triângulo(polígono) == verdadeiro
    RETORNA;
poligono_temporário = polígono;
vertices_activos[numero de vértices do polígono];
FAZER
{
    posição_alterada = obtem_posição_inicial( poligono_temporário );
    diagonal_valida = falso;
    diagonal_valida = verifica_diagonal( poligono_temporário, posição_alterada,
posição_alterada +1, posição_alterada +2);
    SE diagonal_valida == verdadeiro
    {
        posição_real1 = obtem_posição_real( posição_alterada, vertices_activos );
        posição_real2 = obtem_posição_real( posição_alterada+1, vertices_activos );
        posição_real3 = obtem_posição_real( posição_alterada+2, vertices_activos );
        triângulo = forma_triângulo(posição_real1, posição_real2, posição_real3);
        adiciona(triângulo, triângulos);
        diagonal = forma_diagonal(posição_real3, posição_real1);
        adiciona(diagonal, diagonais);
        apaga(poligono_temporário[posição_alterada+1]);
        vertices_activos[posição_real2] = falso;
    }
}ENQUANTO numero de vértices do polígono temporário >= 3;
//quando resta um triângulo não é criada uma diagonal
apaga(diagonais[ultima_diagonal]);
}
```

4.3 – Partição e Cobertura

Os triângulos criados são a estrutura elementar de uma partição ou cobertura. A diferença entre uma partição e uma cobertura de um polígono é que os elementos da cobertura podem sobrepor-se, ou seja, podem conter triângulos já utilizados por outros elementos. O princípio básico do algoritmo implementado é o da "conquista" de triângulos. A explicação do algoritmo terá por base a partição de polígonos e sempre que necessário, serão apresentadas as diferenças para a cobertura de polígonos.

O primeiro problema é a escolha do ponto inicial. Qualquer triângulo serve desde que esteja livre. Portanto, é sempre escolhido o primeiro triângulo livre no conjunto, sendo declarado como conquistado. As opções de expansão são definidas pelas diagonais associadas a cada triângulo, permitindo ter noção do que ainda não foi testado. Comecemos por analisar uma diagonal. Conhecendo a diagonal sabemos qual é o triângulo associado e, se o triângulo estiver livre, como dois dos pontos do triângulo já pertencem à partição (diagonal), basta inserir o ponto que falta e testar a sua convexidade (através das *funções D* já implementadas na biblioteca, verificamos se o vértice seguinte do polígono está sempre para o mesmo lado relativamente aos dois anteriores). Se não for, elimina o ponto adicionado e elimina a opção que correspondia à diagonal explorada continuando a análise de eventuais opções ainda existentes. Se for convexa, elimina-se a opção que corresponde à diagonal explorada e adicionam-se as novas opções disponibilizadas pelo triângulo conquistado. Quando acabarem as opções significa que este elemento da partição está concluído. Segue-se novo ciclo de determinação de um triângulo inicial para a obtenção de um novo elemento da partição. Quando não se conseguir encontrar nenhum triângulo livre significa que já foram obtidos todos os elementos da partição.

Quanto à cobertura, é muito semelhante: exige-se igualmente um triângulo livre para se iniciar a cobertura mas explora-se sempre todas as opções independentemente destas envolverem triângulos já utilizados. Neste último caso é ainda necessário adicionar uma restrição relativamente à adição de triângulos que já existam na própria cobertura. Tal como no algoritmo da triangulação, foi decidido utilizar índices, variáveis inteiras, em vez das próprias coordenadas dos pontos, que são números em vírgula flutuante.

Considere-se o polígono triangulado da Figura 20 e o algoritmo de partição. O primeiro triângulo adicionado ao conjunto foi o triângulo formado pelos pontos (0, 1, 2), sendo que este constitui um elemento da partição que vamos tentar expandir. Adicionamos o triângulo adjacente (0, 2, 3), que está livre e testamos a convexidade do polígono resultante. É convexo portanto adiciona-se a opção de expansão, a diagonal (0,3). Adiciona-se o triângulo seguinte (0, 3, 4), que está livre, e testa-se a convexidade do polígono resultante. Não é convexo portanto elimina-se o último triângulo adicionado. O elemento actual da partição não tem mais opções de expansão pelo que está encontrado o primeiro elemento da partição. Como ainda há triângulos livres, forma-se um novo elemento da partição com o primeiro triângulo livre que se encontrar. O resto da análise é semelhante. O resultado da decomposição do polígono da Figura 19 numa partição convexa pode ser observado na Figura 21.

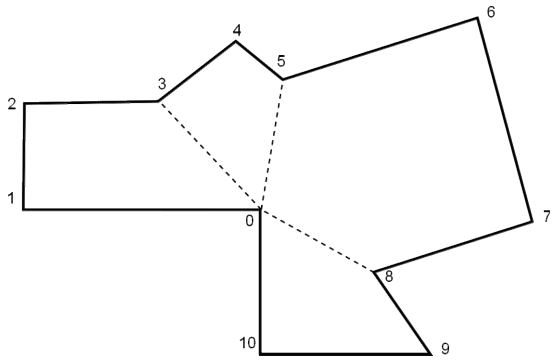

Figura 21 – Decomposição de um polígono simples numa partição convexa.

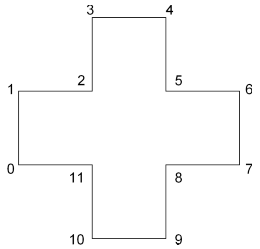

Figura 22 - Polígono simples. Deste polígono é criada uma partição com 3 elementos e uma cobertura com 2 elementos.

4.3.1 – Estruturas de dados

- polígono: polígono simples a decompor;
- triângulo: cada triângulo é definido pelos índices de três pontos, tem de uma a três diagonais associadas e possui uma variável booleana para saber se já foi conquistado;
- diagonal: cada diagonal é definida pelos índices de dois pontos, tem dois triângulos associados.

4.3.2 – Descrição das funções

- obtem_triângulo_adjacente(): determina o triângulo, adjacente à diagonal, que não está na partição corrente. Neste caso não necessita de saber quais os triângulos em determinado elemento da partição porque através do campo intrínseco de cada triângulo relativo à conquista sabemos se está disponível ou não. No caso da cobertura é diferente, porque pode ser usado mais que uma vez, só não no mesmo elemento da cobertura. Por isso é que se criou a estrutura triângulos_neste elemento_cobertura;
- encontra_indice_inserçao(): Encontra a aresta de um elemento da partição que é a diagonal a ser testada. Vai inserir o 3º vértice do triângulo a testar entre os vértices da diagonal.
- adiciona_ao elemento_partição(): Adiciona o vértice do triângulo que não pertence à diagonal, no índice de inserção;
- obtém_coordenadas(): todas os elementos da partição/cobertura obtida estão definidos com os índices dos vértices do polígono. Portanto antes de sair é necessário copiar as coordenadas apontadas por cada índice.

4.3.3 – Algoritmo

```
algoritmo partição(polígono, diagonais, triângulos, cobertura)
{
//SE cobertura == verdade podemos adicionar triângulos já conquistados, sobreposição
permitida
REPETE
{
    //se não há opções disponíveis, cria novo elemento da partição com um triângulo livre
    SE opções == 0
    {
        encontrou_triângulo_livre = falso;
        PARA (i=0; i < numero de triângulos, i++)
            SE (triângulo[i].conquistado == falso) E encontrou_triângulo_livre == falso)
            {
                indice__triângulo_inicial = i;
```

```
                encontrou_triângulo_livre = verdade;
            }
        SE encontrou_triângulo_livre == verdade
        {
            triângulos[indice_triângulo_inicial].conquistado = verdade;
            SE cobertura == verdade)
            {
                //se está aqui significa que este elemento da cobertura está vazio
                adiciona( indice_triângulo_inicial, triângulos_neste elemento_cobertura);
            }
            adiciona( triângulos[indice_triângulo_inicial], elemento_partição_temporário);
            triângulo_a_testar = triângulo[indice_triângulo_inicial];
                    //adiciona diagonais do triângulo às opções
            adiciona( triângulo_a_testar.diagonais, opções);
        }
    }
      //esta condição é necessária para quando não encontrar um triângulo livre
    SE encontrou_triângulo_livre == verdade
    {
        //testa cada opção
        REPETE
        {
            ambos_triângulos_conquistados = falso;
            diagonal_a_testar = diagonal[opção[0]];
            //encontra triângulo associado à diagonal a testar, excluindo triângulos neste
elemento da partição
            SE cobertura == falso
            {
                triângulo_a_testar = obtem_triângulo_adjacente(diagonal_a_testar);
                //um dos triângulos já pertence à partição, testamos o adjacente
                SE triagulo_a_testar.conquistado = verdade
                    ambos_triângulos_conquistados;
            }
            SE cobertura == verdade
            {
                //seleciona um triângulo que ainda não pertence a este elemento da cobertura
                triângulo_a_testar = obtem_triângulo_adjacente(diagonal_a_testar,
triângulos_neste elemento_cobertura);
            }
            //só é permitido explorar um triângulo se não está conquistado ou se estamos a
fazer uma cobertura
            SE ambos_triângulos_conquistados == falso OU cobertura == verdadeiro
            {
                indice_inserção = encontra_indíce_inserção;
                adiciona_ao elemento_partição(triângulo_a_testar, diagonal_a_testar,
indice_inserção, elemento_partição_temporário);
                convexo = falso;
                convexo = testa_convexidade(elemento_partição_temporário);
                SE convexo == falso
                    apaga(elemento_partição_temporário [indice_inserção]);
                SE convexo == verdade
                {
                    triângulos[triângulo_a_testar.indice].conquistado = verdade;
                    SE cobertura = verdade
                        adiciona(triângulo_a_testar.indice, triângulos_neste
elemento_cobertura);
                    //adiciona novas opções
                    adiciona( triângulo_a_testar.diagonais, opções);
                }
                //uma opção foi explorada, pode ser apagada
                apaga(opção[0]);
            }
        }ENQUANTO numero de opções > 0;
    }
    //acabaram as opções, guarda elemento da partição
    SE encontrou_triângulo_livre = verdade
    {
```

```
        adiciona(elemento_partição_temporário, indices_elementos_partição);
        apaga(partição_temporária);
    }
}ENQUANTO encontrou_triângulo_livre = verdade;
coordenadas_elementos_partição = obtem_coordenadas(indices_elementos_partição,
polígono);

RETORNA coordenadas_elementos_partição;
}
```

4.4 – Remoção de Contornos Interiores

A capacidade de lidar com peças com buracos é uma das novas funcionalidades a adicionar à biblioteca geométrica. Uma peça com buracos é representada por um polígono "não simples"[11], que tem obrigatoriamente um contorno exterior e poderá ter um ou mais contornos interiores. No caso de existirem contornos interiores (situação facilmente detectável já que ocorre apenas quando o polígono "não simples" tem mais do que um contorno) é necessário proceder à remoção dos contornos interiores. Este é um passo indispensável uma vez que mais nenhuma outra função na biblioteca geométrica tem a capacidade de lidar com os referidos contornos interiores. A esta função cabe então a tarefa de "transformar" os polígonos "não simples" em vários polígonos simples de forma a que as outras funções possam actuar sobre eles correctamente.

A maneira mais simples de entender a operação de remoção de contornos interiores é através do exemplo da Figura 23. A ideia base é dividir o polígono com contornos interiores por uma linha horizontal que passa pelo centro de um dos contornos interiores. A aplicação desta ideia permite remover um contorno interior, obtendo-se dois polígonos simples. A peça deve ser dividida tantas vezes quanto o numero contornos interiores existentes. No caso em que uma recta horizontal traçada intersecte mais que um contorno interior (segmento (8, 15)) não é necessário voltar a traçar uma nova para estes, dado que o importante é que haja dois pontos de intersecção em cada contorno interior para efectuar a decomposição. Um polígono com contornos interiores é dividido tantas vezes quantas forem as intersecções. Depois de serem traçadas as rectas horizontais, são calculadas as intersecções com os contornos exterior e interiores e, como todos os pontos da recta têm ordenada constante, são ordenadas segundo a abcissa. Este passo irá permitir a criação das diagonais entre intersecções consecutivas. Uma vez que só nos interessam diagonais entre contornos diferentes (são usadas para "saltar" entre contorno, ou seja, P_{ext}- P_{int} e P_{int1} -P_{int2}) são eliminadas todas as que não respeitam esta condição (por exemplo a diagonal (17, 20)). O método de criação dos polígonos simples consiste resumidamente em começar num ponto original do contorno (exterior ou interior) e, quando se encontrar uma diagonal, passar para o ponto adjacente (quando encontramos o ponto 2 continuamos para o ponto 20), até se atingir o ponto inicial, onde o polígono simples é concluído. É importante não esquecer que os pontos do contorno exterior estão ordenados segundo o sentido horário, enquanto os interiores estão no sentido oposto, garantindo-se assim que os diferentes polígonos simples formados só partilham as diagonais. Como se estabeleceu que quando for atingida uma diagonal trocamos de contorno, o ponto inicial não pode pertencer a uma diagonal. Se fosse esse o caso iria ocorrer um conflito, pois poderia passar-se para o contorno errado (já

[11] Neste contexto um polígono não simples é composto por vários contornos.

percorrido). Como os pontos das diagonais são os únicos que são partilhados podem ser utilizados duas vezes (em sentidos inversos), enquanto que os restantes só podem ser utilizados uma vez (condição importante para determinação do ponto inicial). Mais uma vez por razões de eficiência são utilizados índices em vez das próprias coordenadas. Foi criada uma estrutura adequada em que os pontos (incluindo as intersecções de cada contorno) estão ordenados consecutivamente (e por contorno) permitindo saber somente através do índice qual o contorno a que pertence.

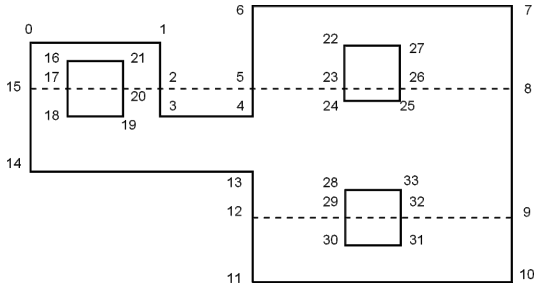

Figura 23 – Remoção de contornos interiores em polígonos com "buracos".

4.4.1 – Estruturas de dados

- peça: é um polígono, contendo obrigatoriamente um contorno exterior e poderá ter contornos interiores. Estão também associados os polígonos simples que vão ser criados nesta função assim como os convexos que vão ser obtidos dos primeiros (a reunião de cada conjunto de polígonos simples ou convexos constitui a peça). Contêm também a estrutura com os índices referida anteriormente e o conjunto de diagonais determinado nesta função.
- cria_recta[polígono]: vector booleano que define se é permitido traçar uma recta para intersectar determinado contorno;
- intersecções[recta]: guarda intersecções por recta;
- polígono_global: contem todos os vértices dos contorno s e respectivas intersecções, de forma a poder utilizar um índice global;
- índices_globais: conjunto de vectores que relacionam o índice de cada contorno (número de coluna) com o índice global (valor guardado). O índice do contorno é o número de linha.
- ordena_intersecções: ordena as intersecções de acordo com a abcissa de cada ponto;
- posições_livres: vector inteiro que contem o numero de vezes que podemos utilizar cada ponto. Podemos utilizar duas vezes os pontos de uma diagonal e só uma os restantes.

4.4.2 – Descrição das funções

- obtém_recta(): cria a recta horizontal que passa no centro do contorno interior à entrada, necessitando do contorno exterior para dimensiona-la correctamente.
- obtém_intersecções_e_insere(): traça recta, determina pontos de intersecção entre esta e os contornos e insere à medida que encontra;
- adiciona(): coloca o primeiro elemento dentro do segundo;
- obtém_diagonais(): determina diagonais entre contornos diferentes, guardando o índice_global das intersecções correspondentes;
- obtém_posições_livres(): ver posições_livres;

- verifica_se_diagonal(): verifica se o ponto actual pertence a uma diagonal, devolve variável booleana;
- actualiza_índice(): devolve índice do ponto adjacente à diagonal que contém o ponto com índice_actual;
- preenche_coord(): a partir dos índices à entrada devolve as coordenadas do elemento correspondente em polígono_global, preenchendo os polígonos simples com as coordenadas respectivas.

4.4.3 – Algoritmo

```
algoritmo remove_contornos_interiores(peça)
{
PARA todos os contornos
    cria_recta[contorno] = verdade;
SE peça.número_contornos > 1 // Se forem mais que 1 significa que há contornos interiores
{
    PARA todos os contornos interiores i
    {
        SE cria_recta[contorno] == verdade
        {
            recta = obtém_recta(contorno exterior, contorno interior);
            PARA todos os contornos j
            {
                intersecções_por_recta = obtém_intersecções_e_insere(recta, contorno [i]);
                SE j != i
                    cria_recta[contorno j] = falso;

                    }
            adiciona(intersecções_por_recta, intersecções);
            apaga(intersecções_por_recta);
             }
    }
    adiciona(todos os contornos, polígono_global);
    índices_globais[número de contornos][número de vértices de cada contorno];
    k=0;
    PARA todos os contornos
    {
        PARA todos os vértices de cada contorno
        {
            adiciona(k, contorno_temporário);
            k++;
        }
        adiciona(contorno_temporário, índices_globais);
        apaga(contorno_temporário);
    }
    PARA todas as rectas
        ordena_intersecções(intersecções[recta]);
    diagonais = obtém_diagonais(intersecções, índices_globais);
    posições_livres = obtém_posições_livres(diagonais, índices_globais);
    REPETE
    {
        actualizou = falso;
        encontrou_índice_inicial = falso;
        índice_inicial = encontra_índice_inicial(índices_globais, posições_livres,
encontrou_índice_inicial);
        índice_actual = índice_actual;
        SE encontrou_índice_inicial == verdade
        {
            REPETE
            {
                terminou_polígono_simples = verdade;
                adiciona(índice_actual, polígono_simples_temp);
                posições_livres[índice_actual]--;
```

```
                encontrou_diagonal = verifica_se_diagonal(índice_actual, diagonais);
                SE encontrou_diagonal == verdade
                {
                    índice_actual = actualiza_índice(índice_actual, diagonais);
                    adiciona(índice_actual, polígono_simples_temp);
                                posições_livres[índice_actual]--;
                }
                índice_actual++;
                SE índice_actual == índice_inicial E número de elementos de
polígono_simples_temp > 1
                    terminou_polígono_simples = verdade;
                    }ENQUANTO(terminou_polígono_simples == falso);
            }
        SE encontrou_índice_inicial == verdade
        {
            adiciona(polígono_simples_temp, polígonos_simples_índices);
            apaga(polígono_simples_temp);
             }
}ENQUANTO(encontrou_índice_inicial == verdade);
peça.polígonos_simples = preenche_coord(polígono_global, polígonos_simples_índices);

RETORNA;
}

//peça não tem contornos interiores
RETORNA;
}
```

4.5 – Cálculo de *NFPs* de Polígonos Convexos

Como já foi referido, a implementação do algoritmo que calcula o *NFP* exterior será baseada no algoritmo proposto por Cuninghame-Green [12] em 1989. Iniciamos o algoritmo com a inversão do polígono orbital (ver diferença de Minkowski, explicada em 2.3.2). Em seguida, juntamos as arestas dos dois polígonos, calculamos o ângulo de cada uma em relação ao eixo positivo das abcissas e procedemos à ordenação das arestas segundo o ângulo calculado. O *NFP* constrói-se inserindo aresta a aresta segundo a ordem determinada, mantendo a sua magnitude e direcção.

4.5.1 – Estrutura de dados

- aresta: cada aresta tem dois pontos associados, inicial e final, um ângulo e uma variável booleana que define se a aresta pertence ao polígono orbital ou ao estacionário.

4.5.2 – Descrição das funções

- obtém_arestas(): copia as arestas dos polígonos e retira o offset em relação à origem, de forma a poder calcular os índices e inserir no *NFP* facilmente;
- adiciona_aresta(): adicionar uma aresta ao *NFP* equivale à adição de um ponto dado que o primeiro ponto da aresta é que foi adicionado imediatamente antes. Os pontos são sempre actualizados com as coordenadas do ponto anterior uma vez têm que iniciar onde a anterior acabou;
- calcula_offset_global(): as verdadeiras coordenadas de um ponto no *NFP* são calculadas adicionando as coordenadas do último ponto de uma aresta ordenada com as coordenadas do ponto inicial da aresta seguinte, sendo que as arestas têm de

pertencer a polígonos diferentes. Subtraindo as coordenadas do ponto do *NFP* que corresponde à intersecção entre estas duas arestas temos o desvio do polígono em relação à posição correcta.

4.5.3 – Algoritmo

```
algoritmo NFP_convexo(polígono estacionário, polígono orbital)
{
    //diferença de Minkowski: roda polígono orbital 180º
    orbital_rodado = roda_polígno(orbital, PI);

arestas = obtem_arestas(estacionário, orbital_rodado);
ordena_arestas(arestas);
offset = coordenada(0,0);
adiciona( coordenada(0,0), NFP );
PARA todas as arestas
{
    adiciona_aresta_NFP(aresta, offset, NFP);
    offset = ultimo vértice adicionado;
}

//o NFP está deslocado em relação à posição correcta
calcula_offset_global(arestas, NFP);
actualiza(NFP, global_offset);

RETORNA NFP;
}
```

4.6 – União de Polígonos

A biblioteca original já tem a capacidade de unir dois polígonos simples, desde que os polígonos estejam em contacto e não existam sobreposições. É necessário fornecer como parâmetros de entrada os polígonos a unir e a posição em que se deve colocar o segundo polígono, sendo que a união é realizada através dos pontos de contacto. Qualquer espaço entre os polígonos originais é perdido.
Este algoritmo foi desenvolvido com o objectivo de unir um conjunto de polígonos convexos sobrepostos. Constitui o último passo da formação de *NFPs* de polígonos não convexos, a executar depois da criação dos *NFPs* convexos descrita no algoritmo anterior. Uma das vantagens deste algoritmo de união é que analisa áreas interiores livres.

Inicialmente são calculadas todas as intersecções entre os polígonos, sendo eliminadas as que se encontram no interior de algum polígono (só necessitamos de intersecções que pertençam ao contorno exterior ou interior do polígono). As restantes intersecções são adicionadas aos polígonos envolvidos como vértices. Começando no vértice com menor ***x*** e menor ***y*** (vértice que garantidamente pertence ao contorno que queremos calcular) percorremos o polígono *A* até encontrar uma intersecção. Como uma intersecção está associada a dois polígonos e foi inclusivamente inserida como um vértice, trocamos de polígono, actualizando o índice do vértice actual e passamos agora a percorrer *B*. Aplicamos o mesmo método até atingirmos o vértice inicial, obtendo o contorno exterior. Como eliminamos intersecções dentro de polígonos, neste momento só utilizamos intersecções contidas no contorno exterior. Podemos concluir sobre a existência de um contorno interior se ainda existirem intersecções que não tenham sido utilizadas. Isto é justificado pelo facto de todos os polígonos à entrada serem convexos, portanto sem contornos interiores, sendo

impossível que da sobreposição resulte um contorno interior sem intersecções. Tal como para o contorno exterior continuamos a adicionar vértices, trocando de polígono quando atingirmos uma intersecção, e terminamos com o regresso ao vértice inicial. Só se procura um novo contorno interior se ainda existirem intersecções por utilizar. Agora que a ideia base do algoritmo já foi explicada, passamos a uma análise mais detalhada.

Para permitir a correcta construção dos polígonos estabeleceram-se três restrições básicas que têm de ser respeitadas quando lidamos com intersecções. A primeira refere que não podemos trocar de polígono quando o ponto seguinte (à intersecção) nesse polígono tiver sido usado (considera-se um ponto como *usado* se este fizer parte do contorno exterior/interior ou se estiver dentro de algum polígono). Considera-se que contornos exteriores e interiores são independentes, sem pontos ou arestas comuns e que cada intersecção envolve dois polígonos. A segunda refere que no caso de segmentos colineares, com o mesmo sentido, entre os polígonos originais, em que a colinearidade se inicia e acaba numa intersecção, só se troca de polígono no fim, ou seja, na última intersecção. O que se quer evitar é o regresso ao polígono analisado anteriormente (caso de troca na primeira intersecção para o novo polígono, e regresso na segunda, após o segmento colinear). A terceira só é aplicada aquando da selecção do ponto inicial para o cálculo dos polígonos interiores e consiste em não permitir que o ponto inicial seja uma intersecção cujo ponto seguinte se for uma intersecção envolva os mesmos polígonos. Tem como objectivo impedir a selecção de segmentos no interior de polígonos. A primeira e segunda regras impedem a troca, continuando a análise do polígono actual até se encontrar outra intersecção, onde se verifica de novo se esta troca se pode efectuar.

Na Figura 24 podemos verificar exemplos de aplicação das regras. Quando estamos no ponto 9, que é uma intersecção, o segmento que o une ao ponto seguinte, também uma intersecção, é colinear sendo aplicada a segunda regra e só trocamos de polígono no ponto 10. Quanto à selecção do ponto inicial do contorno interior veja-se o exemplo do ponto 16: tanto o ponto 16 como o 21 estão localizados fora dos polígonos mas o segmento que os une não o está, tendo de ser aplicada a terceira regra para evitar este tipo de situações.

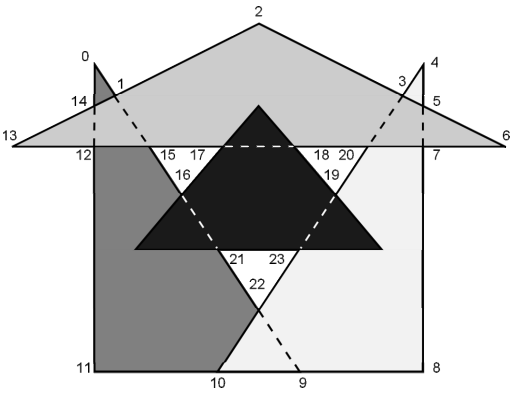

Figura 24 - Conjunto de polígonos sobrepostos.

Durante a projecção deste algoritmo definiu-se uma intersecção como um ponto comum entre contornos de dois polígonos convexos. Fomos confrontados mais tarde (ver exemplos no capítulo seguinte) com a existência de intersecções que envolviam mais de três polígonos, situação para a qual este algoritmo não está preparado. Como envolve alterações ao nível mais básico (definição de intersecção) são necessárias também modificações nas funções de mais alto nível, facto que é impraticável face ao tempo disponível para a conclusão desta

dissertação. Sendo assim, sugerimos uma possível solução para este problema. Poderia manter-se a estrutura do algoritmo actual, mas o seu desenvolvimento iria ser demasiado complexo. A abordagem correcta passa então por implementar uma estrutura iterativa: unir os polígonos um a um, calcular as intersecções desses polígonos e actualizar contorno exterior/interior (o algoritmo implementado calcula todas as intersecções no início e lida com todos os polígonos originais). Basicamente, sempre que adicionamos um polígono convexo ao conjunto, temos de calcular as intersecções entre esse polígono e o *NFP* existente, inserir estas intersecções como pontos do *NFP* (no contorno exterior ou interior) e calcular o *NFP* resultante (actualizando-o). Assim lidamos com dois polígonos simples de cada vez (o que permite manter a definição actual de intersecção) em que um polígono é convexo e o outro (o que define o *NFP*) pode ser um polígono simples com "buracos". A existência de contornos interiores não é problemática dado que, através das estruturas de dados apropriadas, guardamos sempre os vértices de cada contorno.

4.6.1 – Estruturas de dados

- intersecções: a estrutura que representa uma intersecção contém os índices dos polígonos envolvidos, o índice do vértice que representa a intersecção em cada polígono, a coordenada do ponto e o quadrado da distância do ponto ao fim da aresta de cada polígono;
- pontos_usados: variável que define se um ponto já foi usado ou não.
- polígonos_finais: contorno exterior e, se existirem, contornos interiores.

4.6.2 – Descrição das funções

- calcula_distâncias(): calcula distância de cada ponto de intersecção ao fim da aresta;
- adiciona(): adiciona primeiro elemento ao segundo;
- apaga_intersecções_interiores(): remove intersecções que se encontram no interior de qualquer polígono;
- actualiza_índice_referência(): no início da análise de cada aresta actualiza o índice referente ao vértice inicial da aresta. É obrigatório com a adição das intersecções;
- apaga(): apaga elemento, se tiver mais que um significa que apaga o primeiro elemento referido da estrutura que representa o segundo;
- menor_distância(): a distância de uma intersecção já está incluída na estrutura que a define, portanto compara vários até encontrar a menor;
- adiciona_intersecções_polígono(): como as intersecções já estão ordenadas, basta inserir uma de cada vez na aresta respectiva;
- obtém_pontos_usados(): inicializa pontos_usados. Todos os vértices dentro dos polígonos convexos à entrada são considerados usados para não haver hipótese de o serem realmente.
- obtém_ponto_inicial_exterior(): procura o ponto com menor y e menor x;
- respeita_regras(): verifica se são respeitadas as regras acima descritas, se é possível trocar de polígono;
- troca_polígono(): uma intersecção define dois pontos, um em cada polígono intersectado, logo quando esta função é chamada é actualizado (trocado) o vértice e o polígono;

- obtém_ponto_inicial_interior(): o ponto inicial para o contorno interior ou é um ponto normal ou é uma intersecção. Como no pior caso podem só existir intersecções (um contorno interior só pode ser criado através da intersecção de polígonos convexos, não existem polígonos à entrada com "buracos"), procura a primeira intersecção livre (as que já foram usadas são apagadas) que respeite a primeira e terceira regras. Aqui é essencial a escolha do polígono e vértices correctos relativos à intersecção porque o ponto seguinte ao inicial não pode estar no interior de um polígono nem ter sido usado.

4.6.3 – Algoritmo

```
algoritmo une_polígonos_sobrepostos (polígonos)
{

intersecções = calcula_intersecções(polígonos);
calcula_distâncias(intersecções);
apaga_intersecções_interiores(intersecções);
PARA todos os polígonos
{
    índice_referência = 0;
    PARA todas as arestas
    {
        intersecções_por_aresta = procura_intersecções(aresta);
        SE numero de intersecções_por_aresta > 0
        {
            actualiza_índice_referência(índice_referência);
            REPETE
            {
                intersecção_mais_perto = menor_distância(intersecções_por_aresta);
                actualiza_índices_inserção(intersecção_mais_perto);
                adiciona(intersecção_mais_perto, intersecções_por_polígono);
                apaga(intersecção_mais_perto, intersecções_por_aresta);
                    }ENQUANTO(numero de intersecções_por_aresta >0);
             }
        SENÃO
            índice_referência++;
      }
    adiciona_intersecções_polígono(intersecções_por_polígono, polígono);
    apaga(intersecções_por_polígono);
}
pontos_usados[numero de polígonos][numero vértices/polígono ]; //inclui intersecções
pontos_usados = obtém_pontos_usados(polígonos);
obtém_ponto_inicial_exterior(polígonos, vértice_inicial, polígono_inicial);
vértice_actual = vértice_inicial;
polígono_actual = polígono_inicial;
terminou_NFP = falso;
terminou_ifp = falso;
começou_ifp = falso;
REPETE
{
    SE terminou_NFP = falso      //adiciona ao polígono externo
        adiciona(polígonos[ polígono_actual ][ veértice_actual ], NFP_temporário);
    SENÃO    //adiciona ao contorno interior
        adiciona(polígonos[ polígono_actual ][ vértice_actual ], ifp_temporário);
    pontos_usados[ polígono_actual ][ vértice_actual ] = verdade;
    actualizou = falso;
    PARA todas as intersecções
    {
        SE vértice_actual = intersecção E actualizou = falso;
            SE respeita_regras(vértice_actual, polígono_actual, polígonos);
            {
                troca_polígono(vértice_actual, polígono_actual, intersecção);
```

```
                apaga(intersecção, intersecções);
                actualizou = verdadeiro;
            }
    }
    vértice_actual++;
    SE vértice_actual = vértice_inicial E polígono_actual = polígono_inicial E
terminou_NFP = verdade
    {
        terminou_ifp = verdade;
        adiciona(ifp_temporário, polígonos_finais);
        apaga(ifp_temporário);
      }
    SE vértice_actual = vértice_inicial E polígono_actual = polígono_inicial E
terminou_NFP = falso
    {
        terminou_NFP = verdade;
        adiciona(NFP_temporário, polígonos_finais);
        apaga(NFP_temporário);
      }
    SE (vértice_actual = vértice_inicial E polígono_actual = polígono_inicial E
terminou_NFP = verdade E número intersecções > 0 E começou_ifp = falso) OU
(terminou_ifp = verdadeiro E número intersecções > 0)
    {
        //nesta altura todos os pontos livres pertencem a polígonos interiores (pode ser mais
      que um polígono)
        obtém_ponto_inicial_interior(polígonos, intersecções, vértice_inicial,
polígono_inicial);
        começou_ifp = verdade;
        terminou_ifp = falso;
      }
}ENQUANTO terminou_NFP = falso OU numero de intersecções > 0 OU (começou_ifp =
verdade E terminou_ifp = falso);
RETORNA polígonos_finais;
}
```

4.7 - Cálculo do *NFP* de polígonos simples com “buracos”

Inicialmente, é chamada a função que decompõe este tipo de polígonos em polígonos simples e posteriormente, a triangulação e a partição encarregam-se de gerar os polígonos convexos de todos os polígonos. A determinação da maior partição convexa é independente de uma eventual divisão feita inicialmente para o tratamento de contornos interiores, na medida em que, na triangulação, as diagonais e triângulos geradas são guardadas na mesma estrutura. A partir dos polígonos convexos é calculado o *NFP* através do método de ordenação das arestas. De forma a obter o *NFP* global, incluindo contornos interiores, é chamada a função de união de polígonos convexos.

4.7.1 - Algoritmo

```
algoritmo NFP_alternativo
{
remove_contornos_interiores(poligonos)
PARA cada poligono
    triangula(polígono, diagonais, triângulos);
    partição(polígono, diagonais, triângulos, falso);
poligonos_convexos = NFP_convexo(polígono estacionário, polígono orbital)
```

```
une_polígonos_sobrepostos (polígonos_convexos)
}
```

Depois desta apresentação detalhada de todos os algoritmos implementados, resta-nos testar para provar que foram implementados correctamente e comparar com o que existia anteriormente.

Capítulo 5

Validação dos Algoritmos Geométricos e Testes Computacionais

5.1 – Introdução

Os algoritmos implementados durante a realização deste trabalho (de dissertação) acabaram de ser detalhadamente apresentados no capitulo anterior. Este capítulo será dedicado ao teste e validação desses algoritmos de forma a provar que foram correctamente implementados. Serão ainda demonstradas as novas funcionalidades adicionadas à biblioteca geométrica relativas à capacidade de lidar com polígonos com "buracos", recorrendo-se a exemplos de elevada complexidade. Será também apresentada uma comparação entre o algoritmo de determinação de *NFPs* anteriormente implementado, baseado na estratégia de deslizamento orbital, e o algoritmo desenvolvido nesta dissertação, baseado na estratégia de decomposição em polígonos convexos. Finalmente serão tecidos alguns comentários finais.

5.2 - Validação dos algoritmos

Nesta secção vão ser apresentados quatro conjuntos de testes computacionais. Em cada conjunto são testadas as novas funcionalidades implementadas: remoção de contornos interiores (quando necessário), triangulação, partição, e finalmente o cálculo de *NFPs* convexos.

Foram testados polígonos mais complexos, tipicamente utilizados em investigação nesta área [15], e verificou-se, através da comparação directa do *NFP* gerado pela função implementada anteriormente (método de deslizamento orbital) e do gerado através da sobreposição dos *NFPs* convexos (abordado nesta tese), que estes eram coincidentes (caso em que são abordados polígonos sem contornos interiores). Este facto implica um funcionamento correcto das funções de tratamento de contornos interiores, triangulação, partição, e geração de *NFPs* convexos, uma vez que estão todas envolvidas na operação de cálculo de *NFPs* entre polígonos simples com "buracos".

Irá ser representado o resultado da sobreposição dos *NFPs* convexos obtidos de forma a ilustrar o *NFP* global que é possível obter e comparando com o que é obtido através do método de deslizamento.

Pretende-se assim demonstrar o correcto funcionamento de cada algoritmo e também de todo o conjunto. O novo método de cálculo do *NFP* de polígonos simples envolve a utilização dos *NFPs* convexos, que por sua vez utilizam os polígonos convexos resultantes da partição. Como já foi referido o algoritmo triangulação é essencial ao algoritmo de partição.

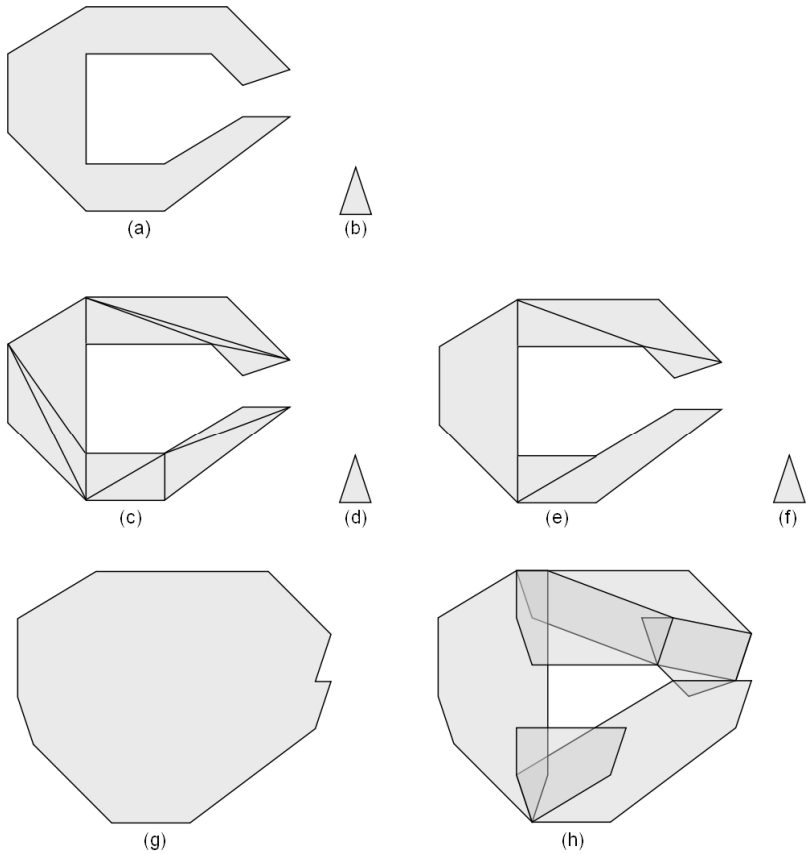

Figura 25 - Teste 1.

No primeiro teste, representado na Figura 25, são manipulados os polígonos *a* e *b*, sendo que o primeiro é o estacionário e o segundo é o orbital. Em *h* está representada a sobreposição dos *NFPs*, sendo este calculados através da partição representada em *e* e *f*. Podemos observar que a partição obtida em *e* corresponde ao agrupamento de triângulos representados em *c*. O principal aspecto observado é a diferença entre o *NFP* gerado pelo método de deslizamento (em *g*) e o resultante da sobreposição dos *NFPs* convexos (em *h*): uma vez que a entrada da concavidade do polígono em *a* é menor que o polígono orbital em *b* não é possível analisar a concavidade com o método de deslizamento.

Consideremos o segundo teste, representado na Figura 26. Este é um problema típico de "encaixe de dentes de serra". Como no teste anterior, o polígono *a* é o estacionário e o *b* é o orbital. A triangulação destes polígonos resulta em *c* e *d* respectivamente. Conquistando triângulos permite-nos obter a partição de cada polígono, representada em *e* e *f*. Neste caso o *NFP* gerado pelos dois métodos é exactamente igual. Neste caso o número de *NFPs* convexos já é razoavelmente elevado.

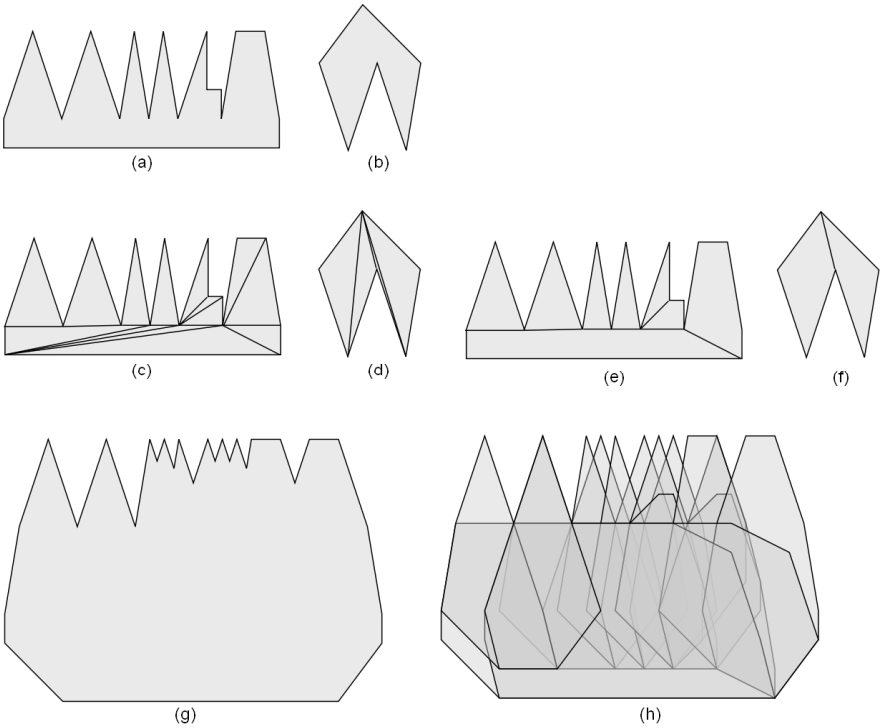

Figura 26 - Teste 2.

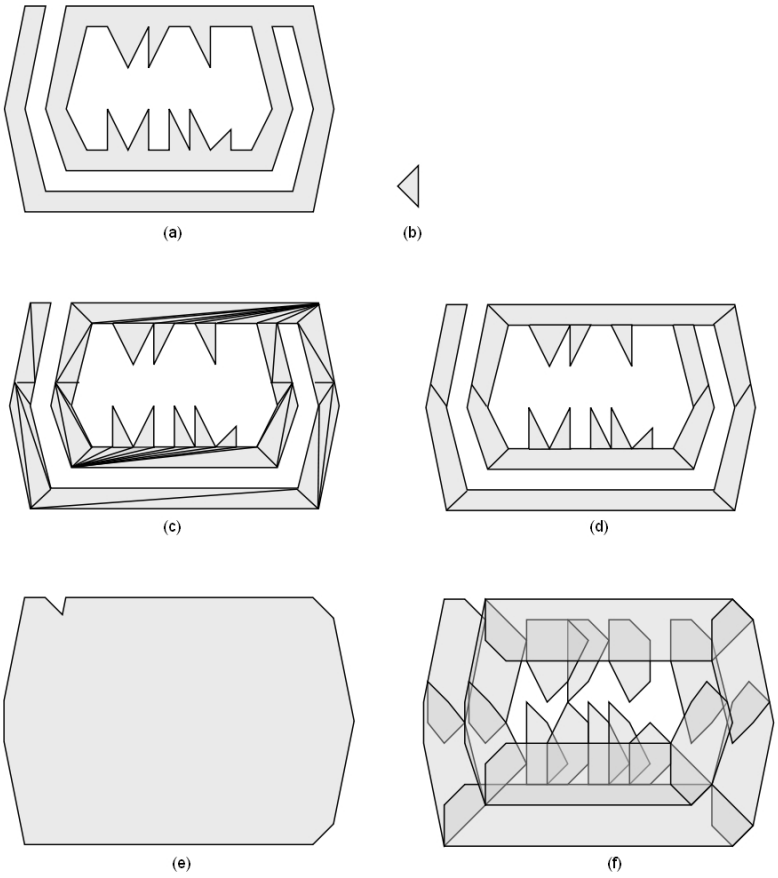

Figura 27 - Teste 3.

O terceiro teste já envolve um polígono com um "buraco". Podem observar-se as diagonais horizontais (em *c*) que foram determinadas no algoritmo de remoção de contornos interiores. Essas diagonais separam dois polígonos simples que foram triangulados separadamente. O resultado desta operação pode ser visualizado em *c*. Em *d* pode observar-se que a partição já lida com o polígono original, agrupando os triângulos da ilustração anterior. Neste caso é evidente a incapacidade do algoritmo de deslizamento (*e*) em lidar com o contorno interior, ao contrário do método de sobreposição de *NFPs* convexos (*f*).

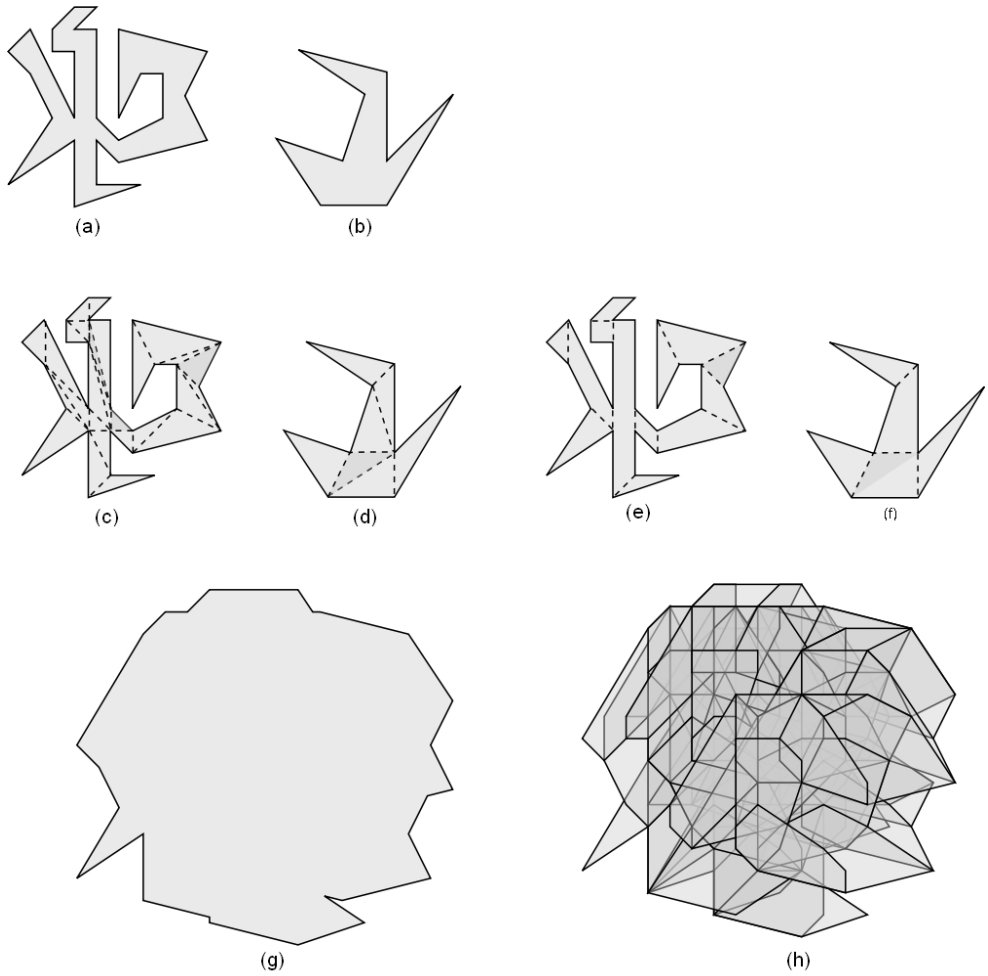

Figura 28 - Teste 4.

O quarto teste é muito realizado na literatura, devido ao elevado número de vértices dos polígonos envolvidos e ao elevado número de subpolígonos convexos a que dão origem (*e* e *f*). Em *c* e *d* está representada a triangulação dos polígonos. Mais uma vez o *NFP* calculado é idêntico. Repare-se no número elevado de *NFPs* convexos gerados em *h*, cerca de 65.

Os testes realizados demonstraram o correcto funcionamento destes algoritmos, ainda que estes testes não tenham sido extensos por falta de tempo.

5.3 – Comentários Finais

Uma vez que o algoritmo de união de polígonos convexos só funciona correctamente no caso das intersecções envolverem apenas dois polígonos, decidiu-se não incluir esta operação nos testes, uma vez os polígonos testados apresentam elevado nível de complexidade. Esta complexidade está associada ao elevado número de polígonos convexos gerados na decomposição. Sendo *m* e *n* o número de polígonos convexos obtidos através da decomposição de *A* e *B* respectivamente, vão ser criados *m x n NFPs* convexos, existindo sempre intersecções envolvendo mais que três polígonos. Este facto torna inútil a utilização do algoritmo de união. Contudo este algoritmo é capaz de lidar com polígonos sobrepostos que gerem intersecções simples.

Não é óbvia a vantagem de um método de cálculo do *NFP* face a outro. Se por um lado o método de deslizamento orbital se encontra algo limitado aquando do tratamento de concavidades, o algoritmo baseado na diferença de Minkowski revela uma complexidade por vezes elevada aquando da sobreposição dos polígonos para geração do *NFP*, sendo esta proporcional ao número de polígonos sobrepostos. Tudo depende portanto do tipo de polígonos a tratar.

Capítulo 6

Conclusões

6.1 – Comentários finais

Atingiram-se os objectivos propostos para esta dissertação.

Na pesquisa bibliográfica destaca-se a pesquisa sobre algoritmos de decomposição e sobre os algoritmos de determinação de *NFPs*. Em face deste estudo foram seleccionados os algoritmos de decomposição de polígonos a implementar.

A interface *web* implementa as funcionalidades previstas, permitindo a demonstração das novas funcionalidades desenvolvidas.

Das funcionalidades adicionadas à biblioteca geométrica destacam-se a decomposição de polígonos por triangulação e em polígonos convexos por partição e cobertura. Estes algoritmos de decomposição permitiram desenvolver o principal resultado deste trabalho que consiste numa nova estratégia de obtenção de *NFPs*: a partir da prévia decomposição em polígonos convexos. Foi ainda implementado um algoritmo que permite a união de polígonos sobrepostos, ainda que só em casos específicos. No entanto, para os algoritmos de *nesting* não é muito importante obter um *NFP* único dado que até é uma vantagem obter polígonos mais simples (convexos), transformando um problema complicado em muitos problemas simples. Adicionalmente, a biblioteca adquiriu a capacidade de lidar com polígonos com "buracos".

6.2 – Desenvolvimentos futuros

Como desenvolvimentos futuros propõe-se o desenvolvimento de um novo algoritmo de união, que aproveite o algoritmo de união de dois polígonos já desenvolvido, de forma a que seja possível obter o resultado final do cálculo do *NFP* através da decomposição convexa. Foi proposta uma solução possível que consiste essencialmente pela sobreposição iterativa de cada polígono convexo ao polígono corrente, actualizando-o.

Sugere-se também a implementação do método de partição segundo a projecção das arestas e o método de J. Bennell [15], uma forma alternativa de calcular o *NFP* sem ser

necessário decompor os polígonos, de forma a poder comparar com os algoritmos descritos nesta dissertação.

Referências

[1] Dowsland, K. A., Dowsland, William B., "Solution approaches to irregular nesting problems", *European Journal of Operational Research*, vol. 84, no.3, 1995.

[2] J. Oliveira, "Problemas de Posicionamento de Figuras Irregulares: Uma perspectiva de optimização", Tese de Doutoramento, FEUP, 1995.

[3] A. Gomes, "Novas Contribuições para o Problema de Posicionamento de Figuras Irregulares", Tese de Mestrado, FEUP, 1995.

[4] Mahadevan, A., "Optimisation in computer aided pattern packing", Ph.D. thesis, North Carolina State University, 1984.

[5] CGAL: www.cgal.org.

[6] Hertel, Mehlhorn, "Fast Triangulation of the Plane with respect to simple polygons", *Inform. Control*, 1985.

[7] J. O'Rourke, "Computational Geometry in C", Cambridge University Press, 1998.

[8] B. Chazelle and D. P. Dobkin, "Optimal convex decompositions", *Computational Geometry,* G. T. Toussaint, Amsterdam, Netherlands, 1985.

[9] J. Mark Keil, "Decomposing a Polygon into Simpler Components", *SIAM Journal on Computing*, vol. 14, no. 4, 1985.

[10] D. Cohen-Or, S. Lev-Yehudi, A. Karol, A. Tal , "Inner-cover of Non-convex Shapes", *The 4th Israel-Korea Bi-National Conference on Geometric Modeling*, Tel-Aviv, 2002.

[11] Konopasek, M., "Mathematical Treatments of Some Apparel Marking and Cutting Problems", U.S. Department of Commerce Report, 1981.

[12] Cunninghame-Green, R., 1989. Geometry, "shoemaking and the milk tray problem", *New Scientist* 12th August 1989, no. 1677, pp. 50–53.

[13] J. A. Bennell, K. A. Dowsland, and W. B. Dowsland, "The irregular cutting-stock problem - a new procedure for deriving the no-fit polygon", *Computers & Operations Research,* vol. 28, no. 3, 2001.

[14] Pankaj K. Agarwal, Eyal Flato, and Dan Halperin, "Polygon decomposition for efficient construction of Minkowski sums," *Computational Geometry* 21, 2002.

[15] Bennell, J. A. and Song, X. 2008. A comprehensive and robust procedure for obtaining the nofit polygon using Minkowski sums. *Comput. Oper. Res.* 35, 1. 2008.

[16] Solving Combinatorial Problems: An XML-based Software Development Infrastructure, Rui Barbosa Martins, Maria Antónia Carravilla, Cristina Ribeiro (2006).

[17] ESICUP: EURO Special Interest Group on Cutting and Packing (2005). Disponível em http://www.fe.up.pt/esicup.

[18] GLOBALNest Project Team: NestingXML Schema (2005) Disponível em: http://globalnest.fe.up.pt/nesting/nesting.xsd.

[19] NGL: www.fe.up.pt/ngl.

[20] Milenkovic V, Daniels K, Li Z. "Placement and compaction of non convex polygons for clothing manufacture", *Fourth Canadian Conference on Computational Geometry*, Newfoundland, 1992.

[21] Stoyan YG, Ponomarenko LD., "Minkowski sum and hodograph of the dense placement vector function", *Reports of the SSR Academy of Science*, 1977.