Universidade Federal do Paraná

Departamento de Informática

André Luiz Pires Guedes

# Introdução à Geometria Computacional

Relatório Técnico
RT_DINF 002/2001

Curitiba, PR
2001

# Introdução à Geometria Computacional

André Luiz Pires Guedes

Depto. de Informática - UFPR

II Encontro Regional da SBMAC
Depatamento de Matemática - UFPR
Curitiba - PR
1994

# Prefácio

A Geometria Computacional é uma disciplina com reconhecimento recente, embora seja uma das mais antigas áreas de computação, sendo anterior aos computadores.

A Computação tem suas raízes na Matemática, e a união destas duas ciências resulta na *Matemática Aplicada*. Aqui temos a aplicação de conceitos das duas para a solução de problemas.

Dentro da Matemática Aplicada, a *Geometria Computacional* se destaca por tratar de um dos mais antigos ramos da Matemática, a *Geometria*, com o que há de mais puro em Ciência da Computação, a *Teoria de Algoritmos*.

Espero, com este curso, preparado para o *II Encontro Regional da SBMAC*, divulgar e incentivar a pesquisa desta disciplina, tão pouco povoada aqui no Brasil.

Gostaria de agradecer a colaboração de Maria Helena Olinisky, e dos professores Tânia Martins Preto e Alexandre Direne, na correção de possíveis erros durante a confecção deste texto, e ao Departamento de Informática da UFPR, que me deu o suporte necessário.

# Sumário

**1 Introdução** **1**

**2 Problemas Clássicos** **2**
2.1 Ponto em Polígono . . . . . 2
2.2 Ponto mais Próximo . . . . . 4
2.3 Fecho Convexo . . . . . 5
2.4 Triangulação . . . . . 8

**3 Técnicas Básicas** **13**
3.1 Dividir-para-conquistar . . . . . 13
3.2 Transformações Geométricas . . . . . 13
3.3 Dualidade . . . . . 14
3.4 Varredura do Espaço . . . . . 14
3.5 Estruturas Hierárquicas . . . . . 16

**4 Problemas Maiores** **19**

# Capítulo 1

# Introdução

O nome *Geometria Computacional* é bastante recente, e se refere ao estudo de algoritmos para a solução de problemas geométricos. Esta disciplina tem raízes bem antigas. Começou com a Geometria Euclidiana Clássica, que com seus axiomas, determinava construções geométricas (algoritmos) baseadas em operações simples.

Este tipo de estudo ficou parado por algum tempo, enquanto a "moda" era a *prova por contradição*, que não gerava nenhum método ou algoritmo de construção. Quando voltaram a aparecer *provas construtivas*, principalmente por conta dos computadores, os algoritmos reapareceram nos estudos geométricos. Entretanto, só em 1985 foi publicado o primeiro livro sobre o assunto, [PS85], escrito por Preparata e Shamos.

Alguns autores atribuem aos estudos geométricos, o nascimento dos estudos de algoritmos e suas complexidades.

Nos dias de hoje, os textos já são mais facilmente encontrados, embora o mais completo ainda seja o livro de Preparata e Shamos. Apareceram outros como o livro de Edelsbrunner [Ede87], e textos como [GS89], e temos até referências em português, como [FC91].

Para abordagens mais profundas, os artigos ainda são a melhor referência. Os artigos do simpósio anual da ACM em Geometria Computacional são bons representantes do que se faz atualmente nesta área.

Atualmente existem muitos trabalhos sobre *Animação de Algoritmos*, que consiste em apresentar didaticamente diversos algoritmos através de imagens e gráficos, o que certamente requer operações geométricas.

Este texto está organizado da seguinte maneira: no segundo capítulo serão apresentados alguns problemas clássicos e alguns esboços de solução. No terceiro são discutidas algumas das técnicas mais usadas na resolução de problemas geométricos, e no quarto, que finaliza o trabalho, é feita uma rápida apresentação de dois problemas mais complexos, que exigem técnicas bem mais apuradas.

# Capítulo 2

# Problemas Clássicos

Neste capítulo abordaremos os problemas clássicos da Geometria Computacional, os quais deram início ao estudo de formas eficientes para resolver problemas geométricos.

## 2.1 Ponto em Polígono

Um problema bastante conhecido é o de saber se um determinado ponto $p$ está no interior de um certo polígono plano simples $P$.

Usando o fato de que um polígono simples é uma *Curva de Jordan*, e por isso divide o plano em duas regiões conexas e abertas, podemos construir um algoritmo que utilize uma semi-reta partindo de $p$. Esta semi-reta estará alternadamente dentro e fora do polígono em questão, mudando de estado nos pontos de intersecção com a borda (ver figura 2.1). Se o número de intersecções for impar, o ponto está dentro, se for par, está fora.

Para calcularmos o número de intersecções de uma semi-reta $r$ (digamos, horizontal para a direita de $p$) com o polígono $P$, precisaremos calcular sua intersecção com cada uma das arestas de $P$. Percebam que, se a semi-reta $r$ passar por um dos vértices de $P$, teremos a contagem alterada (ver figura 2.2).

Aqui temos duas opções para resolver o problema. Uma é escolher qualquer outra semi-reta, por exemplo, girando a semi-reta $r$ poucos graus. Quando nenhum dos vértices pertencer a semi-reta, efetuar o cálculo das intersecções. Esta opção tem a desvantagem de ser preciso verificar se os vértices pertencem a semi-reta.

A outra opção é tratarmos de forma diferenciada as intersecções nos vértices. Nos casos em que o vértice é máximo (ou mínimo) local (ver figura 2.2 (a)), a semi-reta não entra nem sai do polígono, portanto, para que a contagem fique certa, o número de intersecções deve ser par. Nos demais casos a semi-reta realmente entra ou sai do polígono, e o número de intersecções deve ser ímpar (ver figura 2.2 (b)).

Se considerarmos cada aresta como sendo um segmento de reta aberto no extremo inferior e fechado no extremo superior, o número de intersecções nos vértices máximos locais será 2 (dois), que é par, nos mínimos locais será 0 (zero), que podemos considerar como par para os nossos propósitos, e 1 (um) nos demais casos.

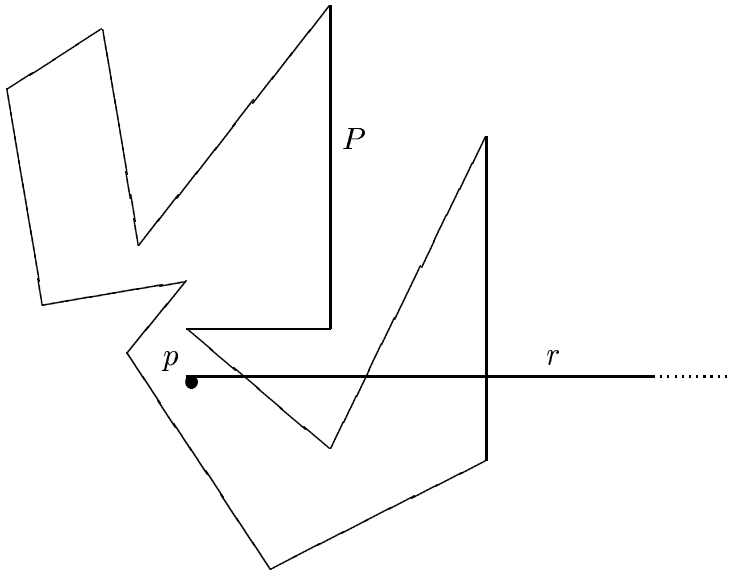
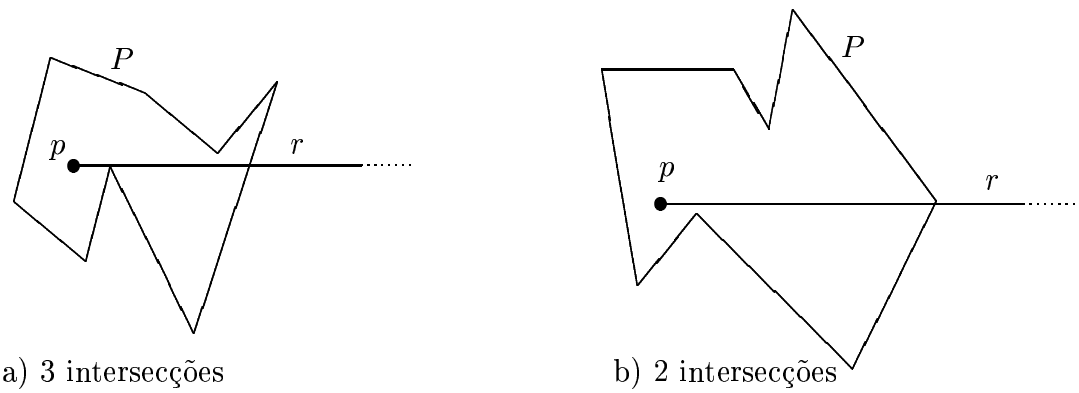



Figura 2.1: Semireta

Observem também que temos que descartar as arestas horizontais. Tente descobrir o motivo!

Calcular a intersecção de uma semi-reta com um segmento de reta é uma operação simples, que pode ser feita em tempo constante. Sendo assim, supondo que o polígono $P$ tem $n$ arestas, temos que calcular $n$ intersecções, portanto, nosso pequeno algoritmo tem complexidade $\mathcal{O}(n)$ no pior caso.




a) 3 intersecções b) 2 intersecções

Figura 2.2: Casos de intersecção nos vertices

## 2.2 Ponto mais Próximo

Este problema aparece, por exemplo, quando temos uma carta com um certo endereço destino, e precisamos enviá-la para algum posto dos correios para que seja entregue. Evidentemente, é conveniente que os carteiros não precisem andar muito para entregar as cartas, portanto, a nossa carta deve ser enviada ao posto mais próximo do endereço destino.

O problema então é o seguinte: dados $n$ pontos no plano, $x_i$, com $1 \leq i \leq n$, e um ponto destino $p$, dizer qual o ponto $x_i$ mais próximo de $p$.

Uma forma direta e ingênua de atacar o problema é usar a *força bruta*. Calculamos todas as distâncias $d_i = d(p, x_i)$, do ponto $p$ ao ponto $x_i$, e procuramos a menor. Esta forma funciona, mas vamos complicar um pouco. E se quisermos resolver este problema para muitas cartas? Será que não podemos fazer melhor?

Como os $n$ pontos $x_i$ são fixos, podemos organizá-los previamente, para que não seja necessário calcularmos todas as distâncias a cada carta. O ideal seria se conseguissemos guardar em cada ponto do plano a informação de qual o ponto mais próximo. Como o plano é infinito e contínuo, isto é impossível!

E se limitarmos o plano? Sim, podemos limitar o plano, já que na prática teremos limites. Limites de países, por exemplo. Mesmo assim o número de pontos continua infinito. Só nos resta dividir o plano em um número finito de regiões, para as quais possamos achar o ponto mais próximo de forma rápida.

Podemos observar que em torno de cada ponto $x_i$ existe uma região $V_i$ onde qualquer ponto $p \in V_i$ tem como ponto mais próximo o ponto $x_i$. A divisão do plano nestas regiões tem o nome de *Diagrama de Voronoi* que veremos na seção 3.3.

Calcular estas regiões e saber em qual delas está o ponto $p$ não são tarefas fáceis, mas é uma opção!

Que tal uma idéia mais simples? Vamos dividir a região do plano em pequenos retângulos iguais. Saber em qual retângulo está o ponto $p$ é bastante simples (ver figura 2.3). Você consegue descobrir como?

E para saber qual o ponto mais próximo de $p$? Se, para cada retângulo, guardarmos uma lista de todos os possíveis pontos, é só usar o algoritmo inicial (força bruta) com os pontos da lista do retângulo onde está $p$.

Falta ainda saber como escolher os pontos para as listas. Todos os pontos que estão no retângulo $R$ devem pertencer à lista $L_R$ do retângulo $R$. Como um ponto $x$ pode estar em um dos vértices, e o ponto $p$ no outro, qualquer ponto, mesmo fora do retângulo, que esteja a uma distância menor ou igual à distância entre $x$ e $p$ deve estar na lista. Ou seja, qualquer ponto que esteja a uma distância menor ou igual a diagonal do retângulo, $d_R$, de qualquer ponto do retângulo $R$ deve estar na lista (veja figura 2.4). Como a região resultante não é simples, escolheremos o menor círculo que englobe esta região, que será o círculo de raio $1.5d_R$ centrado no centro $C_R$ do retângulo $R$. Chamaremos esta região de $U_R$.

Caso não existam pontos na região $U_R$, colocaremos na lista $L_R$ o ponto mais próximo do centro $C_R$.

O pré-processamento, ou seja, a geração da malha e das listas $L_R$, demora um

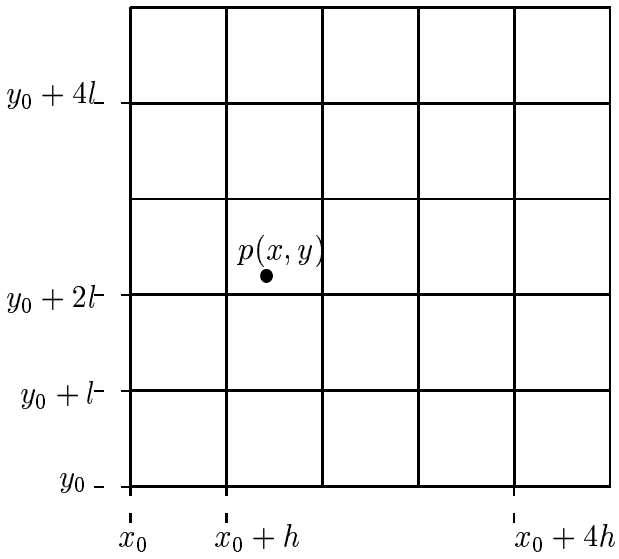



Figura 2.3: Subdivisão em retângulos de lados $l$ e $h$

tempo $\mathcal{O}(kn)$, onde $k$ é o número de retângulos, e a estrutura de dados (listas $L_R$) ocupam um espaço $\mathcal{O}(kn)$ também. Mas se os retângulos tiverem a proporção dos lados dentro de um certo intervalo, então cada ponto $x_i$ aparece no máximo em um número constante $S$ de listas ( $S \leq 27$ se o menor lado não é menor que a metade do maior). Isto faz com que o espaço seja $\mathcal{O}(n)$.

O algoritmo então fica da seguinte forma:

**Dados:** a malha de retângulos, as listas $L_R$, e o ponto $p$.

**Saída:** ponto $x$, o mais próximo de $p$.

**Passo 1:** Ache o retângulo $R$ da malha, onde se encontra o ponto $p$.

**Passo 2:** Calcule a distância de $p$ a todos os pontos da lista $L_R$ e ache o mais próximo, $x$.

Este algoritmo tem complexidade de pior caso igual a do algoritmo inicial, mas como o número de pontos em cada lista $L_R$ é bem menor que $n$ (na média $nS/k$, onde $S$ é a constante mencionada acima), o algoritmo se torna mais eficiente com o aumento do número de retângulos.

## 2.3 Fecho Convexo

O fecho convexo $FC(X)$ de um conjunto $X \subset \Re^d$ é a menor região convexa do $\Re^d$ que contém o conjunto $X$.

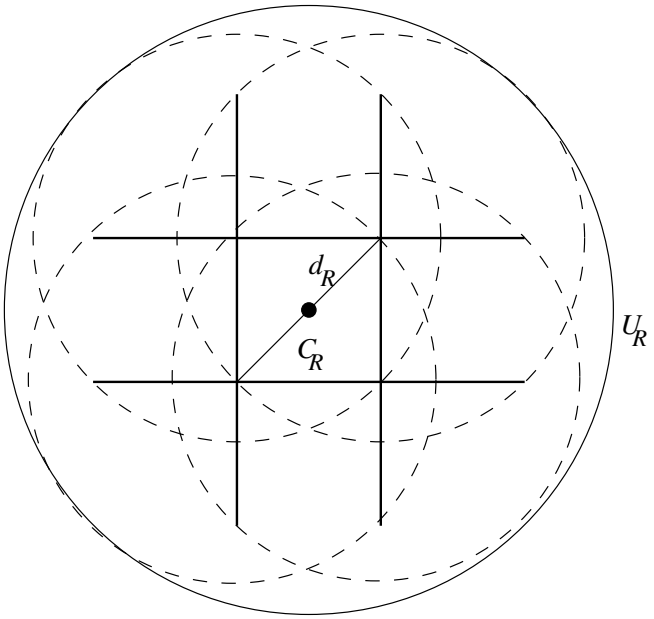



Figura 2.4: Região de pontos próximos

Achar o fecho convexo de um conjunto de pontos, por exemplo, é um problema que aparece quando queremos organizar o conjunto, agrupar os pontos em uma região simples.

Em quase todos os casos o conjunto $X$, do qual se quer obter o fecho convexo $FC(X)$, é um conjunto finito de pontos, e o fecho convexo será um politopo (veja figura 2.5).

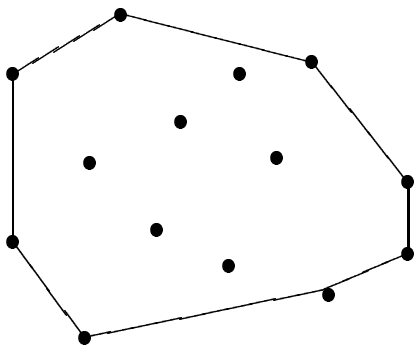


Figura 2.5: Fecho convexo

Como um politopo pode ser representado? Em dimensão 2, temos um polígono, que pode ser descrito por uma lista ordenada (no sentido anti-horário, por exemplo) de seus vértices. Em dimensão 3, temos um poliédro, que precisa de estruturas um pouco mais complicadas para armazenar as faces que compõem a fronteira.

Em dimensões maiores, as coisas se complicam ainda mais, pois precisariamos de

estruturas que representassem as faces de todas as dimensões.

Uma aplicação interessante para o fecho convexo seria na melhor organização dos pontos no problema anterior, onde queríamos saber qual o ponto do conjunto $X = \{x_1, x_2, \ldots, x_n\}$ mais próximo de um certo ponto $p$. Se o ponto $p$ está no interior de $FC(X)$, usamos o algoritmo que discutimos na seção anterior (com ou sem pré-processamento), se estiver fora, o ponto mais próximo será um dos vértices de $FC(X)$.

Perceba que precisamos usar aqui o algoritmo para saber se um certo ponto está no interior de um polígono.

O fecho convexo pode ser definido como o conjunto das combinações convexas dos pontos de $X = \{x_1, x_2, \ldots, x_n\}$.

$$FC(X) = \{p = \sum_{i=1}^{n} a_i x_i | \sum_{i=1}^{n} a_i = 1, a_i \geq 0\}$$

A fronteira $\partial FC(X)$ de $FC(X)$ é o conjunto onde no máximo $d$ índices são maiores que zero, onde $n$ é a dimensão do espaço.

E como construir o fecho convexo de um conjunto $X = \{x_1, x_2, \ldots, x_n\}$ de pontos do plano?

Dividindo o conjunto $X$ em dois usando uma reta vertical, por exemplo, que passe no meio dos pontos, teremos os conjuntos $X_1$ e $X_2$. Se calcularmos os fechos $FC(X_1) = (x_{i_1}, x_{i_2}, \ldots, x_{i_k})$ e $FC(X_2) = (x_{i_{k+1}}, x_{i_{k+2}}, \ldots, x_{i_n})$ de $X_1$ e $X_2$, ambos ordenados no sentido anti-horário, o fecho convexo de $X$ pode ser calculado como parte dos dois fechos mais duas novas arestas, que chamaremos de pontes. Veja na figura 2.6. Assim, calculamos os dois fechos, $FC(X_1)$ e $FC(X_2)$, recursivamente, até que tenham 3 ou menos vértices (quando o fecho é trivial). Só resta saber como achar as pontes.

Estas pontes podem ser encontradas facilmente a partir de aproximações sucessivas. Escolhemos os pontos mais próximos da reta, um de cada lado, digamos $x_{i_u} \in X_1$ e $x_{i_v} \in X_2$, e elevamos (descemos) o segmento $x_{i_u} x_{i_v}$, ora trocando o primeiro, ora trocando o segundo (veja figura 2.7).

O algoritmo seria o seguinte:

Faça $a = u$ e $b = v$;
Repita
  Enquanto o ângulo do segmento $x_{i_a} x_{i_b}$ para o segmento $x_{i_{prox(a)}} x_{i_b}$
  for no sentido horário faça
    $a \leftarrow prox(a)$;
  Enquanto o ângulo do segmento $x_{i_a} x_{i_b}$ para o segmento $x_{i_a} x_{i_{ant(b)}}$
  for no sentido anti-horário faça
    $b \leftarrow ant(b)$;
até que não ocorra mudanças;

Onde a operação *prox* em $a$ retorna o próximo vértice, e *ant* em $b$, o anterior na lista de vértices, que deve ser considerada circular.

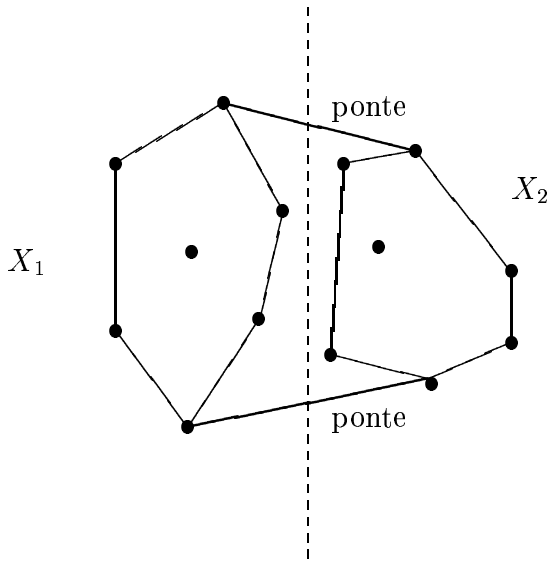



Figura 2.6: Pontes para construção do fecho convexo

O segmento $x_{i_a} x_{i_b}$ resultante é uma das pontes. Faça a mesma coisa para a outra ponte, sendo que $a = v$ e $b = u$.

Desta forma teremos complexidade $\mathcal{O}(n)$ para calcular as pontes, já que percorreremos os vértices dos fechos $FC(X_1)$ e $FC(X_2)$ no máximo 3 vezes (uma para achar os pontos mais próximos da reta, e uma para cada ponte).

Como o algoritmo se baseia na divisão do problema em dois menores e depois unifica as soluções, a complexidade final fica $\mathcal{O}(n \log n)$ se as divisões forem sempre próximas da metade. No pior caso, quando as divisões são totalmente desproporcionais, a complexidade sobe para $\mathcal{O}(n^2)$.

Este comportamento é semelhante ao algoritmo de ordenação *quicksort*.

Existem diversos outros algoritmos para resolver este problema, mas devido à semelhança com ordenação, o limite mínimo para este problema é $\Omega(n \log n)$. Para referência a um outro algoritmo, veja [Pre79].

## 2.4 Triangulação

Outra forma de organizar pontos, é definir sobre eles uma relação de vizinhança, ou seja, ligar pontos segundo algum critério de forma a estabelecer uma subdivisão do espaço que ocupam.

Uma destas organizações é a triangulação, que, embora o nome sugira, não se limita a dividir o plano em triângulos, mas qualquer espaço em simplexos.

Simplexos são extensões de triângulos em outras dimensões, como segmentos de reta, tetraedros, e etc. Chamamos de faces de um simplexo, todos os subsimplexos que o formam, como os vértices e arestas de um triângulo.

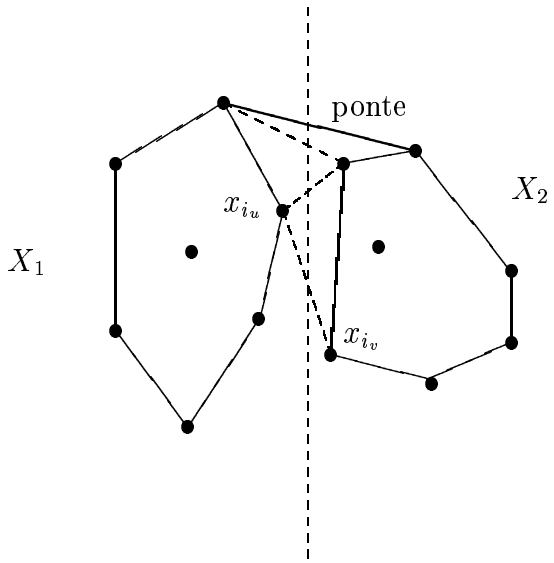



Figura 2.7: Algoritmo para encontrar as pontes

Mais uma vez vamos nos limitar ao plano, para não complicar muito, já que o objetivo não é esse.

Uma triangulação $\mathcal{T}$ é um conjunto de simplexos[1] tais que:

- Todas as faces de $t \in \mathcal{T}$ também pertencem a $\mathcal{T}$;
- se $a, b \in \mathcal{T}$ e $a \cap b \neq \emptyset$ então existe $c \in \mathcal{T}$ tal que $c = a \cap b$.

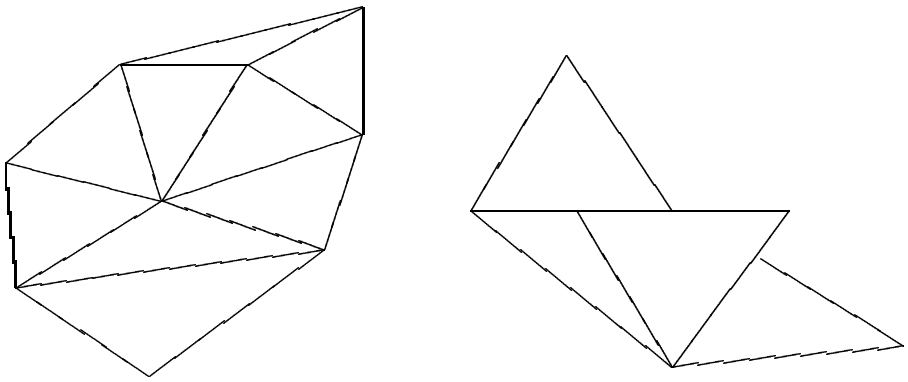
Assim, a figura 2.8 (a) apresenta uma triangulação, enquanto que na figura 2.8 (b) a segunda propriedade não é satisfeita.

Uma triangulação qualquer nem sempre é adequada para um certo problema. Para a maior parte dos algoritmos que precisam de triangulações, os triângulos precisam ser "gordinhos", parecidos com o triângulo equilátero. Essas triângulações especiais usam critérios de "gordura" para seus triângulos. Uma destas é a triangulação de Delaunay, que usa o seguinte critério:

> *Um certo triângulo $t$ faz parte da triangulação de Delaunay $\mathcal{T}$ somente se o círculo formado pelos vértices do triângulo não contém nenhum outro vértice de $\mathcal{T}$ em seu interior.*

Na verdade este critério é fraco, pois podem existir 4 vértices co-circulares, $a, b, c$ e $d$, sendo que não existam outros pontos no interior do círculo, o que faria com que os 4 triângulos, $abc$, $abd$, $acd$ e $bcd$, satisfizessem o critério, mas somente 2 deles poderiam fazer parte da triângulação. Isso faz com que a triangulação de Delaunay não seja única.

[1] Uma triangulação é um *Complexo Simplicial* ([Ede87, PS85])

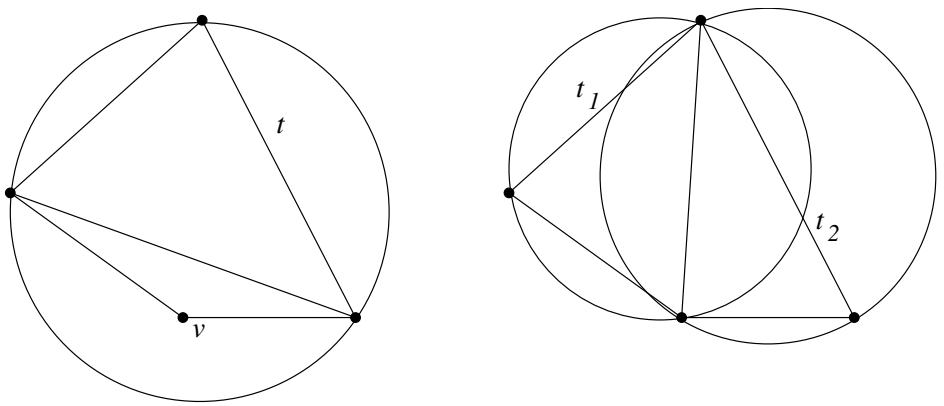


(a) (b)

Figura 2.8: a) Exemplo de triangulação, b) coleção qualquer

Se assumirmos que o conjunto de pontos não contém 4 pontos na situação acima, podemos garantir que a triangulação é única, e o critério, além de condição necessária, passa a ser também uma condição suficiente.

Na figura 2.9 vemos ilustrado o critério acima, e podemos observar que, se uma certa aresta atrapalha o critério, podemos trocá-la por outra.




(a) (b)

Figura 2.9: Critério de Delaunay

Podemos reescrever este critério com relação às arestas, e teríamos o seguinte teorema:

**Teorema 2.1 (Critério de Delaunay).** *Uma aresta uv pertence a triangulação de Delaunay $\mathcal{T}$ se, e somente se, existe um círculo c do qual uv é corda e que não contém nenhum vértice de $\mathcal{T}$ em seu interior.*

**Prova:** A necessidade é trivial:

Seja $t \in \mathcal{T}$ um triângulo. Suas 3 arestas estão em $\mathcal{T}$. O círculo que passa pelos seus vértices tem suas arestas como cordas e não possui nenhum vértice em seu interior.

Agora a suficiência:

Suponha que $uv$ seja tal que existe um círculo $c$ como no enunciado do teorema. Posso escolher $c$ de forma que seja maximal, ou seja, aumentar o seu raio faria com que algum ponto passasse a fazer parte de seu interior. Existem no máximo 2 e no mínimo 1 círculo nessa situação (veja figura 2.10). Escolha $c$ de forma que o raio seja o menor entre os maximais. Como $c$ é maximal, existe um vértice, $w$, na fronteira de $c$. Então $c$ passa por 3 vértices de $\mathcal{T}$ e não contém nenhum outro vértice em seu interior, logo o triângulo $uvw$ faz parte de $\mathcal{T}$, e consequentemente suas 3 arestas também. □

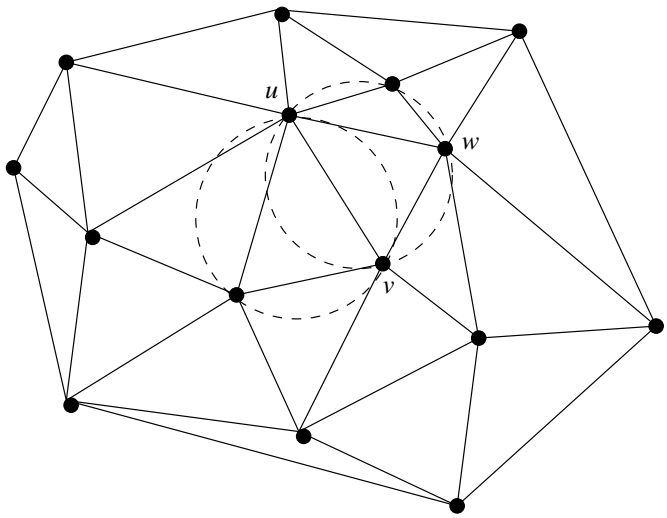



Figura 2.10: 2 círculos maximais da aresta $uv$

Este critério sugere um algoritmo simples, a partir de uma triangulação qualquer, procurar as arestas que atrapalham o critério e trocá-las por suas opostas, como na figura 2.9. Certamente este algoritmo termina, já que ao trocar uma aresta nos aproximamos da triangulação de Delaunay. Mas não é muito eficiente.

Que tal um algoritmo incremental? Partindo de uma triangulação inicial, com 3 vértices não necessariamente no conjunto, mas que formem um triângulo que envolva todos os outros, incluir os vértices um a um, descobrindo em qual triângulo eles estão, subdividindo este triângulo em três, como mostra a figura 2.11, e atualizando a triangulação. As três novas arestas respeitam o critério de Delaunay (veja [GS85]), mas as arestas vizinhas não se sabe. Como elas podem ser trocadas, é preciso que o processo seja expandido até que todos estejam em ordem.

Esta expansão não vai muito longe na prática, pelo seguinte argumento: para as arestas estarem erradas, é preciso que o novo ponto esteja razoavelmente próximo delas para que atrapalhe o critério de Delaunay, já que nenhum outro ponto atrapalhava

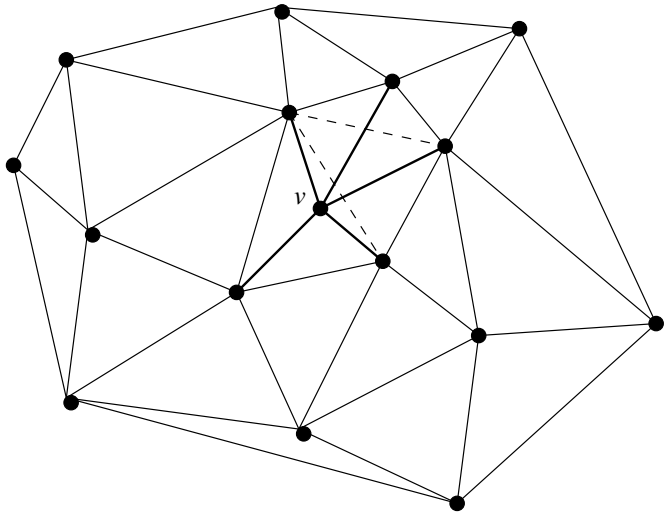



Figura 2.11: Divisão de um triângulo em 3

até o momento. Outro detalhe importante é o fato de que todas as arestas novas são incidentes ao novo ponto (veja [GS85, lema 7.4]).

No pior caso este algoritmo tem complexidade $\mathcal{O}(n^2)$, porém é considerado um bom algoritmo.

# Capítulo 3

# Técnicas Básicas

Neste capítulo apresentaremos algumas das técnicas usadas para resolver problemas geométricos. Algumas destas técnicas são usadas em algoritmos genéricos, e outras são específicas de geometria computacional.

## 3.1 Dividir-para-conquistar

Esta técnica é muito usada em toda ciência da computação. Um exemplo bastante conhecido é o algoritmo para ordenação *quicksort*, que pode ser encontrado em quase todos os livros sobre algoritmos. O algoritmo para achar o fecho convexo de um conjunto de pontos no plano apresentado na seção 2.3 é outro exemplo.

A base desta técnica é a recursão, dividindo o problema em duas instâncias menores, resolvendo estas instâncias recursivamente e depois agrupando as soluções.

Um fato bastante importante diz respeito à complexidade de algoritmos que usam esta técnica:

> *Se um certo problema é dividido exatamente ao meio em um algoritmo com a técnica* dividir-para-conquistar *e os procedimentos para a divisão e unificação juntos têm complexidade* $\mathcal{O}(n)$*, então o algoritmo tem complexidade* $\mathcal{O}(n \log n)$*. E se esses procedimentos tem complexidade* $\mathcal{O}(1)$*, então o algoritmo tem complexidade* $\mathcal{O}(\log n)$*.*

## 3.2 Transformações Geométricas

Conhecida também como *redução* , esta técnica se baseia na redução de um problema em outro, utilizando transformações geométricas.

Uma equivalência interessante existe entre calcular a triangulação de Delaunay e o fecho convexo $3D$ de pontos sobre a superfície de um parabolóide (veja [GS89]).

Um outro exemplo mais simples é entre ordenação de números reais e o fecho convexo $2D$ de pontos sobre uma parábola (veja [FC91]).

Com esta técnica podemos descobrir limites inferiores para a complexidade de alguns algoritmos, como por exemplo o de fecho convexo. Como o fecho convexo resolve o problema da ordenação, que tem limite inferior $\Omega(n \log n)$, então não pode ter complexidade menor que isso.

## 3.3 Dualidade

Muitas estruturas possuem uma espécie de *inversa*, como os números reais, as matrizes, e etc. Esta dualidade, como é chamada, pode se apresentar de diversas formas. Suas características são tais que o dual de um objeto (primal) deve ter estrutura semelhante, e o dual do dual é o primal.

Esta técnica já é usada há muito tempo, por exemplo em programação linear, onde podemos resolver o problema dual ao invés do primal.

Aqui apresentaremos o *Diagrama de Voronoi* como sendo o dual da *Triangulação de Delaunay*. Normalmente o Diagrama de Voronoi é apresentado primeiro.

Se em cada circuncentro de cada triângulo da Triangulação de Delaunay for colocado um vértice, e entre cada par de novos vértices que estiverem em triângulos vizinhos, for colocada uma aresta coincidente com a mediatriz, teremos o Diagrama de Voronoi do conjunto de pontos iniciais.

Como visto na seção 2.2, o Diagrama de Voronoi divide o plano em regiões onde o ponto mais próximo de todos dentro de cada uma é o ponto inicial que está dentro dela.

O Diagrama de Voronoi é bastante útil, e pode ser encontrado facilmente através da Triangulação de Delaunay, em tempo linear, portanto preciso de tempo $\mathcal{O}(n \log n)$ para achá-lo.

Outra dualidade interessante é a de polígonos. Seja um polígono $P$ de vértices $x_1, x_2, \ldots, x_n$. O polígono $Dual(P)$, dual de $P$, é tal que seus vértices são $\frac{1}{v_1}, \frac{1}{v_2}, \ldots, \frac{1}{v_n}$, onde $v_i$ é o ponto da aresta $x_i x_{i+1}$ mais próximo da origem (considere $x_{n+1} = x_1$), ou seja, $v_i$ é perpendicular à aresta $x_i x_{i+1}$ com módulo igual a distância dela a origem.

Um resultado bastante surpreendente, que usa a dualidade é o seguinte:

Sejam $P$ e $Q$ dois polígonos planares, $Dual$ o operador de dualidade como descrito acima, e $FC$ o operador de fecho convexo, então

$$P \cap Q = Dual(FC(Dual(P) \cup Dual(Q))).$$

Este resultado nos diz que para calcular a intersecção de dois polígonos, posso calcular o fecho convexo de seus duais, e depois dualizar. Como achar o dual é linear, temos uma relação entre intersecção de polígonos e o fecho convexo no plano.

## 3.4 Varredura do Espaço

A técnica de varredura é uma das mais importantes, sendo usada em diversos algoritmos. Ela se apresenta como uma variação de algoritmos incrementais, onde o que é

incrementado é a posição de um hiperplano, que vai "varrendo" o espaço e efetuando operações.

Um exemplo bastante ilustrativo seria o de pintar um polígono planar. Com uma reta horizontal passando de baixo para cima, temos anotado a cada instante as intersecções desta reta com as arestas do polígono, e com o mesmo argumento usado para saber se um ponto está dentro ou fora de um polígono (seção 2.1), pinto os pedaços de reta que estão dentro.

As técnicas de varredura usam dois tipos de estruturas de dados, uma chamada de *agenda*, que guarda os eventos futuros que já foram previstos, e outra com o nome de *jornal*, que representa os acontecimentos do momento. Com os acontecimentos do jornal, a agenda pode ser atualizada, removendo ou inserindo eventos.

Veremos isso no seguinte problema, dados $n$ segmentos de reta no plano, descobrir todas as intersecções entre estes segmentos.

Com a técnica de varredura, teremos uma linha, por exemplo, vertical, indo da esquerda para a direita, e a agenda deve conter, inicialmente os vértices mais a esquerda de todos os segmentos, ou seja, $n$ vértices. Estes eventos na agenda, marcam em que momento a linha vai encontrar os segmentos (veja figura 3.1).

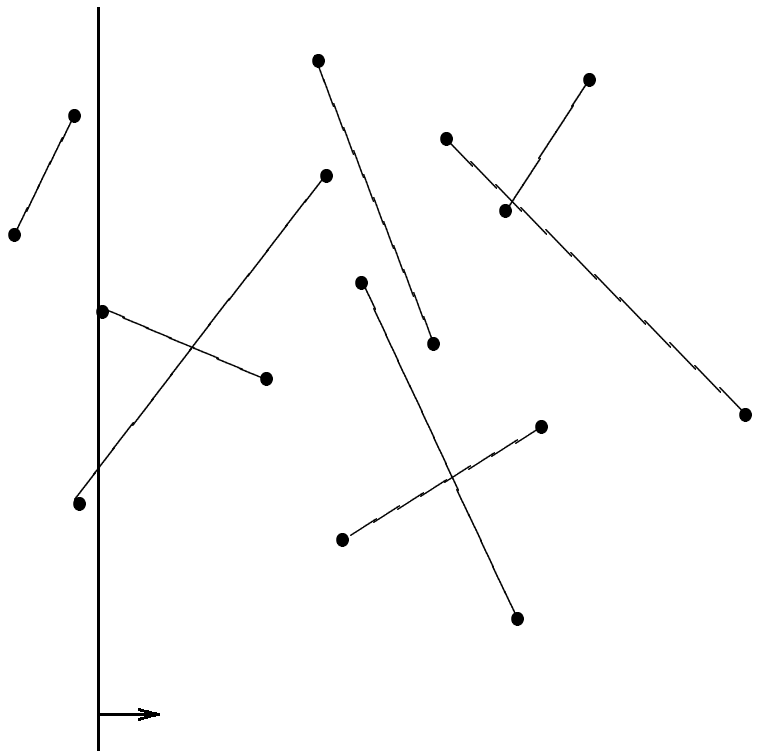


Figura 3.1: Intersecção de segmentos por varredura

A linha então, passa para o primeiro vértice, e o jornal registra que a linha está cruzando o primeiro segmento. Quando a linha chegar ao segundo vértice, o jornal já registra que existem dois segmentos em contato com a linha. Neste ponto, podemos

calcular a intersecção, se houver, destes dois segmentos. Assim, a cada novo segmento encontrado, calculamos suas intersecções com os segmentos do jornal.

A cada segmento encontrado, colocamos na agenda o seu ponto final, para que possamos retirá-lo do jornal quando ele terminar.

Este tem complexidade de pior caso $\mathcal{O}(n^2)$, mas como só calculo intersecções entre segmentos do jornal, na prática a complexidade é quase linear.

Outros problemas, como intersecção de poliédros, podem ser resolvidos com esta técnica (veja [HMMN84]).

## 3.5 Estruturas Hierárquicas

Muitos algoritmos precisam de estruturas de dados eficientes para que possam executar suas tarefas no tempo esperado. Vejamos então algumas destas estruturas. Daremos ênfase às estruturas hierárquicas, visto que são as mais comuns além, é claro, das listas.

Normalmente uma hierarquia é representada por uma árvore, o que também tem muita relação com os algoritmos recursivos.

Uma generalização das árvores binárias de busca, para dimensões maiores é a árvore $k-d$, ou $k$-dimensional. Esta árvore é uma árvore binária, mas a cada nível a chave de busca é trocada.

Suponha que temos um banco de dados com o nome, salário, idade, e altura dos funcionários de uma firma. Se quisermos saber quais os funcionários que tem altura entre $h_{min}$ e $h_{max}$, idade entre $a_{min}$ e $a_{max}$ e salário entre $s_{min}$ e $s_{max}$, como resolver?

Seja $T$ uma árvore binária com todos os registros do banco de dados, de forma que no nível 0 (raiz) a chave é o salário, no nível 1, é a idade, no nível 3, a altura, e no nível 4 voltamos ao salário.

Assim, podemos saber quais os registros que satisfazem à restrição acima fazendo algumas buscas binárias. Na verdade posso precisar descer por dois ramos da árvore, e o algoritmo tem pior caso linear (pois todos os registros podem estar na resposta). Perceba que a árvore deve estar balanceada.

Vejamos um exemplo no plano. Dado um conjunto $P = \{p_1, p_2, \ldots, p_n\}$ com $n$ pontos no plano, (veja figura 3.2), vamos construir uma árvore $2-d$ e usá-la para resolver o problema de saber quais destes pontos está dentro de uma região retangular $R$.

Ordene os pontos por uma de suas coordenadas, digamos $x$, e escolha a mediana $p_{i_0} = (x_{i_0}, y_{i_0})$. Este ponto será a raíz da árvore. Sejam $P_e$ e $P_d$ dois subconjuntos de $P$ tais que $P_e \cup P_d = P - \{p_{i_0}\}$, e os pontos com coordenada $x$ menor ou igual a $x_{i_0}$ estão em $P_e$, e os com coordenada $x$ maior que $x_{i_0}$, em $P_d$. Estes dois conjuntos serão as sub-árvores esquerda ($P_e$) e direita ($P_d$) de $p_{i_0}$.

Repita este processo, escolhendo a proxima coordenada, no caso $y$, para os conjuntos $P_e$ e $P_d$, montando assim a árvore $T$ da figura 3.3 em tempo $\mathcal{O}(n \log n)$. Você consegue explicar o motivo desta complexidade?

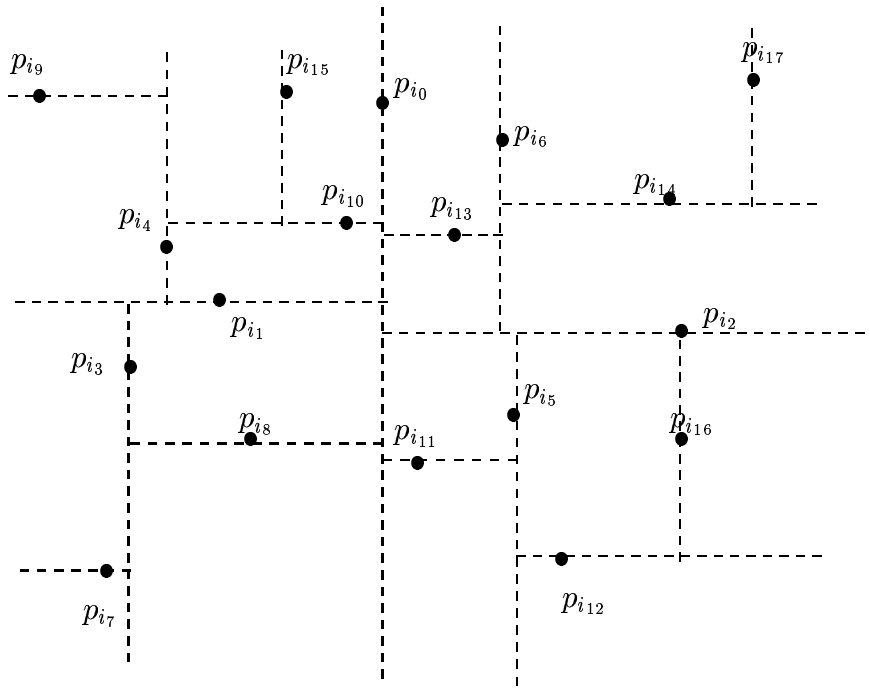



Figura 3.2: Construção da árvore $2-d$

Com a árvore construída, passemos para o problema. Dada uma região $R$ do plano, enumerar os pontos do conjunto $P$ que estão no interior de $R$. O que faremos é descer na árvore decidindo se cada ponto no caminho está no interior de $R$, e se algum dos lados da árvore pode ser descartado. Suponha que a variável $R$ tenha dois campos, *menor* e *maior*, cada um sendo um vetor de duas posições, representando os dois pontos extremos de $R$, inferior esquerdo e superior direito. Suponha também que os nós da árvore são estruturas, que além dos ponteiros *esq* e *dir*, têm o campo *ponto* que é o ponto propriamente dito.

Apresentaremos o procedimento *Busca* que recebe o raíz de uma árvore, um índice da coordenada usada como chave (1 para $x$ e 2 para $y$), e a região $R$, e tem como saída o conjunto de pontos da árvore que estão no interior de $R$.

O procedimento é o seguinte:

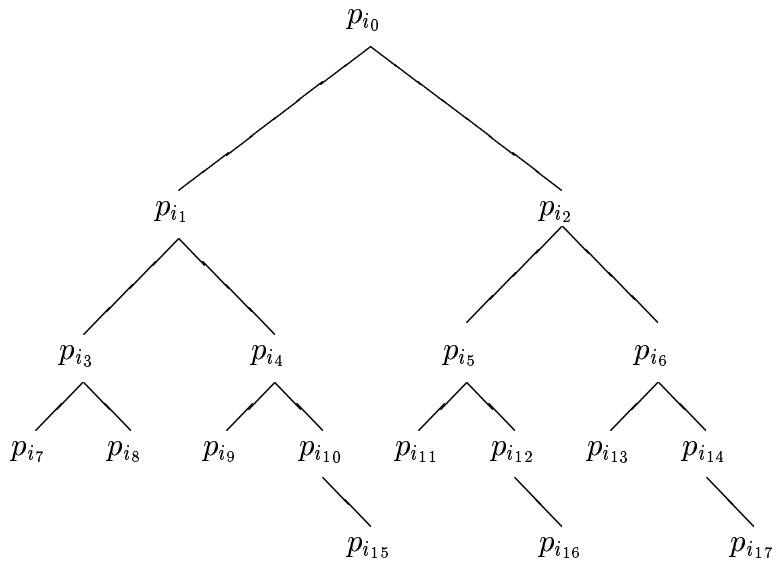



Figura 3.3: Árvore $2-d$

```
Busca ( v, chave, R )
{
    /* cálculo da outra chave */
    outra ← 3− chave;
    se v.ponto[chave] < R.menor[chave] então
        Busca ( v.dir, outra, R );
    senão
    {
        se v.ponto[chave] > R.maior[chave] então
            Busca ( v.esq, outra, R );
        senão
        {
            /* se o ponto está dentro de R */
            se (v.ponto[outra] ≥ R.menor[outra])
            & (v.ponto[outra] ≤ R.maior[outra]) então
                inclua v na resposta;
            Busca ( v.esq, outra, R );
            Busca ( v.dir, outra, R );
        }
    }
}
```

Para resolver o nosso problema, basta executar *Busca ( Raíz(T), 1, R )*, onde *Raíz(T)* retorna o ponteiro para o nó raíz da árvore $T$.

# Capítulo 4

# Problemas Maiores

Até agora tratamos de problemas simples, mas a Geometria Computacional, embora seja atraente, com muitas técnicas, também tem problemas difíceis. Problemas como o de achar a intersecção de dois poliédros convexos pode parecer fácil, mas se o objetivo é achar o algoritmo ótimo, então começa a complicar.

O artigo [Cha89], de Chazelle, estuda o problema acima, e afirma existir um algoritmo de complexidade $\mathcal{O}(n+m)$, onde $n$ e $m$ são os tamanhos dos dois polígonos. Este algoritmo usa e abusa de dualidade, fecho convexo, triangulação de Delaunay ($3D$ no caso). Não apresenta uma implementação, pois é muito teórico.

Um outro problema difícil é o de montagem/desmontagem de objetos formados por $n$ peças. Este problema foi estudado por Stolfi, e tem uma de suas soluções efetuando-se a subdivisão de um espaço de dimensão 5, no qual estão representados os movimentos que cada peça pode fazer.

Estes dois problemas são exemplos de como podemos complicar a vida de quem procura soluções ótimas para problemas simples.

# Referências Bibliográficas


[Cha89] Bernard Chazelle. An optimal algorithm for intersecting three-dimensional convex polyhedra. Technical report, Department of Computer Science, Princeton University, fevereiro 1989. (Extended Abstract at IEEE - 1989, pages 586 - 591).

[Ede87] Herbert Edelsbrunner. *Algorithms in Combinatorial Geometry*, volume 10 of *EACTS Monographs on Theoretical Computer Science*. Springer-Verlag, Germany, 1987.

[FC91] Luiz Henrique de Figueiredo and Paulo César Pinto Carvalho. *Introdução à Geometria Computacional*. IMPA / CNPq, Rio de Janeiro, RJ, 1991.

[GS85] L. Guibas and J. Stolfi. Primitives for the manipulation of general subdivisions and the computation of voronoi diagrams. *ACM Transactions on Graphics*, 4(2):74 – 123, abril 1985.

[GS89] L. Guibas and J. Stolfi. Ruler, compass, and computer: The design and analysis of geometric algorithms. Technical report, Digital Equipments Coorporation - DEC, Systems Research Center, Palo Alto, California, USA., fevereiro 1989.

[HMMN84] Stefan Hertel, Martti Mäntylä, Kurt Mehlhorn, and Jurg Nievergelt. Space sweep solves intersection of convex polyhedra. *Acta Informatica*, 21:501 – 519, 1984. Springer-Verlag.

[Pre79] F. P. Preparata. An optimal real-time algorithm for planar convex hulls. *Communications of the ACM*, 22(7):402 – 405, julho 1979.

[PS85] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introdution*. Springer-Verlag, New York, USA, 1985.