Luiz Henrique de Figueiredo
e Paulo César Pinto Carvalho

Introdução à Geometria Computacional

LUIZ HENRIQUE DE FIGUEIREDO e PAULO C.P. DE CARVALHO
Instituto de Matemática Pura e Aplicada
Estrada Dona Castorina, 110
Jardim Botânico
22460 - Rio de Janeiro-RJ

ISBN
85-244-0061-7

Conselho Nacional de Desenvolvimento Científico e Tecnológico
INSTITUTO DE MATEMÁTICA PURA E APLICADA
Estrada Dona Castorina, 110
22.460 - Rio de Janeiro-RJ

Prefácio

Este livro se originou em notas de aulas de cursos dados na PUC-Rio em 1990 e no IMPA em 1991. Os alunos destes cursos estavam tipicamente no início de um programa de Mestrado em Computação ou Matemática, mas nem todos possuíam pré-requisitos formais em Computação (análise de complexidade de algoritmos, estruturas de dados). Na realidade, muitos dos alunos estavam mais interessados nas aplicações, principalmente à Computação Gráfica, do que no estudo teórico de algoritmos ótimos.

Este curso apresenta uma introdução aos aspectos teóricos dos algoritmos geométricos sem perder de vista a necessidade de implementá-los na prática. Deste modo, são apresentados alguns algoritmos que, apesar de não serem ótimos, podem ser implementados com facilidade. Na verdade, todos os algoritmos apresentados neste livro foram implementados e estão disponíveis em disco para os leitores.

O conteúdo deste livro sofreu influência de diversos trabalhos. Em especial, destacamos os textos de Preparata-Shamos [PS] e Guibas-Stolfi [GS2] sobre Geometria Computacional e de Sedgewick [S] sobre algoritmos em geral.

Agradecemos a André Antunes Nogueira da Silva pela correção de vários erros e a Carlos Gustavo Tamm de Araújo Moreira por sugestões que permitiram simplificar alguns algoritmos.

Agradecemos também à Comissão Organizadora do 18º Colóquio Brasileiro de Matemática pela oportunidade de divulgar estas notas.

Rio de Janeiro, maio de 1991.

Paulo Cezar Pinto Carvalho
Luiz Henrique de Figueiredo

Índice

1

Preliminares

1.1 Introdução

O objetivo deste curso é apresentar uma introdução à **Geometria Computacional**, que é a disciplina que faz um estudo sistemático de algoritmos para problemas geométricos.

Geometria Computacional—pelo menos com esse nome e com o enfoque atual—é uma disciplina extremamente nova. O primeiro livro publicado sobre o assunto [PS] é de 1985 e a maioria dos trabalhos é também bastante recente. O enfoque atual em Geometria Computacional é estudar problemas geométricos sob o ponto de vista da **Análise de Complexidade de Algorimos**, que também é uma área recente [GJ].

No entanto, as origens da Geometria Computacional são muito mais antigas e ligadas à evolução da Geometria Euclidiana. Uma parte relevante do estudo clássico de Geometria Euclidiana está ligada ao estudo de construções geométricas. O caráter algorítmico presente neste estudo é bastante claro: existe um certo número de operações elementares, a serem executadas com régua e compasso, a partir das quais as construções desejadas deverão ser efetuadas.

Eis um possível elenco de construções elementares para a Geometria Clássica:

- C_1: dados dois pontos a e b, obter a reta definida por eles;
- C_2: traçar um círculo de centro o e raio igual à medida do segmento ab;
- C_3: obter a interseção de duas retas, de dois círculos, ou de uma reta e um círculo.

Este conjunto de construções elementares é suficiente para as chamadas **construções com régua e compasso.**

Tomemos, por exemplo, o problema de obter o círculo que contém três pontos não colineares a, b e c dados. A solução do problema consiste em obter o centro do círculo, que é a interseção das mediatrizes dos segmentos ab, ac e bc. Para obter tal ponto, podemos usar a seguinte sequência de construções elementares:

- aplicamos C_2 para traçar, com centros em a e b, círculos de mesmo raio (por exemplo, de medida igual à do segmento ab);
- usamos C_3 para obter os pontos p e q de interseção destes círculos (a existência de pontos de interseção requer que o raio seja maior que a metade de ab, o que ocorre caso o raio seja tomado igual a ab);
- usamos C_1 para obter a reta definida por p e q, que é a mediatriz de ab;
- aplicamos a sequência acima para obter a mediatriz de ac;
- usamos C_3 para obter o centro o do círculo como a interseção das mediatrizes de ab e ac.

Outras idéias presentes no moderno tratamento de complexidade computacional também podem ter suas origens associadas ao estudo de construções geométricas. Lemoine, por exemplo, introduziu o conceito de **simplicidade** de uma construção, definida pelo número de construções elementares que a compõem. Esta noção corresponde à noção moderna de complexidade de um algoritmo. Deve-se frisar, no entanto, que uma boa parte da motivação para o desenvolvimento da teoria da complexidade computacional parece ter estado ausente dos estudos clássicos de construções geométricas. Atualmente, é de grande importância a análise do desempenho de algoritmos em função do tamanho do problema (isto é, da quantidade de dados que deverá ser processada pelo algoritmo). Na geometria euclidiana clássica, os problemas de interesse tem tamanho limitado e, por esta, razão, este tipo de análise não é relevante.

De forma análoga ao estudo de construções geométricas euclidianas, a Geometria Computacional moderna se ocupa de algoritmos para a resolução de problemas geométricos. A diferença é que estes problemas geométricos são agora representados por dados armazenados em computador. O papel das construções geométricas, por sua vez, é representado por algoritmos que operam sobre estes dados. Desta forma, para estabelecer os fundamentos teóricos da Geometria Computacional, é necessário escolher um modelo computacional que estabeleça como um problema geométrico pode ser representado num computador e que fixe as

operações elementares (ou **primitivas**) que irão operar sobre estas representações. Na próxima seção, discutimos um possível modelo computacional a ser adotado.

1.2 Modelos de complexidade computacional

Uma importante parte da Teoria da Computação tem por objetivo o estudo de modelos de computação, que procuram abstrair e simplificar o comportamento de computadores digitais. O mais conhecido destes modelos é a **máquina de Turing**, utilizado para estabelecer resultados básicos em computabilidade e complexidade computacional. Recentemente, Blum, Shub e Smale propuseram um modelo mais apropriado para o estudo de algoritmos que envolvem aritmética real [BSS].[1] O estudo formal de tais modelos e das importantes questões teóricas a ele associados foge ao objetivo deste curso.

Neste texto, vamos descrever informalmente o modelo computacional a ser adotado e enunciar, sem demonstração, os resultados de interesse para a Geometria Computacional. Para um tratamento mais preciso do assunto, o leitor pode consultar, por exemplo, [Me].

Inicialmente, procurarmos estabelecer, informalmente, os conceitos de **problema**, **instância** de um problema e **algoritmo** para resolver um problema. Para tornar mais concreta nossa discussão, consideremos um exemplo específico de um problema:

PONTO EM POLÍGONO: *Dado um polígono plano simples*[2] *P e um ponto p do plano, decidir se p é interior ou não ao polígono P.*

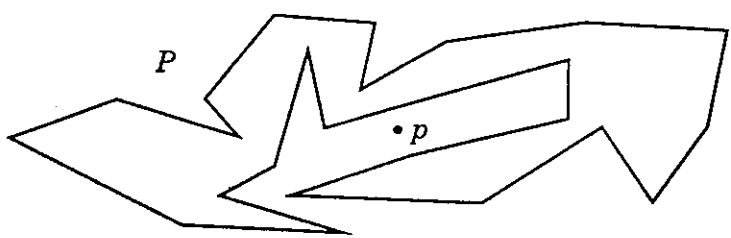


Figura 1.1 – Problema PONTO EM POLÍGONO.

[1] Veja também o livro texto [Bl], apresentado neste Colóquio.
[2] Um polígono simples é um polígono em que os lados não se cruzam, a não ser nos vértices.

Cada par (P, p) caracteriza uma instância específica deste problema. Um algoritmo para PONTO EM POLÍGONO é uma lista de passos que, dada qualquer instância (P, p), pára após um número finito de passos com a conclusão correta a respeito da inclusão de p no polígono P.

Algumas questões de interesse para este problema poderiam ser:

• Existe algum algoritmo que resolve o problema? (Isto exige que o algoritmo pare após um número finito de passos para qualquer instância do problema).

• Dado um certo algoritmo A para o problema, quão eficiente é este algoritmo? Dados dois algoritmos A e B, qual deles é superior?

• Dentre todos os algoritmos que resolvem o problema, qual deles é o melhor?

Para responder à primeira pergunta, é preciso definir que algoritmos (isto é, que passos) são válidos. Para isto é necessário, antes de mais nada, estabelecer a natureza dos dados sobre os quais as instruções de tais algoritmos devem operar. Os problemas de interesse para a Geometria Computacional são tais que suas instâncias podem ser definidas por um conjunto finito de pontos $x_1, x_2, ..., x_n$ do espaço euclidiano d-dimensional $\mathbf{R}^d$ (mais comumente do $\mathbf{R}^2$ ou $\mathbf{R}^3$). Cada um destes pontos x_i é representado por suas coordenadas, que são números reais. Portanto, em última análise, os algoritmos de interesse devem operar sobre números reais. Assim, modelos computacionais convenientes para a Geometria Computacional são aqueles nos quais:

• cada elemento de armazenamento de dados é capaz de armazenar um único número real;

• são considerados válidos quaisquer passos envolvendo operações aritméticas (+, –, x, /), extração de raiz quadrada e comparações entre números reais.

Um modelo computacional com estas características e muito usado para obtenção de resultados teóricos em computação é o de **árvores de decisões algébricas**. Neste modelo, um algoritmo é representado por uma árvore em que cada nó representa um passo, no qual uma certa função algébrica é avaliada. De acordo com o sinal do resultado, o algoritmo prossegue para um dos descendentes do nó em questão.

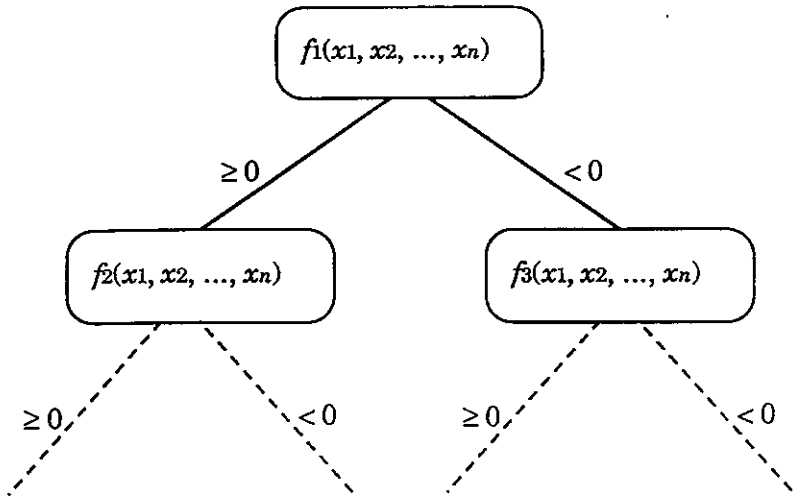


Figura 1.2 – Uma árvore de decisões algébricas.

O número de nós de uma árvore de decisão é, em geral, muito maior que o número de instruções presentes em um algoritmo linear com instruções de desvio (*branches*) mas pode-se demonstrar que os modelos são equivalentes, no sentido que cada um pode simular o outro (ver [Me] para se aprofundar nesta discussão).

A principal dificuldade para obter respostas para as duas últimas questões formuladas no início desta seção é estabelecer uma medida de eficiência para um algoritmo. Parece ser natural medir a **eficiência** (ou a complexidade) de um algoritmo pelo tempo necessário à sua execução, que por sua vez pode ser medido pelo número de passos elementares necessários ao seu término (no caso de uma árvore de decisão, este número é representado pela profundidade do nó terminal do algoritmo). No entanto, há ainda uma complicação séria: é concebível que dois algoritmos A e B sejam tais que A opera melhor sobre certas instâncias enquanto B opera melhor sobre outras.

Uma maneira de uniformizar este estudo é exprimir a complexidade do algoritmo em função do tamanho da instância. Ainda assim, permanece a dificuldade, já que, mesmo sobre instâncias de mesmo tamanho, o desempenho pode não ser uniforme. Há duas formas principais de lidar com este obstáculo: pode-se considerar um desempenho médio sobre todas as

instâncias de um certo tamanho n (admitindo que as instâncias ocorrem segundo alguma lei de distribuição de probabilidades) ou pode-se simplesmente considerar o caso mais desfavorável.

A segunda abordagem é mais comum, por dois motivos: primeiro, não é sempre que existe uma distribuição de probabilidade natural para as instâncias de um problema; em segundo lugar, o estudo probabilístico de algoritmos é, em geral, demasiadamente complexo e só se consegue obter resultados em casos relativamente simples.

Resumindo esta discussão, a complexidade de um algoritmo para resolver um certo problema P, será expressa por uma função f que, para cada natural n, dá o número $f(n)$ de passos necessários a resolver P na sua instância mais desfavorável de tamanho n.

Muitas vezes, porém, não é possível encontrar exatamente quantos passos são necessários para se resolver uma instância de tamanho n. Em lugar disso, nos contentamos em obter uma função f tal que, para alguma constante k e para n suficientemente grande, $kf(n)$ seja uma cota superior para o número de passos necessários a resolver uma instância de tamanho n. Dizemos, neste caso, que o algoritmo tem **complexidade assintótica** $O(f(n))$. Temos, assim, a seguinte

Definição 1.1: Um algoritmo A para resolver P tem **complexidade** $O(f(n))$ se existe uma constante $k > 0$ e um natural N tais que, para qualquer instância de P de tamanho $n > N$, o número de passos de A necessários para resolver esta instância é, no máximo, $kf(n)$.

A inclusão da constante multiplicativa k nesta definição tem várias consequências. Por um lado, ela automaticamente torna equivalente dois modelos computacionais em que as operações elementares de um possam ser simuladas através de um número limitado de operações elementares do outro. Isto faz com que, ao afirmarmos que um certo algoritmo tem complexidade $O(n^2)$, esta afirmativa seja válida para uma família ampla de modelos computacionais, em que, à família de operações elementares descrita acima, sejam acrescentadas outras primitivas mais convenientes (por exemplo, ângulo entre vetores). Qualquer conjunto de primitivas que requeiram um número limitado de operações elementares pode ser acrescentado ao modelo computacional sem alterar a complexidade de qualquer algoritmo.

Outra consequência (esta desfavorável) da inclusão da constante multiplicativa é que, na prática, algoritmos de mesma complexidade teórica podem ter desempenhos bem diferentes, já que a constante multiplicativa embutida na complexidade pode ser arbitrariamente grande. Além

disso, a análise tem caráter assintótico, já que o número de passos só precisa ser limitado por $kf(n)$ para valores de n superiores a um certo N. Na prática, pode ocorrer que os casos de interesse sejam exatamente aqueles em que n é menor que N.

De todo modo, a análise assintótica do caso mais desfavorável é a principal técnica de análise de algoritmos e fornece informação a respeito de como o tempo necessário para a execução do algoritmo cresce com o tamanho da instância. Um "bom" algoritmo é aquele no qual tal crescimento não seja muito rápido [E]. De um modo geral, complexidades polinomiais são aceitas como demarcando a fronteira entre bons e maus algoritmos. Uma área extremamente importante de complexidade computacional procura separar problemas para os quais se conhece algoritmos polinomiais de outros para os quais tais algoritmos não são conhecidos. Um dos mais importantes problemas em aberto na Teoria da Complexidade Computacional é obter um algoritmo polinomial para uma classe de problemas (chamados de problemas **NP-completos**) ou demonstrar a não existência de tais algoritmos [GJ].

1.3 Um exemplo: algoritmos para ordenação

Para sedimentar estas idéias, consideraremos exemplos de algoritmos que resolvam o problema a seguir.

ORDENAÇÃO: *Dados n números reais $x_1, x_2, x_3, ..., x_n$, colocá-los em ordem crescente.*

A escolha deste problema não é casual. Além de algoritmos para sua resolução serem simples de analisar, o problema de ordenação desempenha um papel relevante em Geometria Computacional, como veremos nos próximos capítulos.

Uma solução simples do problema é recorrer ao algoritmo de **ordenação por seleção**:

Algoritmo 1.1: Ordenação por Seleção

```
para i = 1..n-1
        m = i
        para j = i+1..n
              se x_j < x_m então m = j
        troque x_i com x_m
```

Para analisar tal algoritmo, a primeira observação é que o tamanho de uma instância é dada pelo número n de reais a serem ordenados. Na primeira execução do corpo do *loop* principal, são feitas $n-1$ comparações e 1 troca, num total de n operações. Na segunda execução, $n-2$ comparações e 1 troca, e assim por diante. Logo, o número de passos requerido pelo algoritmo é:

$$n + (n-1) + (n-2) + \ldots + 1 = \frac{n(n+1)}{2} = \frac{n^2}{2} + \frac{n}{2}.$$

Assim, o algoritmo tem complexidade $O(n^2)$.

Um algoritmo mais sofisticado é o chamado **Mergesort**. Este algoritmo usa um dos paradigmas fundamentais de construção de algoritmos. Este paradigma, que obtém a solução de um problema através da resolução de problemas de tamanho menor, é conhecido como **dividir-para-conquistar** e é o segredo da eficiência de muitos algoritmos. No Mergesort, ao invés de ordenar o conjunto de n números, dividimos este conjunto em dois conjuntos com $n/2$ elementos cada, ordenamos cada um destes conjuntos e reunimos estes conjuntos já ordenados. Para ordenar cada um dos conjuntos com $n/2$ elementos, aplicamos recursivamente o mesmo algoritmo. Temos, então, o seguinte algoritmo:

Algoritmo 1.2: mergesort(x,n)

```
se n<2 então retorne              {caso básico da recursão}
m = n/2
l = x[1..m]                       {separação}
r = x[m+1..n]
mergesort(l,m)                    {recursão}
mergesort(r,n-m)
i = 1
j = 1
para k = 1..n                     {combinação}
      se l_i < r_j então
            x_k = l_i;    i = i+1
      senão
            x_k = r_j;    j = j+1
```

(O código acima assume que existem elementos auxiliares (chamados **sentinelas**) tais que $l_{m+1} = +\infty = r_{n-m+1}$.)

Vejamos se este algoritmo é mais eficiente que o "Ordenação por Seleção" descrito acima. Para simplificar a análise, suponhamos que n é da forma $n = 2^p$, onde p é um inteiro positivo. Seja $T(n)$ a complexidade do algoritmo, expressa pelo número de passos necessários à sua execução. O algoritmo consiste em duas chamadas recursivas do mesmo algoritmo para instâncias de tamanho $n/2$, seguidas por uma etapa em que, a partir de dois conjuntos já ordenados, deve-se obter sua união, também ordenada. É fácil ver que esta união é obtida em um número de passos que é proporcional a n. Logo, a complexidade $T(n)$ é dada por:

$$T(n) = 2T(n/2) + kn,$$

onde k é uma constante. Por sua vez, $T(n/2)$ é dada por

$$T(n/2) = 2T(n/4) + kn/2\ ,$$

o que fornece

$$T(n) = 4T(n/4) + 2kn.$$

Continuando o mesmo processo, obtemos sucessivamente:

$$T(n) = 8T(n/8) + 3kn$$

...

$$T(n) = 2^pT(n/2^p) + pkn = nT(1) + k\ (\log_2 n)\ n.$$

Daí, concluímos que:

$$T(n) = O\ (n \log n).$$

Como o crescimento da sequência $n \log n$ é mais lento que o de n^2 (no sentido que $\lim_{n\to\infty} \frac{n \log n}{n^2} = 0$), este algoritmo é superior ao anterior do ponto de vista assintótico.

A análise acima é característica de algoritmos recursivos do tipo dividir-para-conquistar, isto é, de algoritmos que possuem a seguinte estrutura:

Algoritmo Dividir-para-Conquistar:

- **SEPARAÇÃO:** decompor a instância do problema em duas instâncias de tamanhos menores;
- **RECURSÃO:** aplicar recursivamente o mesmo algoritmo para cada uma destas instâncias;
- **COMBINAÇÃO:** combinar os resultados para obter a solução da instância original.

Uma análise análoga à utilizada para o Mergesort fornece o seguinte resultado para algoritmos com esta estrutura:

Teorema 1.1: *Se um algoritmo recursivo do tipo dividir-para-conquistar para resolver um problema decompõe uma instância de tamanho n em duas instâncias de tamanho n/2 do mesmo problema, e se o processamento necessário à execução das etapas de SEPARAÇÃO e COMBINAÇÃO tem, no total, complexidade O(n), então o algoritmo resultante tem complexidade O(n log n).* ■

Utilizaremos este princípio diversas vezes nos próximos capítulos para obter algoritmos com esta complexidade.

Um outro algoritmo para ordenação que também utiliza o paradigma de dividir para conquistar é o chamado **Quicksort** [Ho]. Seu nome é indicativo do bom desempenho alcançado pelo algoritmo na prática. Neste algoritmo, toma-se um elemento do conjunto (por exemplo, x_1) e a etapa de SEPARAÇÃO consiste em dividir o conjunto a ser ordenado em dois subconjuntos: o primeiro é composto por todos os elementos menores que x_1 e o segundo por todos os elementos maiores ou iguais a x_1. Cada um destes conjuntos é ordenado utilizando recursivamente o mesmo algoritmo. A etapa de COMBINAÇÃO é completamente trivial neste algoritmo, já que a sequência ordenada consiste em listar os elementos (já ordenados) que são menores que x_1, seguidos pelos elementos (também já ordenados) que são maiores ou iguais a x_1. Uma descrição mais precisa do algoritmo é dada a seguir. O algoritmo consiste em chamar `quicksort(1,n)`, onde `quicksort(r,s)` é dado por:

Algoritmo 1.3: quicksort(r,s)

```
se s ≤ r então retorne                (caso básico)
v = x_r
i = r
j = s+1
repita                                (separação)
      repita i = i+1 até x_i ≥ v
      repita j = j-1 até x_j ≤ v
      troque x_i com x_j
até j ≤ i
troque x_i com x_j
```

```
    troque xi com xs
    quicksort(r,i-1)                (recursão)
    quicksort(i+1,s)
Obs: não há uma etapa explícita de combinação.
```

(O código acima assume sentinelas $x_0 \leq x_i \leq x_{n+1}$.)

O bom desempenho deste algoritmo na prática se deve ao fato de que, caso os elementos a serem ordenados sejam escolhidos aleatoriamente em um certo conjunto, o processo de separação tende a dividir o conjunto dado em dois conjuntos com aproximadamente o mesmo número de elementos. Como o processamento da separação requer $O(n)$ passos, uma análise semelhante à desenvolvida acima permite demonstrar que a complexidade média do algoritmo é $O(n \log n)$ (sob a suposição de que os elementos a serem ordenados são independentemente escolhidos segundo alguma distribuição de probabilidade).

No entanto, a complexidade deste algoritmo no pior caso é $O(n^2)$. Para se constatar este fato, basta observar que, caso o conjunto já esteja ordenado, o algoritmo descrito acima vai executar $n-1$ separações. A primeira exige $n-1$ comparações, a segunda $n-2$, e assim por diante, num total de $O(n^2)$ comparações.

Consideremos agora a variação do problema de ordenação na qual desejamos obter soluções parciais. Mais precisamente, queremos ordenar um conjunto $x_1, x_2, ..., x_n$ de números reais, examinando os elementos x_i um de cada vez, de tal forma que, após examinar x_i, já tenhamos ordenado $x_1, x_2, ..., x_i$. Algoritmos deste tipo de são chamados **incrementais** ou *on-line*. O algoritmo de **ordenação por inserção** descrito abaixo resolve o problema de ordenação incremental em tempo quadrático:

Algoritmo 1.4: Ordenação por Inserção

```
para i = 2..n-1
        v = xi
        j = i
        enquanto xj-1 > v
                xj = xj-1
                j = j - 1
        xj = v
```

(O código acima assume sentinela $x_0 \leq x_i$.)

Este algoritmo descobre a posição correta para x_i no segmento $x_1 ... x_{i-1}$ ao mesmo tempo que o coloca lá. Poderíamos utilizar o fato que este segmento inicial está ordenado para acelerar esta busca, usando pesquisa binária, por exemplo. Entretanto, esta modificação não altera a complexidade pois continuamos levando um tempo linear para colocar x_i na posição correta. Poderíamos, então, tentar usar uma lista encadeada para evitar este movimento. No entanto, não é possível fazer busca binária em listas encadeadas.

Felizmente, existem estruturas de dados capazes de realizar pesquisas e inserções em tempo logarítmico. Estas estruturas de dados são, em geral, árvores binárias modificadas de modo a garantir equilíbrio. A implementação destas **árvores balanceadas** é delicada e foge ao escopo deste texto (para uma discussão completa, veja, por exemplo, [Me, S]).

Usando árvores balanceadas, é possível implementar algoritmos incrementais ótimos para ordenação. Na realidade, árvores balanceadas resolvem o problema de ordenação **dinâmica** (no qual é permitido retirar elementos já inseridos) de maneira ótima pois são capazes de realizar inserções, remoções e pesquisas em tempo O(log n).

A existência de estruturas de dados capazes de serem pesquisadas e atualizadas em tempo logarítmico é fundamental para vários algortimos ótimos em geometria computacional, ainda que, às vezes, seja questionável o uso destas estruturas sofisticadas em problemas práticos, dada a dificuldade de implementação.

1.4 Cotas inferiores

Na seção anterior descrevemos diversos algoritmos para o problema de ordenação. O melhor dentre os algoritmos obtidos (Mergesort) foi capaz de ordenar n números em tempo O(n log n). Uma pergunta cabível neste ponto é se existe algum algoritmo para o problema de ordenação que tenha desempenho assintótico superior ao do Mergesort. Mais exatamente, gostaríamos de saber qual é a menor complexidade assintótica possível entre todos os algoritmos que resolvem o problema de ordenação. Caso seja possível obter uma resposta a esta pergunta, estaremos em condições de falar a respeito da complexidade assintótica intrínseca ao problema de ordenação.

Obter a complexidade de um problema (isto é, obter um algoritmo para ele cuja complexidade seja uma cota inferior entre as complexidades de todos os algoritmos possíveis

para o problema) é, em geral, uma tarefa difícil. Uma vez mais, esta complexidade é expressa de forma assintótica.

Definição 1.2: Dizemos que a complexidade de um problema P tem **cota inferior** dada por $\Omega(f(n))$ se existe uma constante k tal que todo algoritmo que resolve P requer pelo menos $kf(n)$ passos para resolver alguma instância de tamanho n. Dizemos que a complexidade de P é $\theta(f(n))$ se, além disso, existir um algoritmo de complexidade $O(f(n))$ para P (neste caso, dizemos que este algoritmo é **ótimo** para P).

A obtenção de resultados neste sentido exige cuidados maiores ao se especificar a classe de algoritmos considerados. Uma escolha adequada para Geometria Computacional é a classe das árvores de decisões algébricas, consideradas na seção 1.2 (todos os algoritmos de ordenação vistos na seção anterior podem ser simulados usando tal modelo, já que eles utilizam apenas comparações). Para tal família de algoritmos, é possível obter resultados que estabelecem cotas inferiores para problemas de interesse, incluindo o problema de ordenação.

Teorema 1.2: *Sob o modelo de árvores algébricas de decisão, o problema de ordenação tem complexidade $\theta(n \log n)$ (logo, o algoritmo Mergesort, que tem esta complexidade, é ótimo para o problema).*

Prova: A complexidade de uma árvore de decisão é dada pela sua profundidade p, isto é, pela profundidade máxima de uma de suas folhas. No caso do problema de ordenação, cada uma das folhas da árvore de decisão deverá estar associada a uma possível ordenação dos n elementos. Como há $n!$ ordenações possíveis para um conjunto com n elementos, a árvore possui no mínimo $n!$ folhas. Mas o número máximo possível de folhas numa árvore de decisões de profundidade p é obtido quando todas as folhas ocorrem a esta profundidade e cada nó que não é uma folha tem exatamente 2 descendentes. Neste caso, o número de folhas é igual ao número de nós à profundidade p, que é 2^p.

Logo, a profundidade p de uma árvore de decisão que possua pelo menos $n!$ folhas satisfaz:

$$2^p \geq n!,$$

isto é,

$$p \geq \log_2 (n!).$$

Mas a aproximação de Stirling para $n!$ [GKP] fornece

$$n! \approx \sqrt{2\pi n}\,(n/e)^n,$$

ou seja:

$$\log_2(n!) \approx \log_2(\sqrt{2\pi n}) + n\log_2 n - n\log_2 e \geq kn\log n$$

(onde a desigualdade vale para n suficientemente grande para qualquer $k < 1$).

Logo a complexidade p tem cota inferior $\Omega(n \log n)$. Como existe um algoritmo de complexidade $O(n \log n)$, concluímos que a complexidade do problema de ordenação é $\theta(n \log n)$. ■

1.5 Redução

Os resultados obtidos a respeito do problema de ordenação serão úteis, nos próximos capítulos, para estabelecer cotas inferiores para o problema de determinar o fecho convexo de um conjunto de pontos e para o problema de obter uma triangulação de um conjunto de pontos do plano.

Para tal, recorreremos a um mecanismo que consiste em **reduzir** um problema a outro. A definição a seguir estabelece este conceito.

Definição 1.3: Um problema P_1 é **redutível** (em tempo linear) a um problema P_2 se dada uma instância I_1 de tamanho n de P_1 for possível, em tempo $O(n)$, obter uma instância I_2 de P_2 tal que a resposta de P_1 para I_1 possa ser obtida, em tempo $O(n)$, a partir da resposta de P_2 para I_2. Neste caso, escrevemos $P_1 \propto_{O(n)} P_2$, ou simplesmente $P_1 \propto P_2$.

A idéia intuitiva por trás desta definição é que, se P_1 é redutível a P_2, então P_1 é pelo menos tão fácil de resolver quanto P_2. O teorema a seguir estabelece de que maneira propriedades de complexidade computacional são herdadas através do mecanismo de redução.

Teorema 1.3: *Suponhamos que P_1 e P_2 sejam problemas tais que $P_1 \propto P_2$. Então:*

a) Se existe um algoritmo que resolve P_2 em $O(f(n))$ passos, então também existe um algoritmo que resolve P_1 em $O(f(n))$ passos.

b) Se qualquer algoritmo para P_1 requer, no pior caso, $\Omega(f(n))$ passos, então qualquer algoritmo para P_2 também requer, no pior caso, $\Omega(f(n))$ passos .

Prova:

a) Um algoritmo $O(f(n))$ para P_1 consiste simplesmente em concatenar o algoritmo que converte instâncias de P_1 em instâncias de P_2, o algoritmo $O(f(n))$ para P_2 e o algoritmo que converte a resposta de P_2 na resposta correspondente de P_1. Como a complexidade das etapas de conversão é $O(n)$, a complexidade total do algoritmo permanece $O(f(n))$.

b) Se houvesse um algoritmo capaz de resolver P_2 em tempo assintoticamente inferior a $f(n)$, pela parte (a) também haveria um tal algoritmo para P_1, o que contradiz a hipótese de P_1 requerer $\Omega(f(n))$ passos. ■

Nos próximos capítulos, mostraremos que o problema de ordenação pode ser reduzido a diversos problemas importantes de Geometria Computacional; desta forma seremos capazes de obter uma cota inferior $\Omega(n \log n)$ para tais problemas, utilizando os resultados dos Teoremas 1.2 e 1.3. Isto nos permitirá discutir a existência de algoritmos ótimos para eles.

Nem sempre, no entanto, é possível utilizar o problema de ordenação para estabelecer cotas inferiores para problemas de interesse em Geometria Computacional. Uma análise mais sofisticada pode ser necessária. Um resultado de Ben-Or [BO], que utiliza métodos de Geometria Algébrica, é muitas vezes empregado para estabelecer cotas inferiores para a complexidade destes problemas. Estes teoremas e suas aplicações fogem ao escopo deste trabalho (ver [Me]).

Exercícios

1. No texto, a análise de complexidade para Mergesort foi feita para $n = 2^p$. Complete a análise para n qualquer.

2. Faça uma análise de complexidade para o algoritmo de ordenação por inserção, verificando que ele leva tempo quadrático no pior caso (dê um exemplo de um pior caso). Em contraste com ordenação por seleção, este algoritmo pode levar muito menos tempo para alguns conjuntos; em particular, ordenação por inserção é muito rápida para conjuntos quase ordenados. Tente calcular o número exato de comparações feitas no caso geral. [Sugestão: está relacionado com o número de **inversões da permutação original** [S].]

2

Primitivas Geométricas

Neste capítulo introduzimos o elenco de construções elementares que servirá de base aos algoritmos introduzidos nos próximos capítulos. Cada uma destas construções elementares envolve um número fixo das operações elementares sobre números reais introduzidas no capítulo 1. Desta forma, conforme o discutido anteriormente, elas podem ser incorporadas ao conjunto de operações elementares sem alterar a análise de complexidade assintótica dos problemas e algoritmos.

Algoritmos geométricos frequentemente necessitam estabelecer a posição relativa de objetos geométricos. Perguntas do tipo: "o ponto p é interior ao triângulo abc?", "os segmentos ab e cd se interceptam?" são comuns na execução de algoritmos. Como cada figura geométrica é representada através das coordenadas dos pontos que a definem, as respostas a perguntas deste tipo são obtidas utilizando técnicas elementares de Geometria Analítica.

Estas técnicas podem ser implementadas a partir de operações com vetores, em especial usando os conceitos de norma euclidiana, produto escalar e (no caso de $\mathbf{R^2}$ e $\mathbf{R^3}$) produto vetorial. Descrevemos neste capítulo um conjunto básico de operações primitivas.

2.1 Operações com vetores

Consideramos como primitivas operações que retornem o resultado de operações com vetores, incluindo:

somavetorial $(\boldsymbol{x}, \boldsymbol{y}) = \boldsymbol{x} + \boldsymbol{y}$

multescalar $(\lambda, \boldsymbol{x}) = \lambda \boldsymbol{x}$

prodescalar $(\boldsymbol{x}, \boldsymbol{y}) = \boldsymbol{x}.\boldsymbol{y} = x_1 y_1 + x_2 y_2 + \ldots + x_n y_n$

norma $(\boldsymbol{x}) = \| \boldsymbol{x} \| = \sqrt{x_1^2 + x_2^2 + \ldots + x_n^2}$

onde $x = (x_1, x_2, ..., x_n)$ e $y = (y_1, y_2, ..., y_n)$ são vetores do $\mathbf{R}^n$ e λ é um número real.

2.2 Distâncias e ângulos

Com auxílio das primitivas acima, podemos imediatamente escrever outras funções capazes de retornar distâncias e ângulos. Podemos, por exemplo, definir as primitivas:

$$\text{distância } (x, y) = \text{norma } (x - y),$$

que retorna a distância entre pontos x e y do $\mathbf{R}^n$ e

$$\text{ângulo } (x, y) = \arccos\left(\frac{x \cdot y}{\|x\| \|y\|}\right).$$

A primitiva acima é motivada pela identidade

$$x \cdot y = \|x\| \|y\| \cos\theta,$$

onde x e y são vetores do $\mathbf{R}^2$ ou $\mathbf{R}^3$ e θ é o ângulo (no sentido usual da Geometria) formado por eles. Note que o ângulo entre dois vetores toma valores no intervalo $[0, \pi]$.

2.3 Ângulos orientados no plano

O conceito de ângulo caracterizado acima com auxílio de produto escalar utiliza uma função simétrica em x e y, sendo, portanto, incapaz de distinguir a orientação relativa de x e y. No plano, é muitas vezes importante tomar ângulos orientados. O problema a seguir, por exemplo, será utilizado no capítulo 3 para obtenção do fecho convexo de um conjunto de pontos do plano:

ORDENAÇÃO POLAR: *Dados vetores $x_1, x_2, ..., x_n$ do $\mathbf{R}^2$, ordená-los angularmente no sentido anti-horário.*

A primitiva ângulo entre vetores não é particularmente útil para construir algoritmos para este problema. Observe, por exemplo, que, em ambas as configurações da figura 2.1, os valores de ângulo (x, y), ângulo (x, z) e ângulo (y, z) são respectivamente iguais.

Dado um vetor $x = (x_1, x_2) \neq \mathbf{0}$ de $\mathbf{R}^2$, o **ângulo orientado** definido por x é representado por ângulo (x) e é igual ao comprimento do arco correspondente no círculo unitário, orientado no sentido anti-horário e tomado a partir do eixo horizontal. Observe que,

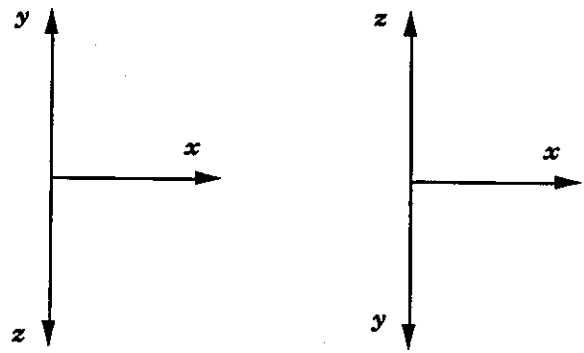


Figura 2.1 – Ângulos entre vetores não determinam orientação relativa.

representando o vetor unitário na direção do semi-eixo horizontal positivo por $\boldsymbol{u} = (1,0)$, temos:

$$\text{ângulo}(\boldsymbol{x}) = \begin{cases} \text{ângulo}(\boldsymbol{u}, \boldsymbol{x}) & \text{se } x_2 \geq 0 \\ -\text{ângulo}(\boldsymbol{u}, \boldsymbol{x}) & \text{se } x_2 < 0 \end{cases}$$

onde

$$\text{ângulo}(\boldsymbol{u}, \boldsymbol{x}) = \arccos\left(\frac{\boldsymbol{u} \cdot \boldsymbol{x}}{\|\boldsymbol{u}\| \|\boldsymbol{x}\|}\right) = \arccos\left(\frac{x_1}{\sqrt{x_1^2 + x_2^2}}\right).$$

Note que a função ângulo definida acima toma valores no intervalo $(-\pi, \pi]$.

De posse da primitiva ângulo orientado, o problema de ordenação polar é facilmente resolvido. Para cada vetor x_i determinamos ângulo (x_i) e ordenamos o conjunto de valores reais resultante utilizando Mergesort. Resulta daí um algoritmo $O(n \log n)$ para ordenação polar.

2.4 Pseudo-ângulos

É importante fazer algumas observações sobre as primitivas *ângulo entre vetores* e *ângulo orientado* definidas acima. Como definidas, ela fazem uso da função transcendente *arc cos* que, embora presente nas bibliotecas de funções matemáticas que acompanham as diversas linguagens de programação, possui duas desvantagens. A primeira é que tal função, não sendo

algébrica, não faz parte do elenco de operações elementares de nosso modelo computacional teórico; a segunda é que, em grande parte dos problemas de interesse em Geometria Computacional, tudo que necessitamos é ser capaz de comparar ângulos. Para tal finalidade, o conceito de ângulo entre vetores pode ser substituído por uma outra medida que seja uma função monótona do ângulo entre eles. Podemos, por exemplo, utilizar a própria função cosseno ou, para manter o sentido das desigualdades,

$$f(\theta) = 1 - \cos\theta \qquad (0 \le \theta \le \pi),$$

que define uma primitiva que podemos chamar de **pseudo-ângulo**, dada por:

$$\text{pseudo-ângulo}\ (x, y) = 1 - \frac{x \cdot y}{\|x\|\,\|y\|}.$$

A função f definida acima é uma função monótona crescente de θ e toma valores no intervalo [0, 2]. O cálculo da nova primitiva pseudo-ângulo envolve apenas operações aritméticas e extração de raiz quadrada[1], o que significa que, além de estar incluída em nosso modelo computacional, apresentará, na prática, um menor tempo de execução. A partir desta primitiva pseudo-ângulo podemos igualmente definir sua versão orientada.

No entanto, preferimos apresentar uma outra alternativa, com um maior apelo geométrico. Como vimos anteriormente, o ângulo orientado correspondente a x é igual ao comprimento do arco orientado correspondente, tomado sobre o círculo unitário centrado na origem. Se substituirmos o círculo unitário por qualquer outra curva contínua que satisfaça a propriedade de que cada semi-reta partindo da origem a corta em um único ponto (isto é, por uma curva que seja o gráfico de uma função em coordenadas polares), a medida do arco tomado sobre esta curva será uma função monótona do arco tomado sobre o círculo unitário. Em consequência, esta nova função pseudo-ângulo poderá ser utilizada para comparar ângulos orientados.

Por exemplo, podemos tomar nosso pseudo-ângulo como medido ao longo do quadrado de vértices (1, 1), (–1, 1) , (–1, –1) e (1, –1) (veja a figura 2.2), o que define uma função definida para todo vetor não-nulo $x \in \mathbf{R}^2$ e tomando valores no intervalo (–4, 4]. Pode-se facilmente verificar (exercício 1) que o cálculo de pseudo-ângulo (x, y) pode ser feito de modo a envolver apenas três comparações, uma soma e uma divisão.

---

[1] Mesmo a extração da raiz quadrada poderia ser evitada, substituindo cos θ por sinal (cos θ) $\cos^2\theta$.

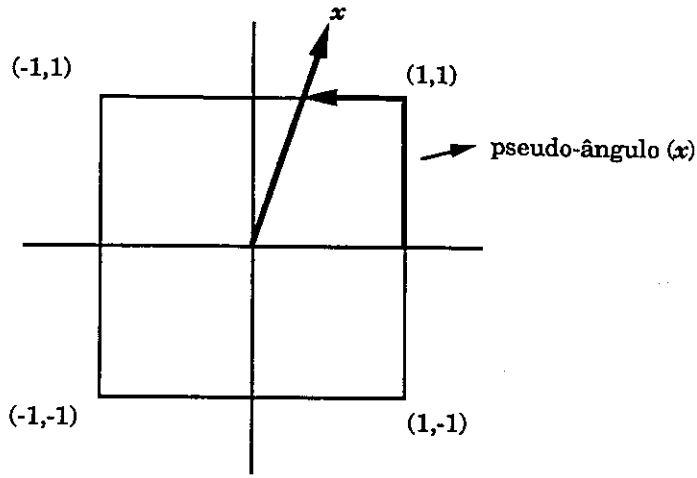


Figura 2.2 – Pseudo-ângulos.

2.5 Produto vetorial

Uma outra forma para determinar orientação relativa de vetores no $\mathbf{R}^2$ e $\mathbf{R}^3$ consiste em empregar produtos vetoriais. O **produto vetorial** é a operação $\times$: $\mathbf{R}^3 \rightarrow \mathbf{R}^3$ definida por:

$$(x_1, x_2, x_3) \times (y_1, y_2, y_3) = (x_2y_3 - x_3y_2, x_3y_1 - x_1y_3, x_1y_2 - x_2y_1).$$

O resultado do produto vetorial de dois vetores não colineares $\boldsymbol{x}$ e $\boldsymbol{y}$ do $\mathbf{R}^3$ é um vetor simultaneamente ortogonal a $\boldsymbol{x}$ e $\boldsymbol{y}$ e orientado de tal modo que a orientação do triedro determinado por $\boldsymbol{x}, \boldsymbol{y}$ e $\boldsymbol{x} \times \boldsymbol{y}$, nesta ordem, é a mesma do triedro definido pelos eixos x, y e z (figura 2.3). Como usualmente os eixos são escolhidos de modo a formar um triedro positivo (isto é, obedecendo à chamada **regra da mão direita**), o mesmo ocorrerá com $\boldsymbol{x}, \boldsymbol{y}$ e $\boldsymbol{x} \times \boldsymbol{y}$.

Uma aplicação desta propriedade do produto vetorial é na determinação da orientação relativa de vetores de $\mathbf{R}^2$. Vetores de $\mathbf{R}^2$ podem ser identificados com os vetores de $\mathbf{R}^3$ cuja terceira componente é nula. O produto vetorial de dois vetores com terceira componente nula tem as duas primeiras componentes nulas. Daí, podemos estender a operação de produto vetorial para o $\mathbf{R}^2$ tomando como resultado o valor escalar da terceira componente. Isto é, definimos $\times$: $\mathbf{R}^2 \rightarrow \mathbf{R}$ por $(x_1, x_2) \times (y_1, y_2) = x_1y_2 - x_2y_1$.

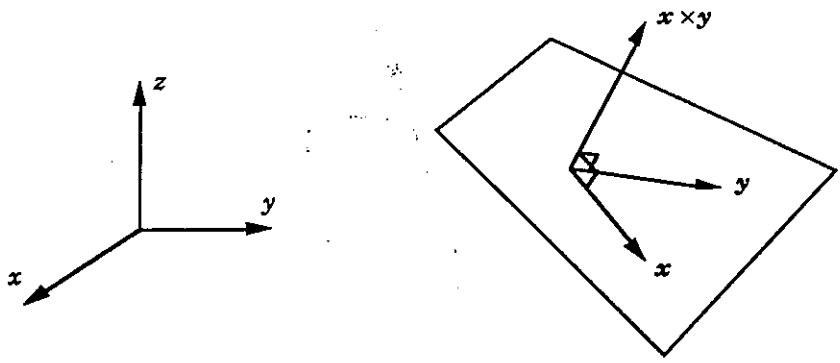
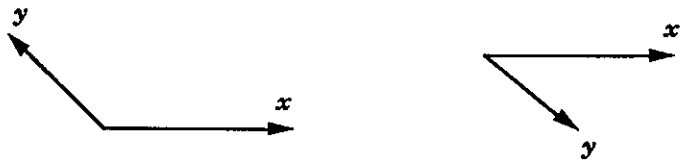


Figura 2.3 – Orientação de $x \times y$.

O sinal positivo ou negativo do produto vetorial $x \times y$ denota se o ângulo orientado de x para y é positivo ou negativo, ou equivalentemente se y está à esquerda ou à direita de x (figura 2.4).



(a) $x \times y > 0$: y à esquerda de x (b) $x \times y > 0$: y à direita de x

Figura 2.4 – Significado do sinal de $x \times y$.

Exemplo: Dados dois segmentos (abertos) ab e cd do plano, determinar se eles se interceptam.

Solução: Os segmentos se interceptam se c e d estão em lados opostos em relação a ab e, ao mesmo tempo, a e b estão de lados opostos em relação a cd (figura 2.5). A primeira condição ocorre se os vetores ac e ad têm orientação oposta com relação ao vetor ab. Isto é, se os produtos vetoriais $(ab \times ac)$ e $(ab \times ad)$ tem sinais opostos. Analogamente, os produtos

vetoriais $(cd \times ca)$ e $(cd \times cb)$ devem ter sinais opostos. Isto é, os segmentos se interceptam se e somente se $(ab \times ac)\ (ab \times ad) < 0$ e $(cd \times ca)\ (cd \times cb) < 0$.

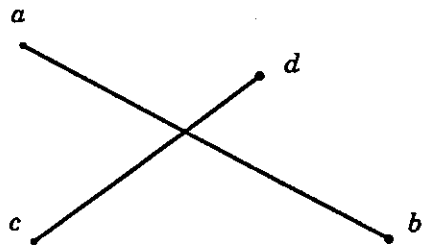


Figura 2.5 – Interseção de segmentos.

2.6 Áreas orientadas de polígonos planos

Se, por um lado, o sinal do produto vetorial de dois vetores permite determinar sua orientação relativa, o valor absoluto do produto vetorial, por outro lado, está relacionado com a área do paralelogramo definido por eles.

Esta relação entre área e produto vetorial provém do fato que a norma do produto vetorial de dois vetores $\boldsymbol{x}$ e $\boldsymbol{y}$ do $\mathbf{R}^3$ satisfaz:

$$\| \boldsymbol{x} \times \boldsymbol{y} \| = \| \boldsymbol{x} \| \| \boldsymbol{y} \| \operatorname{sen} \theta,$$

onde θ é o ângulo formado por $\boldsymbol{x}$ e $\boldsymbol{y}$.

Em consequência, a norma de $\boldsymbol{x} \times \boldsymbol{y}$ é igual à área do paralelogramo formado por $\boldsymbol{x}$ e $\boldsymbol{y}$ (ou , equivalentemente, ao dobro da área do triângulo definido por eles).

No caso de vetores $\boldsymbol{x}$ e $\boldsymbol{y}$ do $\mathbf{R}^2$, o valor do produto vetorial $\boldsymbol{x} \times \boldsymbol{y}$ pode ser visto como a área orientada do paralelogramo definido por eles, que é positiva caso o ângulo orientado de $\boldsymbol{x}$ para $\boldsymbol{y}$ seja positivo e negativa em caso contrário.

Baseados neste fato, podemos obter expressões que calculam áreas de polígonos planos. Começamos com o seguinte resultado:

Teorema 2.1: *Sejam p_1, p_2 e p_3 pontos do $\mathbf{R}^2$ ou $\mathbf{R}^3$. Considere a expressão*

$$S = \frac{1}{2}(op_1 \times op_2 + op_2 \times op_3 + op_3 \times op_1),$$

onde o é um ponto arbitrário. Então:

a) No caso do $\mathbf{R}^3$, S é um vetor normal ao plano de $p_1p_2p_3$, orientado positivamente em relação ao triângulo (isto é, de acordo com a regra da mão direita aplicada ao sentido de rotação de $p_1p_2p_3$) e de norma igual à área de $p_1p_2p_3$.

b) No caso do $\mathbf{R}^2$, S é um escalar igual à área orientada de $p_1p_2p_3$. Isto é, $|S|$ é igual à área de $p_1p_2p_3$ e S é positivo se e somente se p_1, p_2 e p_3, nesta ordem, estão no sentido anti-horário.

Prova: Qualquer que seja o ponto o, temos:

$$\begin{aligned} p_1p_2 \times p_1p_3 &= (op_2 - op_1) \times (op_3 - op_1) \\ &= op_2 \times op_3 - op_2 \times op_1 - op_1 \times op_3 + op_1 \times op_1 \\ &= op_1 \times op_2 + op_2 \times op_3 + op_3 \times op_1. \end{aligned}$$

Logo, $S = \frac{1}{2}(p_1p_2 \times p_1p_3)$, o que imediatamente verifica as afirmações em (a) e (b). Observe que o sentido de rotação de p_1p_2 para p_1p_3, para efeito da regra da mão direita, coincide com o sentido de rotação do triângulo $p_1p_2p_3$. ■

Examinemos com mais detalhe a expressão dada no caso do $\mathbf{R}^2$. Se tomarmos $o = (0,0)$, obtemos a seguinte expressão para a área S do triângulo $p_1p_2p_3$:

$$S = p_1 \times p_2 + p_2 \times p_3 + p_3 \times p_1.$$

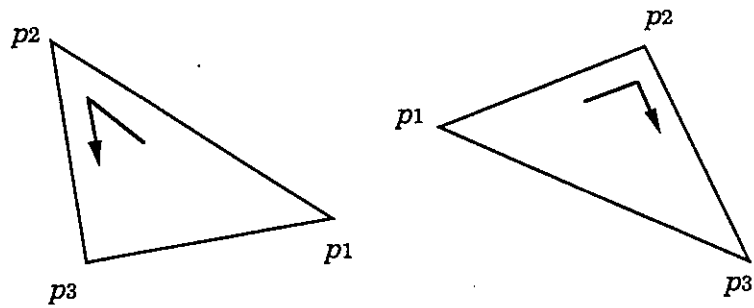
Tomando $p_1 = (x_1, y_1)$, $p_2 = (x_2, y_2)$, $p_3 = (x_3, y_3)$, a expressão acima é equivalente a

$$S = x_1y_2 - x_2y_1 + x_2y_3 - x_3y_2 + x_3y_1 - x_1y_3,$$

que pode ser descrita mnemonicamente pelo quadro:

$$S = \frac{1}{2}\begin{vmatrix} x_1 x_2 x_3 x_1 \\ y_1 y_2 y_3 y_1 \end{vmatrix}.$$

Frequentemente, empregamos a expressão acima com o único intuito de obter a orientação do triângulo $p_1p_2p_3$. Em muitas aplicações é conveniente definir uma primitiva anti-horário (p_1, p_2, p_3), que indica se o triângulo $p_1p_2p_3$ é orientado positivamente (figura 2.6).



(a) sentido anti-horário (positivo) (b) sentido horário (negativo)

Figura 2.6 – Orientações de um triângulo plano $p_1p_2p_3$.

A fórmula de área de triângulo vista acima pode ser generalizada para polígonos simples quaisquer, dados através de sua lista de vértices.

Teorema 2.2: *Seja $p_1p_2 \ldots p_n$ $(n \geq 3)$ um polígono plano simples do $\mathbf{R}^2$ ou $\mathbf{R}^3$. Considere a expressão*

$$S = \frac{1}{2}(op_1 \times op_2 + op_2 \times op_3 + \ldots + op_n \times op_1),$$

onde o é um ponto arbitrário. Então:

a) No caso do $\mathbf{R}^3$, S é um vetor normal ao plano de $p_1p_2 \ldots p_n$, orientado positivamente em relação ao polígono (isto é, de acordo com a regra da mão direita aplicada ao seu sentido de rotação) e de norma igual à sua área.

b) No caso do $\mathbf{R}^2$, S *é um escalar igual à área orientada de* $p_1p_2...p_n$. *Isto é,* $|S|$ *é igual à área do polígono e* S *é positivo se e somente se* $p_1, p_2, ..., p_n$, *nesta ordem, estão no sentido anti-horário.*

Prova: O resultado é demonstrado por indução. A demonstração utiliza o seguinte lema, a ser demonstrado no capítulo 4:

Lema: *Todo polígono simples possui uma diagonal completamente contida no seu interior.*

O resultado certamente é válido para $n = 3$, pelo Teorema 2.1. Suponhamos, por indução, que o resultado é válido para polígonos com menos de n vértices e consideremos o polígono de n vértices $p_1p_2...p_n$. Pelo Lema, $p_1p_2...p_n$ admite uma diagonal completamente contida no seu interior. Sem perda de generalidade, podemos supor que tal diagonal é p_1p_i. Então a diagonal p_1p_i divide o polígono em dois polígonos com menos de n vértices: $p_1p_2...p_i$ e $p_1p_i...p_n$ para os quais, pela hipótese de indução, o resultado é válido (veja a figura 2.7).

Isto é:

$$S' = \frac{1}{2}(op_1 \times op_2 + op_2 \times op_3 + ... + op_i \times op_1) \quad \text{e}$$

$$S'' = \frac{1}{2}(op_1 \times op_i + op_i \times op_{i+1} + ... + op_n \times op_1)$$

são vetores de mesma direção e sentido (já que $p_1p_2...p_i$ e $p_1p_i...p_n$ são coplanares e possuem a mesma orientação) cujas normas representam, respectivamente, as áreas de $p_1p_2...p_i$ e $p_1p_i...p_n$.

Portanto, $S' + S''$ é um vetor, ainda com a mesma direção e sentido, cuja norma é igual à área de $p_1p_2...p_n$. Mas:

$$S' + S'' = \frac{1}{2}(op_1 \times op_2 + op_2 \times op_3 + ... + op_n \times op_1) = S,$$

o que verifica o resultado. ■

Suponhamos agora que o polígono $p_1p_2...p_n$ esteja no $\mathbf{R}^2$, com $p_i = (x_i, y_i)$. Tomando $o = (0,0)$, a área orientada de $p_1p_2...p_n$ é dada por:

$$S = p_1 \times p_2 + p_2 \times p_3 + ... + p_{n-1} \times p_n + p_n \times p_1$$

$$= x_1y_2 - x_2y_1 + x_2y_3 - x_3y_2 + ... + x_{n-1}y_n - x_ny_{n-1} + x_ny_1 - x_1y_n,$$

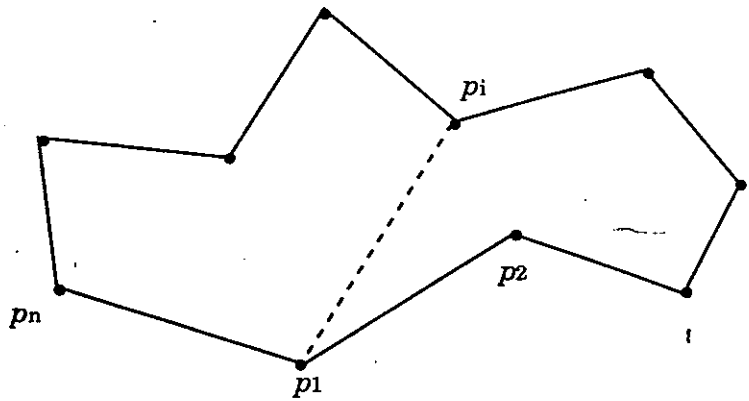


Figura 2.7 – Decompondo o polígono simples por meio de uma diagonal.

que pode, novamente, ser representada pelo quadro mnemônico

$$S = \frac{1}{2}\begin{vmatrix} x_1 x_2 \ldots x_n x_1 \\ y_1 y_2 \ldots y_n y_1 \end{vmatrix}.$$

Observamos que este resultado pode ser generalizado para o cálculo de volumes de poliedros (veja o exercício 2).

2.7 Coordenadas baricêntricas

Um problema importante em Geometria Computacional é determinar se um dado ponto pertence ao interior de um dado polígono simples. Na próxima seção tratamos este problema para um polígono qualquer. Aqui, examinamos uma solução específica para o caso em que o polígono é um triângulo, recorrendo às chamadas coordenadas baricêntricas.

Teorema 2.3: *Sejam p_1, p_2 e p_3 pontos não colineares de $\mathbf{R}^2$. Então cada ponto p do plano pode ser escrito de modo único na forma*

$$p = \lambda_1 p_1 + \lambda_2 p_2 + \lambda_3 p_3,$$

onde λ_1, λ_2 e λ_3 são números reais satisfazendo $\lambda_1 + \lambda_2 + \lambda_3 = 1$. Os coeficientes λ_1, λ_2 e λ_3 são denominados as **coordenadas baricêntricas** *de p em relação a p_1, p_2 e p_3.*

(Observe que, se massas iguais a λ_1, λ_2 e λ_3 são colocadas em p_1, p_2 e p_3, então o baricentro dessa configuração é o ponto p).

Prova: Basta observar que, dados p, p_1, p_2 e p_3 (sendo $p = (x, y)$ e $p_i = (x_i, y_i)$, $i = 1,2,3$), os ternos $(\lambda_1, \lambda_2, \lambda_3)$ satisfazendo as condições dadas são soluções do seguinte sistema de três equações a três incógnitas:

$$\begin{aligned} \lambda_1 x_1 &+ \lambda_2 x_2 + \lambda_3 x_3 = x \\ \lambda_1 y_1 &+ \lambda_2 y_2 + \lambda_3 y_3 = y \\ \lambda_1 &+ \lambda_2 + \lambda_3 = 1 \end{aligned}$$

O determinante do sistema é:

$$D = \begin{vmatrix} x_1 x_2 x_3 \\ y_1 y_2 y_3 \\ 1\ 1\ 1 \end{vmatrix} = \begin{vmatrix} x_1 x_2 x_3 x_1 \\ y_1 y_2 y_3 y_1 \end{vmatrix},$$

que representa o dobro da área do triângulo $p_1p_2p_3$ e é, portanto, não-nulo. Logo o sistema dado tem solução única para cada p. ■

Os valores de λ_1, λ_2 e λ_3 podem ser facilmente obtidos no sistema anterior utilizando a regra de Cramer. Temos, por exemplo,

$$\lambda_1 = \frac{\begin{vmatrix} x x_2 x_3 \\ y y_2 y_3 \\ 1\ 1\ 1 \end{vmatrix}}{\begin{vmatrix} x_1 x_2 x_3 \\ y_1 y_2 y_3 \\ 1\ 1\ 1 \end{vmatrix}} = \frac{S_{pp_2p_3}}{S_{p_1p_2p_3}},$$

onde $S_{pp_2p_3}$ e $S_{p_1p_2p_3}$ são as áreas orientadas dos triângulos pp_2p_3 e $p_1p_2p_3$, respectivamente. Analogamente, temos

$$\lambda_2 = \frac{S_{p_1pp_3}}{S_{p_1p_2p_3}} \text{ e } \lambda_3 = \frac{S_{p_1p_2p}}{S_{p_1p_2p_3}}.$$

As expressões acima permitem associar o sinal das coordenadas baricêntricas com as regiões do plano determinadas pelas retas que contêm os lados do triângulo. Por exemplo, $\lambda_1 > 0$ se e só se o triângulo pp_2p_3 tem a mesma orientação de $p_1p_2p_3$, o que ocorre quando p está no mesmo semi-plano de p_1 em relação à reta que contém p_2p_3. A figura 2.8 mostra as

diversas regiões do plano e os sinais correspondentes das coordenadas baricêntricas. Observe que a localização de um ponto em relação ao triângulo é imediata a partir de suas coordenadas baricêntricas. Como tais coordenadas são facilmente obtidas a partir das primitivas que calculam áreas orientadas, seu uso fornece uma forma compacta de efetuar tal localização (veja, por exemplo, o exercício 4).

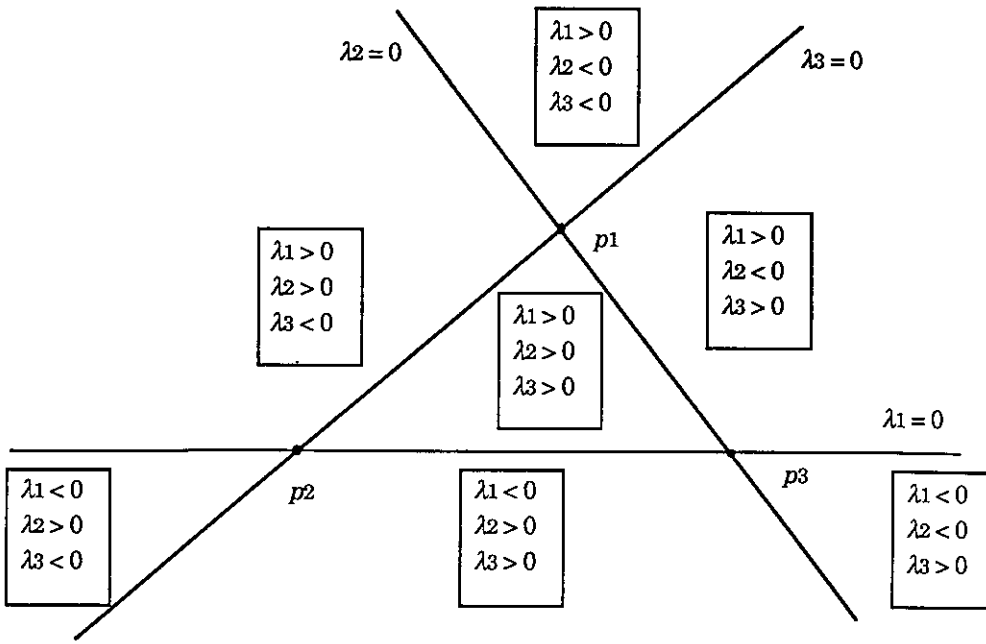


Figura 2.8 – Sinais das coordenadas baricêntricas.

Terminamos esta seção observando que as coordenadas baricêntricas de um ponto podem ser interpretadas como imagens da transformação afim T: $\mathbf{R}^2 \rightarrow \mathbf{R}^3$ tal que $T(p_1) = (1, 0, 0)$, $T(p_2) = (0, 1, 0)$ e $T(p_3) = (0, 0, 1)$. Tal transformação é injetiva e leva o $\mathbf{R}^2$ no plano de equação $x + y + z = 1$.

2.8 Localização de pontos em relação a polígonos

Consideremos novamente o problema PONTO EM POLÍGONO, em que desejamos determinar se um ponto p é interior, exterior ou está na fronteira do polígono simples $P = p_1p_2...p_n$.

O problema faz sentido, já que uma linha poligonal fechada e simples é uma **curva de Jordan** e, como tal, separa o plano em duas regiões conexas e abertas, uma limitada e a outra não, ambas tendo a curva poligonal como fronteira (ver, por exemplo, [He]).

Uma solução para o problema consiste em considerar uma semi-reta L partindo de p e determinar seus pontos de interseção com a linha poligonal. Se p coincidir com um destes pontos de interseção, concluímos que ele está na fronteira de P. Senão, basta contar quantas vezes a semi-reta atravessa a poligonal. No infinito, L se encontra na região exterior. Logo, se o número de cruzamentos for ímpar, o ponto p é interior; caso contrário, p é exterior.

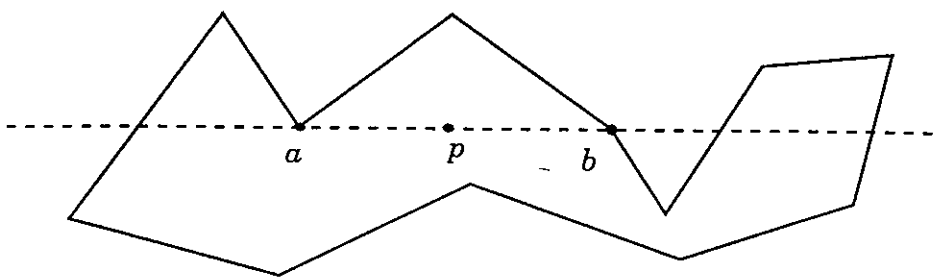


Figura 2.9 – Problema PONTO EM POLÍGONO.

Para obter um algoritmo baseado na descrição acima, é necessário ter cuidado com alguns detalhes. Já que L é arbitrária, optamos por tomar a semi-reta horizontal

$$L = \{(x_0, y_0) + t\,(1, 0) \mid t \geq 0\},$$

para simplificar o cálculo dos pontos de interseção. Os pontos de interseção de L com a linha poligonal são obtidos calculando o ponto de interseção de L com cada lado do polígono. Surgem dificuldades quando L contém vértices do polígono. Quando um ponto de interseção é um vértice, não é necessariamente verdade que L passe do interior para o exterior (ou vice-versa) naquele ponto. No caso da figura 2.9, por exemplo, o ponto a não deve ser contado

como ponto de interseção, enquanto b deve. Problemas surgem também quando L contém um lado do polígono.

Uma forma prática de solucionar os conflitos acima é a seguinte: a interseção de um lado p_ip_{i+1} com L é contada somente se ela ocorrer em um ponto que não seja de ordenada mínima em p_ip_{i+1}. A figura 2.10 mostra cada caso possível e o número de interseções correspondente. Note que os diversos casos de cruzamento são corretamente considerados.

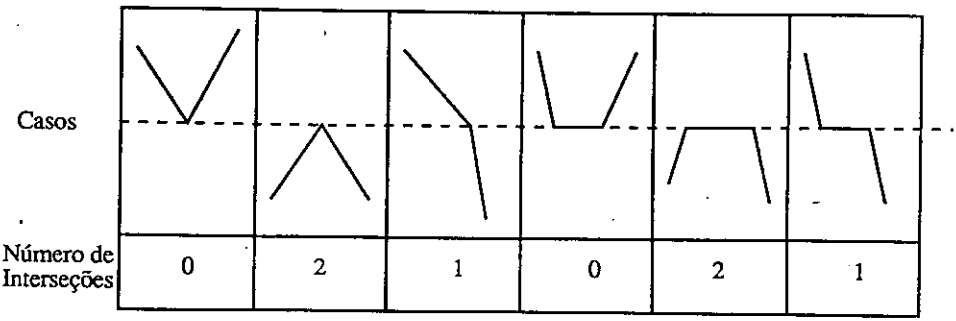


Figura 2.10 – Casos de cruzamento e número de interseções.

Temos então o seguinte algoritmo:

Algoritmo 2.1: Ponto-em-polígono (P, p)

```
dados: p = (x0, y0); pi = (xi, yi), i = 1,..., n; pn+1 = p1.
inicialização: N = 0
para i = 1..n                          {teste cada lado pipi+1 de P}
      se yi ≠ yi+1 então               {lado não é horizontal}
            (x, y) = a interseção de pipi+1 com L
            se x = x0 então
                  p0 é da fronteira: PARE.
            senão, se x > x0 e y > min{xi, xi+1} então {não tem y mínimo}
                  N = N+1.
      senão, se p pertence ao lado horizontal pipi+1 então
                  p é da fronteira: PARE.
se N é ímpar, então p é interior a P; senão, p é exterior.
```

Uma outra alternativa para resolver PONTO EM POLÍGONO consiste em recorrer à noção de índice de rotação. Dada uma linha poligonal fechada $L = p_1p_2...p_n$ (não necessariamente simples) e um ponto p não pertencente a ela, definimos o **índice de rotação** de p em relação a L como

$$\kappa = \frac{1}{2\pi} \sum_{i=1}^{n} \angle(p_i p p_{i+1})$$

onde $\angle(p_i p p_{i+1})$ denota o ângulo orientado do vetor pp_i para o vetor pp_{i+1} e onde tomamos $p_{n+1} = p_1$.

Como cada ângulo orientado acima é igual ao comprimento do arco orientado obtido projetando dois vértices consecutivos sobre um círculo de raio 1 centrado em p, a soma de todos os ângulos corresponde a um número inteiro de voltas no círculo. Logo, κ certamente é um número inteiro.

No caso específico de L determinar um polígono simples (isto é, de não possuir auto-interseções) vale o seguinte:

Teorema 2.4: *Seja $P = p_1p_2...p_n$ um polígono simples e seja p um ponto que não pertence à fronteira de P. Então o índice de rotação κ de p é igual a 0, 1 ou -1 e*

a) p é interior se e somente se $\kappa = \pm 1$.

b) p é exterior se e somente se $\kappa = 0$.

Prova: Suponhamos que $p_1p_2...p_n$ seja orientado no sentido positivo e que o ponto p seja interior a $p_1p_2...p_n$. Para calcular a soma de todos os ângulos orientados determinados por lados do polígono, consideramos todos as semi-retas com origem em p e passando por vértices do polígono. Isto divide o círculo em um certo número (n, a menos de casos degenerados) de setores com interiores disjuntos, cuja união cobre o círculo (figura 2.11).

A contribuição de cada um destes setores para a soma dos ângulos orientados corresponde à diferença entre o número de vezes que um lado do polígono atravessa o setor no sentido positivo e o número de vezes em que ele é atravessado no sentido negativo. Tomemos qualquer setor e uma semi-reta L interna a ele, com origem em p. Como p é interior ao triângulo, o número de interseções de lados do polígono com L é ímpar. Cada vez que L "sai" do polígono,

cruza um lado orientado no sentido positivo e cada vez que "entra", cruza um lado orientado no sentido negativo. Como o número de saídas é exatamente um a mais que o número de entradas, a contribuição do setor é igual a seu comprimento. Logo a soma de todos os ângulos orientados $\angle(p_i p p_{i+1})$ é igual a 2π e o índice de rotação de p é 1.

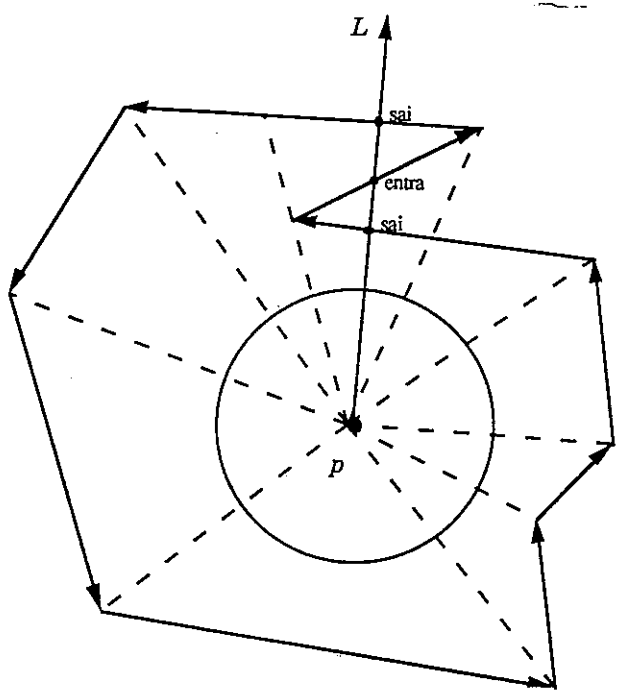


Figura 2.11 – Calculando o índice de rotação.

Caso o ponto seja interior, a semi-reta L descrita acima será cortada um número par de vezes, metade correspondente a entradas e metade a saídas, o que faz com que a contribuição de cada setor seja zero. Isto é, o índice de rotação é igual a 0 para pontos interiores ao polígono. ■

É bom frisar que a análise acima continua valendo se empregamos pseudo-ângulos no lugar de ângulos.

Os dois algoritmos descritos nesta seção resolvem o problema de determinar a posição de um ponto em relação a um polígono simples de n lados em tempo $O(n)$. De um modo geral, o primeiro algoritmo é mais robusto. A descontinuidade do índice de rotação na fronteira do polígono torna o segundo método altamente instável para pontos situados próximos à fronteira (especialmente se eles estão próximos de vértices). Por esta razão, o segundo método só pode ser aplicado com segurança se a proximidade da fronteira for controlada antes de aplicar o algoritmo.

Exercícios

1. Implemente a primitiva pseudo-ângulo introduzida na seção 2.4. Tente utilizar o menor número possível de operações aritméticas e comparações.

2. Escreva um algoritmo `entre(u, v, w)`, que dados vetores **u**, **v** e **w** do $\mathbf{R}^2$, retorna ERRO se **u** e **v** são colineares, SIM se **w** está no ângulo convexo determinado por **u** e **v** e NÃO. caso contrário.

3. a) Mostre que o volume de um tetraedro $p_1p_2p_3p_4$ do espaço tridimensional é dado por

$$V = \frac{1}{6}(op_1 \times op_2 + op_2 \times op_3 + op_3 \times op_4 + op_4 \times op_1),$$

onde o é um ponto arbitrário do $\mathbf{R}^3$.

b) Seja P um poliedro do espaço tridimensional, com faces $F_1, F_2, ..., F_m$. Mostre que o volume de P é dado por:

$$V = \sum_{i=1}^{m} V(F_i),$$

onde $V(F_i)$ é expresso por

$$V(F_i) = \frac{1}{6}(op_1 \times op_2 + op_2 \times op_3 + ... + op_{n-1} \times op_n + op_n \times op_1),$$

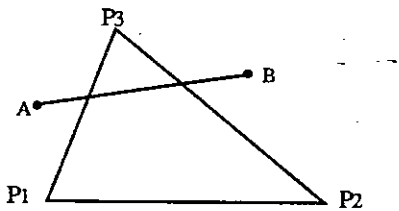
onde o é um ponto arbitrário e $p_1, p_2, ..., p_n$ são os vértices de F_i, tomados na ordem anti-horária quando observados de fora do poliedro.

[Sugestão: Triangule o poliedro; isto é, decomponha-o em tetraedros.]

4. Escreva um algoritmo que utiliza coordenadas baricêntricas para localizar um ponto em relação a um triângulo.

5. O problema de ***clipping*** (ou recorte, ou cerceamento) é extremamente relevante em Computação Gráfica. Dada uma curva e um polígono, deseja-se saber que porções desta curva são interiores ao polígono. Neste problema, consideraremos o caso particular em que a curva é um segmento de reta e o polígono é um triângulo.

Consideremos um triângulo de vértices P_1, P_2 e P_3 e um segmento de extremos A e B. Sejam $(\alpha_1, \alpha_2, \alpha_3)$ e $(\beta_1, \beta_2, \beta_3)$ as coordenadas baricêntricas de A e B em relação a P_1, P_2 e P_3.



a) Que condição deve ser satisfeita para que o segmento AB esteja completamente contido no triângulo?

b) Encontre condições sob as quais podemos garantir que o segmento AB é completamente exterior ao triângulo.

c) Caso os testes em (a) e (b) falhem, necessitaremos obter as interseções de AB com as retas que contêm os lados do triângulo. Diga como determinar cada um destes pontos de interseção, em função das coordenadas baricêntricas de A e B.

6. a) Escreva um algoritmo que determina se dois triângulos do plano são disjuntos.

b) Faça o mesmo para triângulos no espaço.

7. Sejam dados pontos distintos $P_1, P_2, ..., P_n$ do plano. Dê um algoritmo para verificar se o polígono formado por estes pontos, *na ordem indicada*, é convexo (seu algoritmo deve ser linear em n).

8. Suponha que, ao aplicar o algoritmo que determina a posição de um ponto p em relação a um polígono simples $P = p_1p_2...p_n$ baseado no número de interseções de uma semi-reta com os lados de P, armazenamos apenas o primeiro ponto de interseção u (isto é, o mais próximo de p_0) e o lado p_ip_{i+1} em que ele ocorre. Mostre que, sabendo que P é orientado no sentido-horário, esta informação é suficiente para decidir se p é interno, externo ou está na fronteira de P.

9. Extenda os algoritmos de localização de ponto em relação a polígono para determinar se um ponto do $\mathbf{R}^3$ é interior, exterior ou está na fronteira de um poliedro, dado através de sua lista de faces.

3

Fecho Convexo

Neste capítulo examinamos o problema de determinar o fecho convexo de um conjunto finito de pontos, isto é, o menor conjunto convexo que contém tais pontos. A determinação do fecho convexo, além de ser um importante problema para as aplicações, permite examinar algumas das técnicas algorítmicas fundamentais em Geometria Computacional.

3.1 Preliminares

Inicialmente, estabelecemos algumas definições a respeito de convexidade. O leitor pode consultar, por exemplo, [Ro] para uma visão mais completa do assunto.

Um conjunto K do $\mathbf{R}^n$ é **convexo** se quaisquer que sejam $x \in K, y \in K$ e $0 \leq \lambda \leq 1$, tem-se $\lambda x + (1-\lambda)y \in K$ (isto é, se **combinações convexas** de elementos de K pertencem a K). Um ponto $w \in K$ é um **ponto extremo** (ou um **vértice**) de K se não pode ser expresso como combinação convexa de elementos de K distintos de w.

Uma família particular de conjuntos convexos é constituída pelos poliedros convexos. Um **poliedro convexo** é a interseção de um número finito de **semi-espaços** (um semi-espaço é um conjunto da forma $\{(x_1, x_2, ..., x_n) \mid a_1x_1 + a_2x_2 + ... a_nx_n \leq b\}$). Um poliedro limitado é chamado de **politopo**. Polígonos convexos planos e poliedros do espaço tridimensional são casos particulares de poliedros convexos.

O **fecho convexo** de um conjunto finito $C = \{p_1, p_2, ..., p_m\}$ de pontos do $\mathbf{R}^n$ é

$$\text{conv}(C) = \{\lambda_1 p_1 + \lambda_2 p_2 + ... + \lambda_m p_m \mid \lambda_i \geq 0 \quad \text{e} \quad \sum_{i=1}^{m} \lambda_i = 1\}.$$

(isto é, conv(C) é o conjunto de todas as combinações convexas de elementos de C).

É fácil verificar que o fecho convexo de um conjunto finito C fica inteiramente caracterizado pelos seus pontos extremos, que são necessariamente elementos de C. Isto é, se $C' = \{p \in C \mid p \text{ é extremo}\}$, então conv($C'$) = conv($C$) (exercício 2).

O resultado fundamental a seguir, cuja demonstração pode ser encontrada em [Ro], estabelece a equivalência entre fechos convexos de conjuntos finitos e politopos.

Teorema 3.1: *O fecho convexo de um conjunto finito é um politopo. Reciprocamente, todo politopo é o fecho convexo do conjunto (finito) de seus pontos extremos.*

Embora politopos fiquem completamente caracterizados através de seus vértices ou das equações dos hiperplanos que limitam os semi-espaços, consideramos que descrever um poliedro consiste em descrever a sua estrutura facial, que completamente caracteriza a fronteira. Uma **face** (ou um **conjunto extremo**) de um conjunto convexo C é um subconjunto convexo F de C cujos elementos podem ser descritos como combinações convexas de elementos de C somente se estes elementos pertencem a F. Pode-se demonstrar que cada face de C é dada pela interseção de C com um semi-espaço que o contenha.

Note que cada ponto extremo de um conjunto convexo determina uma face de dimensão 0. (A **dimensão** de um subconjunto S de $\mathbf{R}^2$ é a dimensão do espaço vetorial gerado pelos vetores da forma $x - y$, onde x e y são elementos de S.)

Dado um poliedro de dimensão d, suas faces de dimensão 0 são, como vimos, chamadas de **vértices**; as faces de dimensão 1 são as **arestas** e as faces de dimensão $d-1$ são chamadas de **facetas**. Descrever a **estrutura facial** de um poliedro consiste em descrever suas faces de todas as dimensões e as relações de incidência entre elas.

Neste trabalho, consideramos apenas poliedros de dimensão 2 e 3. Poliedros de dimensão 2 são polígonos convexos, possuem apenas faces de dimensão 0 (vértices) e 1 (arestas ou **lados**), e sua estrutura facial é descrita por uma lista circular $v_1a_1v_2a_2v_3a_3v_4a_4...v_na_nv_1$, que caracteriza a incidência de seus vértices e arestas. Em consequência, a descrição de um polígono convexo de n lados requer uma estrutura de dados que ocupa espaço $O(n)$.

Poliedros de dimensão 3 possuem faces de dimensão 0 (vértices), 1 (arestas) e 2 (facetas, ou simplesmente **faces**). A descrição das faces e sua incidência é um pouco mais complexa do que no caso poligonal. Na seção 3.4, discutimos formas eficientes de armazenar tal descrição.

3.2 Fecho convexo bidimensional: complexidade

Nesta seção e na seguinte consideramos o seguinte problema

FECHO CONVEXO BIDIMENSIONAL (FC2D): *Dado um conjunto* $C = \{p_1, p_2, ..., p_n\}$ *de pontos do plano, obter o fecho convexo de* C.

Resolver este problema consiste em determinar quais dos pontos de C são vértices de conv(C) e ordenar estes pontos circularmente, de acordo com sua ocorrência na fronteira de conv(C).

O parágrafo anterior sugere um algoritmo óbvio (mas extremamente ineficiente) para resolver FC2D. Observe que um ponto p_i é vértice de conv(C) se e somente se não pertence a nenhum dos $\binom{n-1}{3}$ triângulos com vértices escolhidos em $C \setminus \{p_i\}$. A determinação dos vértices de conv(C) pode ser feita, portanto, em tempo proporcional a:

$$n \binom{n-1}{3} = O(n^4).$$

Uma vez obtidos os vértices (sem perda de generalidade, suponha que eles são os m primeiros pontos), basta ordená-los no sentido anti-horário. Para tal, basta tomar um ponto interior a conv(C) (por exemplo, o baricentro $o = \frac{1}{n}\sum_{i=1}^{n} p_i$) e ordenar ciclicamente os vetores $op_1, ..., op_n$ em torno deste ponto. Como vimos no capítulo 1, este segundo passo requer apenas tempo $O(n \log n)$.

Como veremos a seguir, o segundo passo do algoritmo grosseiro descrito acima captura toda a complexidade computacional do problema. Isto é, veremos que a complexidade do problema FC2D é dada por $\theta(n \log n)$. Não é surpreendente que $n \log n$ seja uma cota inferior para a complexidade de FC2D. Afinal de contas, parece intuitivamente óbvio que a resolução de FC2D requeira ordenar n vetores 2D, o que é pelo menos tão complexo quanto ordenar n números reais.

De fato, podemos utilizar qualquer algoritmo que resolva FC2D para, com tempo adicional de processamento $O(n)$, ordenar um conjunto de n elementos: dados números reais $x_1, x_2, ..., x_n$, formamos o conjunto $C = \{p_1, p_2, ..., p_n\}$, onde $p_i = (x_i, x_i^2)$ e aplicamos o algoritmo que encontra o fecho convexo dos pontos p_i (veja a figura 3.1). Cada um dos p_i é um vértice de conv(C) e o algoritmo para FC2D ordena os p_i circularmente de acordo com suas abscissas x_i. Basta então, em tempo linear, obter o ponto de menor abscissa e ler, em ordem, as abscissas dos vértices.

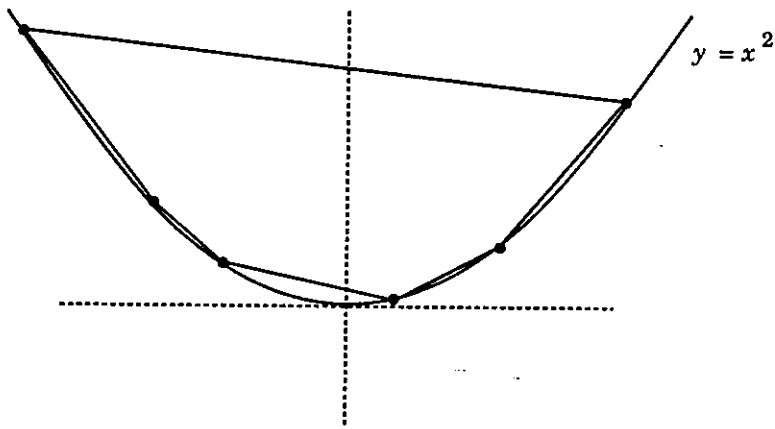


Figura 3.1 – Reduzindo ORDENAÇÃO a FC2D.

O argumento acima mostra que

ORDENAÇÃO $\propto$ FC2D,

o que demonstra que FC2D é pelo menos tão complexo como ORDENAÇÃO, que necessita tempo $\Omega(n \log n)$ no pior caso. Desta forma, acabamos de demonstrar o seguinte:

Teorema 3.2: *Determinar o fecho convexo de um conjunto de n pontos do plano requer, no pior caso, $\Omega(n \log n)$ passos.* ■

Observe que este resultado implica, trivialmente, que a determinação do fecho convexo em d dimensões requer tempo pelo menos $\Omega(n \log n)$, já que o $\mathbf{R}^2$ pode ser mapeado no

subconjunto de $\mathbf{R}^d$ em que as últimas d–2 componentes são nulas. Como veremos a seguir, este limite anterior pode ser atingido nos casos $d = 2$ e $d = 3$. Para uma discussão a respeito do caso d-dimensional, ver [PS].

3.3 Fecho convexo bidimensional: algoritmos

Algoritmo de Jarvis

A primeira idéia que examinaremos para obter um algoritmo eficiente para FC2D é procurar imitar o processo pelo qual o problema seria resolvido através de construções geométricas tradicionais, utilizando papel, lápis e (neste caso) apenas uma régua.

A idéia mais óbvia é, primeiro, obter um dos vértices, como o ponto extremo em uma das direções (um ponto com estas características é necessariamente extremo, pois não pode ser descrito como combinação convexa de outros pontos do conjunto). Podemos, por exemplo, tomar o ponto de menor ordenada (suponhamos que tal ponto seja p_1). Se houver mais de um ponto com a mesma ordenada mínima, tomamos entre eles o de maior abscissa. O ponto que se segue a p_1 no fecho convexo no sentido anti-horário pode ser obtido da seguinte forma. Tomamos a semi-reta paralela ao semi-eixo horizontal positivo Ox passando por p_1 e giramos tal semi-reta em torno de p_1 no sentido anti-horário até que um ponto p_2 seja atingido (no caso de mais de um ponto se encontrar sobre a semi-reta tomamos o mais distante de p_1). Note que, necessariamente p_1p_2 é uma aresta de conv(C) (é um conjunto extremo de dimensão 1) e portanto, p_2 é o próximo vértice do fecho convexo. Agora, a partir de p_2, giramos a semi-reta obtida prolongando p_1p_2, também no sentido anti-horário, até determinar p_3, e assim por diante, até que retornemos ao ponto p_1 (veja a figura 3.2).

Este algoritmo é conhecido como **algoritmo de Jarvis** [Ja] e é descrito mais formalmente a seguir. Nesta descrição, utilizamos a função `próximo(p0,v)`, que retorna o ponto p de C tal que ângulo(v, p_0p) seja mínimo (isto é, o ponto obtido girando uma semi-reta em torno de p_0, a partir da direção dada por v). Observe que, uma vez determinado o ponto inicial p_1 do fecho convexo como o ponto de ordenada mínima, o seu sucessor no fecho convexo pode ser obtido chamando `próximo(p0,(1,0))`.

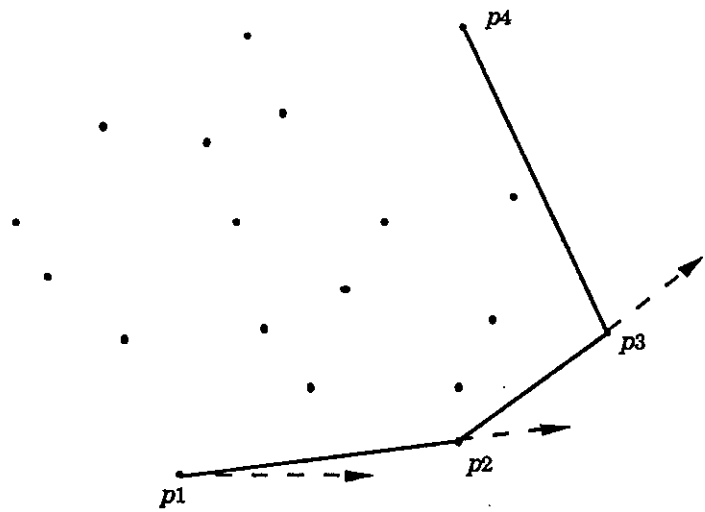


Figura 3.2 – Algoritmo de Jarvis.

Algoritmo 3.1: (Algoritmo de Jarvis)

```
p1 = ponto de ordenada mínima
p2 = próximo (p1, (1,0))
i = 2
repita
        pi+1 = próximo (pi, pi-1pi)
        i = i+1
até que pi = p1.
```

O algoritmo acima determina corretamente o fecho convexo de um conjunto C de n pontos do plano, mas não é ótimo. A determinação do vértice seguinte requer tempo $O(n)$. Como é possível que cada ponto de C seja vértice, esta determinação ocorre também $O(n)$ vezes, daí resultando um algoritmo $O(n^2)$, que reflete uma complexidade superior à cota inferior de $O(n \log n)$.

Na prática, no entanto, o algoritmo de Jarvis pode ter um bom desempenho. Sua complexidade pode ser descrita como sendo $O(vn)$, onde v é o número de vértices de conv(C), que pode ser bem menor que n. Por exemplo, pode-se demonstrar que, se os n pontos são

escolhidos aleatoriamente, e são independente e uniformemente distribuídos sobre o quadrado unitário, então o valor esperado de v é O(log n). Neste caso, portanto, o algoritmo de Jarvis funciona, na média, em tempo O(n log n). Veja o capítulo 4 de [PS] para uma discussão mais abrangente do assunto.

Algoritmo de Graham

Nesta subseção examinamos um algoritmo que procura explorar, de modo explícito, a relação entre ORDENAÇÃO e FC2D. Na seção anterior, vimos um algoritmo ineficiente que fazia isto e que operava ordenando os vértices depois de encontrá-los. A idéia chave no algoritmo de Graham é a seguinte: a determinação dos pontos extremos pode ser feita de modo eficiente se os pontos $p_1, p_2, ..., p_n$ tiverem sido previamente ordenados polarmente em torno de um ponto interior ao seu fecho convexo.

A determinação de um ponto o interior ao fecho convexo não oferece problemas. Por exemplo, o baricentro $o = \frac{1}{n}\sum_{i=1}^{n} p_i$ é um ponto interior ao fecho convexo, obtido em tempo O(n).

Suponhamos, então, que os pontos $p_1, p_2, ..., p_n$ estejam ordenados polarmente em torno de o no sentido anti-horário (em caso de mais de um ponto estar no mesmo raio partindo de o, tomamos apenas o mais distante de o, descartando os demais). Como vimos no capítulo 1, esta ordenação pode ser executada em tempo O(n log n).

O resultado da etapa de ordenação é um polígono que pode ser visto como uma primeira aproximação de conv(C). O polígono resultante $p_1...p_n$ possui a propriedade de que existe um ponto interno (o) que "vê" cada vértice do polígono (figura 3.3). Polígonos que satisfazem esta propriedade são chamados **estrelados**. Desta forma, a etapa de ordenação reduz o problema de encontrar o fecho convexo de $\{p_1, p_2, ..., p_n\}$ ao problema (com mais estrutura) de encontrar o fecho convexo de um polígono estrelado. Este problema tem mais estrutura que o anterior porque a ordem dos vértices em conv(C) é a dada pela ordenação polar e o problema se resume em determinar quais dos pontos de C são efetivamente vértices de conv(C).

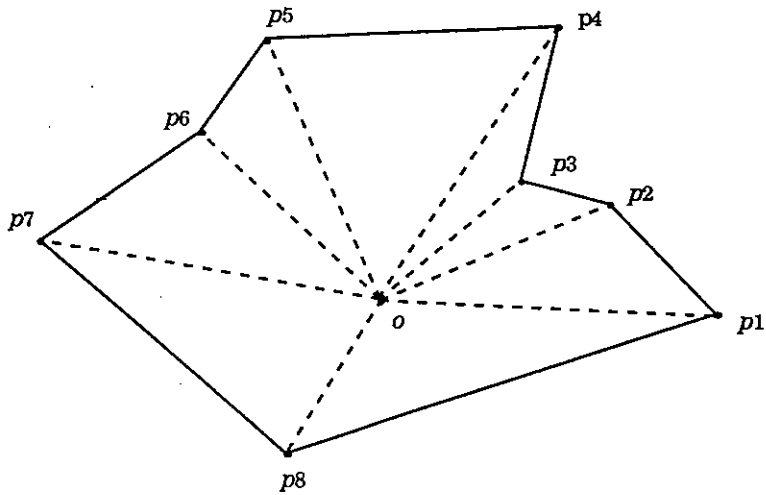


Figura 3.3 – Obtendo um polígono estrelado.

Para obter um algoritmo eficiente para este novo problema, fazemos a seguinte observação. Um polígono simples $p_1...p_n$ é convexo se e somente se cada ângulo interno $\angle(p_{i-1}p_ip_{i+1})$ é convexo. Isto é, se cada triângulo $p_{i-1}p_ip_{i+1}$ tem a mesma orientação que o polígono. Além disso, se um vértice p_i de $p_1p_2...p_n$ viola esta condição, então certamente p_i não é vértice de conv($p_1...p_n$) (ver exercício 3).

Baseados nesta observação, podemos obter o seguinte algoritmo para encontrar o fecho convexo de um polígono estrelado $P = p_1p_2...p_n$: testamos se o ângulo interno em cada p_i é convexo; em caso afirmativo, o polígono estrelado é convexo e portanto é igual a conv(P); caso contrário, removemos de P um ponto p_i que viola a condição de convexidade e repetimos o teste para o novo polígono estrelado. O processo continua até que todos os pontos restantes determinem ângulos convexos; neste ponto, conv(P) terá sido determinado.

O algoritmo esboçado acima tem um ponto crucial: se, em um polígono estrelado $p_1...p_i...p_n$, removemos o vértice p_i de um ângulo côncavo, então o polígono restante $p_1p_2...p_{i-1}p_{i+1}...p_n$ é novamente um polígono estrelado. Isto permite a redução sucessiva do tamanho do polígono estrelado até torná-lo convexo. O algoritmo resultante tem complexidade $O(n^2)$, já que a verificação de convexidade requer tempo $O(n)$ e o número de pontos a remover

é também $O(n)$. Além disso, é fácil conceber um exemplo em que estas cotas superiores sejam atingidas.

No entanto, é possível reescrever o algoritmo acima de modo a transformá-lo num algoritmo linear, evitando voltar à estaca zero cada vez que um ponto p_i é retirado do conjunto C. Começamos determinando um vértice de conv(C) (por exemplo, escolhendo o vértice de maior abscissa). Sem perda de generalidade, suponhamos que p_1 seja o vértice escolhido. Desta forma, o ângulo $\angle(p_n p_1 p_2)$ é necessariamente convexo. O algoritmo examina sucessivamente os vértices $p_2, p_3, ...$, até encontrar um p_i para o qual o ângulo interno correspondente não seja convexo. Neste ponto, ao invés de recomeçar todo o processo, observamos que a remoção de p_i apenas afeta o teste de convexidade do vértice imediatamente anterior p_{i-1}. Se o ângulo $p_{i-2}p_{i-1}p_{i+1}$ continuar sendo convexo, o algoritmo pode continuar, examinando o ângulo em p_{i+1}. Se o ângulo em p_{i-1} for côncavo, p_{i-1} é eliminado e o ângulo em seu predecessor reavaliado.

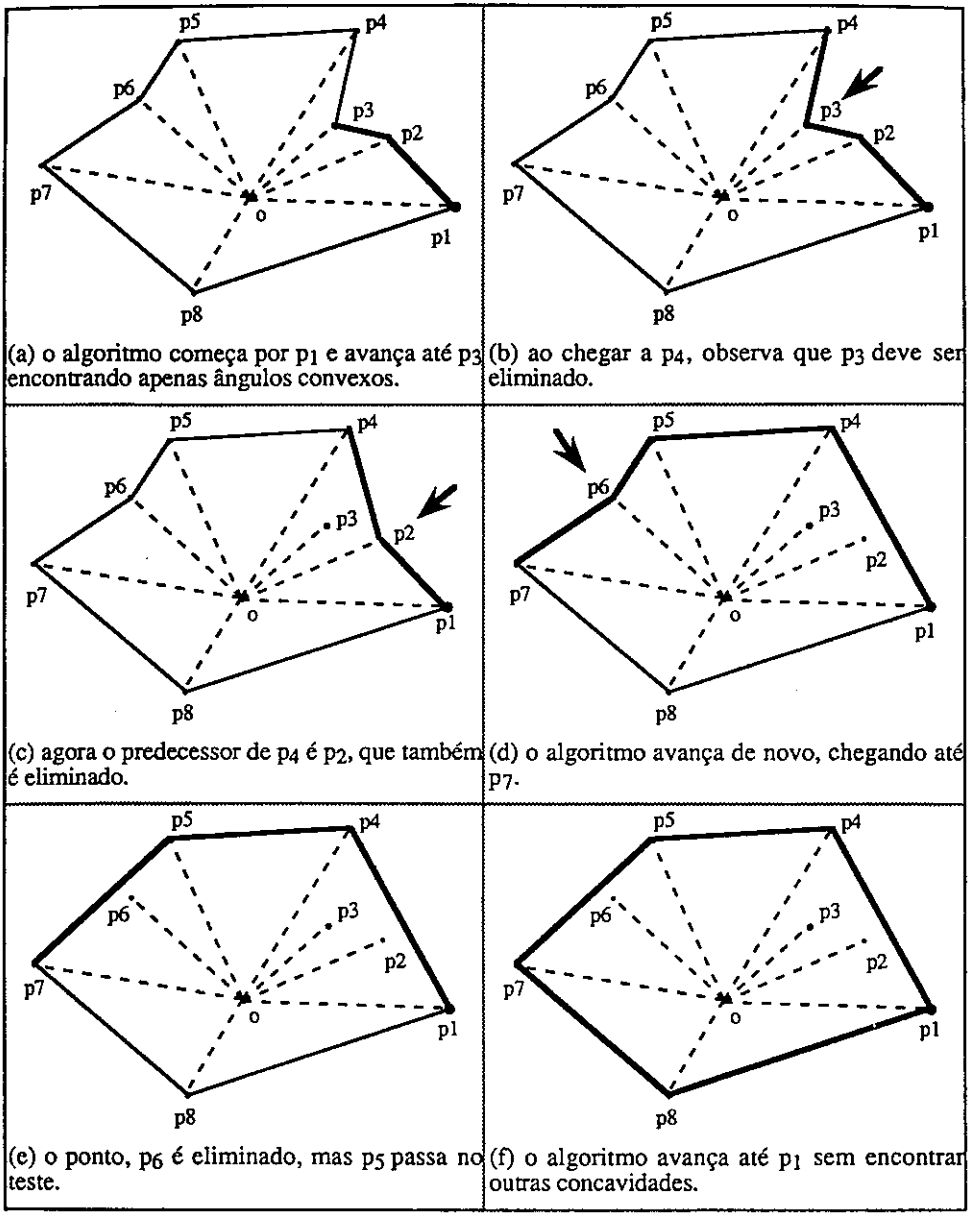
As idéias acima produzem um algoritmo, devido a Graham [G], que é descrito mais formalmente a seguir.

Algoritmo 3.2: (Algoritmo de Graham)

```
Pn+1 = P1                          {feche o ciclo}
q1 = p1                            {p1 é um ponto do fecho convexo}
q2 = p2                            {inclua p2 tentativamente}
h = 2
para i = 3..n+1
      enquanto h > 1 e ∠(qh-1qhpi) não for convexo
            h = h - 1              {remova qh}
      h = h + 1
      qh = pi                      {inclua pi tentativamente}
```

Ao final do algoritmo acima, q contém os h vértices do fecho convexo. Note que é possível usar p no lugar de q já que h é sempre menor que i.

A figura 3.4 mostra diversos passos obtidos quando se aplica este algoritmo ao polígono estrelado da figura 3.3.



(a) o algoritmo começa por p_1 e avança até p_3 encontrando apenas ângulos convexos.

(b) ao chegar a p_4, observa que p_3 deve ser eliminado.

(c) agora o predecessor de p_4 é p_2, que também é eliminado.

(d) o algoritmo avança de novo, chegando até p_7.

(e) o ponto, p_6 é eliminado, mas p_5 passa no teste.

(f) o algoritmo avança até p_1 sem encontrar outras concavidades.

Figura 3.4 – Algoritmo de Graham.

Teorema 3.3: *O algoritmo de Graham corretamente encontra o fecho convexo de um polígono estrelado $p_1p_2...p_n$ em $O(n)$ passos.*

Prova: Note, inicialmente, que o algoritmo termina por que os dois *loops* são limitados. Além disso, cada ponto é incluído ou eliminado da lista q no máximo uma vez, o que mostra que o algoritmo é linear. Finalmente, como apenas pontos que podem ser vértices são incluídos em q e como todos os ângulos nos vértices q_1, ..., q_h determinam um ângulo convexo com seu antecessor e seu sucessor, a lista q final é o fecho convexo de $p_1p_2...p_n$. ■

Desta forma, o algoritmo que consiste em ordenar polarmente os pontos de C_i e, em seguida, aplicar o algoritmo de Graham, obtém o fecho convexo de C em tempo $O(n \log n)$ e é, portanto, ótimo para FC2D.

O algoritmo acima mostra que a cota inferior de $\Omega(n \log n)$ para um algoritmo capaz de resolver FC2D se aplica apenas para o problema geral, em que os n pontos de C não satisfazem qualquer propriedade adicional. Como vimos acima, caso se possua a informação adicional que os pontos são, na ordem dada, vértices de um polígono estrelado, a complexidade do algoritmo pode ser reduzida para $O(n)$. O mesmo ocorre para polígonos simples em geral. Isto é, sabendo que $p_1p_2...p_n$ é um polígono simples (nesta ordem), é possível encontrar o fecho convexo de $p_1p_2...p_n$ em tempo $O(n)$, com um algoritmo apresentado por Graham e Yao [GY]. No entanto, o algoritmo é bem mais complexo do que o que apresentamos acima.

Algoritmos do tipo Dividir-para-Conquistar

Nesta seção, examinamos algoritmos para FC2D que usam, de forma explícita, o paradigma de dividir-para-conquistar. A idéia básica é dividir o conjunto C em conjuntos menores C_1 e C_2 (SEPARAÇÃO), obter seus fechos convexos conv(C_1) e conv(C_2) (RECURSÃO), e combiná-los de modo a obter conv(C) (COMBINAÇÃO). O passo mais importante é o de COMBINAÇÃO, onde, dados polígonos convexos P_1 e P_2, desejamos obter conv $(P_1 \cup P_2)$.

Examinaremos dois algoritmos que implementam estas idéias. Estes algoritmos guardam uma estreita analogia com os algoritmos do tipo dividir-para-conquistar que apresentamos para o problema de ordenação. O primeiro algoritmo, conhecido como **Quickhull** (em analogia com

Quicksort), separa C em dois conjuntos de tal modo que a obtenção de conv($C_1 \cup C_2$) seja trivial, a partir de conv(C_1) e conv(C_2) (em compensação, tal algoritmo não tem controle sobre os tamanhos de C_1 e C_2 e, por esta razão, não consegue atingir a cota inferior de O(n log$n$) para a complexidade). O segundo algoritmo, conhecido como **Mergehull** (em analogia com Mergesort), consegue obter a complexidade de O(n log n), dividindo C em conjuntos de mesmo tamanho e obtendo conv($C_1 \cup C_2$) em tempo linear.

Algoritmo Quickhull

O algoritmo Quicksort descrito no capítulo 1 opera dividindo o conjunto C a ser ordenado em dois subconjuntos C_1 e C_2. O conjunto C_1 contém os elementos de C que são menores que um certo elemento, enquanto C_2 contém os demais elementos. Desta forma, uma vez tendo ordenado C_1 e C_2, para se ordenar C basta concatenar as listas ordenadas.

De maneira análoga, o algoritmo Quickhull recursivamente divide o conjunto C em subconjuntos C_1 e C_2 de forma que conv(C) possa ser obtido simplesmente concatenando as listas de arestas de C_1 e C_2. Para tal, basta observar que, se e e d são dois vértices quaisquer de conv(C), então os conjuntos C_1 e C_2 obtidos tomando os pontos de C que estão em cada um dos semiplanos determinados por ed possuem esta propriedade.

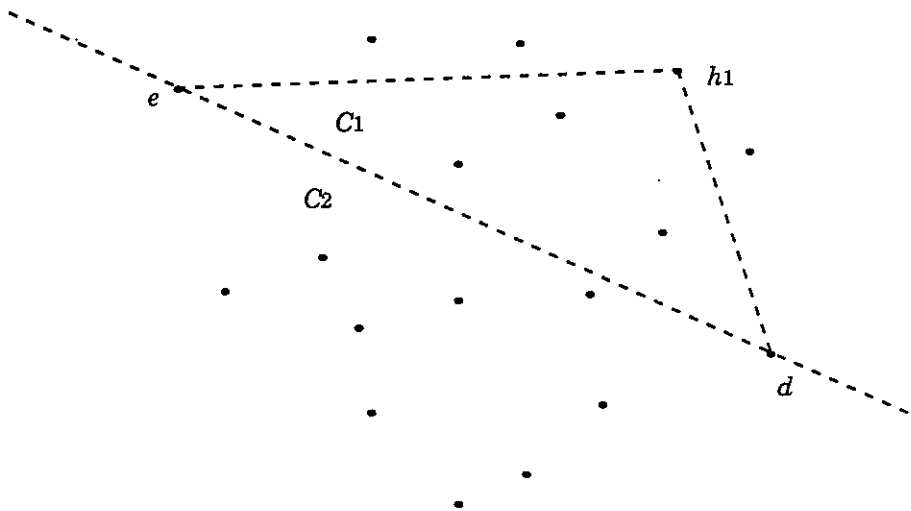


Figura 3.5 – Subdividindo C.

Para obter conv(C_1), subdividimos C_1 da seguinte forma. Dentre todos os pontos de C_1, tomamos h_1 tal que o triângulo ehd tenha área máxima (isto é, o ponto h cuja distância à reta ed é máxima) (figura 3.5). Caso haja mais de um ponto h com esta propriedade, tomamos, entre todos os pontos que determinam a área máxima, aquele para o qual o ângulo $\angle(h_1ed)$ seja máximo. Então:

i) o ponto h_1 certamente pertence a conv(C_1). Note que a reta paralela a ed passando por h_1 é uma reta **suporte** de conv(C_1) (não há pontos de C_1 acima desta reta). O ponto h_1, mesmo que não seja o único ponto de C_1 sobre a reta, certamente é o ponto mais à esquerda e, portanto, é um ponto extremo de conv(C_1).

ii) As retas eh_1 e h_1d (consideradas orientadas desta maneira) dividem C_1 em três subconjuntos. Os pontos situados à direita de eh_1 e à esquerda de h_1d não podem ser vértices de conv(C_1), já que são interiores ao triângulo eh_1d. Os pontos situados à esquerda de eh_1 formam um subconjunto que designaremos C_{11} enquanto os situados à direita de h_1d formam um subconjunto C_{22}. O fecho convexo de C_1 pode, então, ser obtido simplesmente concatenando os vértices em conv(C_{11}) e conv(C_{22}).

Para descrever o algoritmo resultante das idéias acima, utilizaremos a função `Quickhull(C, e, d)`. Seus argumentos são dois pontos e e d do plano e um conjunto C de pontos situados estritamente à esquerda de ed. A função retorna a lista ordenada das arestas de conv($C \cup \{e,d\}$).[1]

Algoritmo 3.3: Quickhull(C, e, d)

```
Se C = {e, d}, retorne o segmento orientado ed
Senão:
      h = ponto de C tal que S_edh seja máxima;
      C1 = conjunto de pontos de C à esquerda de eh;
      C2 = conjunto de pontos de C à esquerda de hd;
      Retorne QUICKHULL(C1, e, h) concatenado com QUICKHULL(C2, h, d).
```

[1] Note que, para facilitar a descrição do algoritmo, decidimos não incluir os pontos e e d em C, o que difere ligeiramente da exposição dos parágrafos anteriores.

Para obter o fecho convexo de um conjunto C podemos, por exemplo, encontrar uma aresta ed de conv(C) (usando, por exemplo o algoritmo de Jarvis). Basta, então fazer uma única chamada a `Quickhull(C, e, d)`.

Uma outra possibilidade é escolher e e d, respectivamente, como os pontos de abscissa mínima e máxima em C e determinar os conjuntos C_1 e C_2, formado pelos pontos de C situados acima e abaixo da reta ed, respectivamente. Para obter conv(C), basta então chamar `Quickhull(C1, e, d)` e `Quickhull(C2, d, e)` e concatenar os resultados.

A figura 3.6 mostra o resultado obtido ao se aplicar o algoritmo Quickhull a um conjunto de pontos do $\mathbf{R}^2$, tendo sido empregada a segunda estratégia descrita acima para dar início ao processo. Os vértices do fecho convexo (exceto e e d) são representados por $h_{i_1 \ldots i_k}$, onde os índices indicam em que nível de recursão cada vértice foi encontrado. No caso da figura, o vértice mais "profundo" foi encontrado no terceiro nível de recursão.

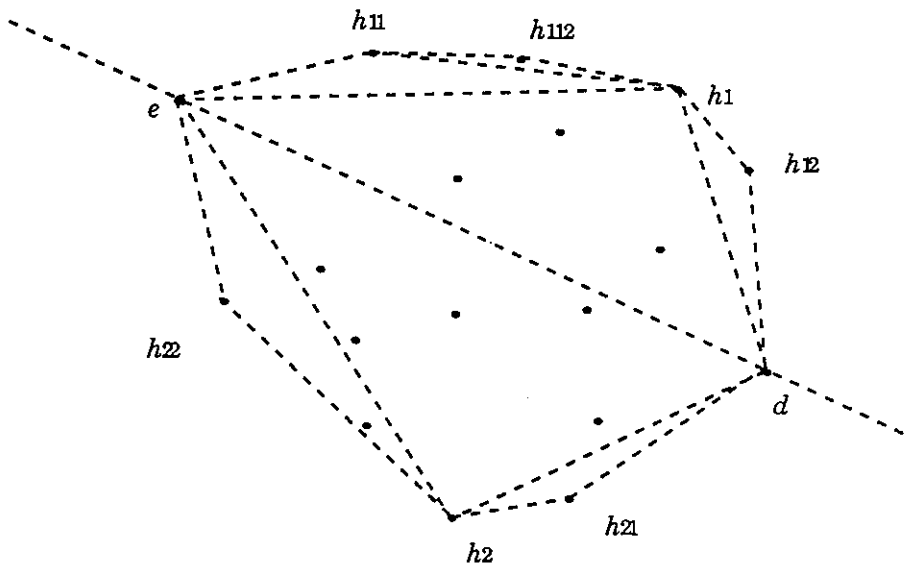


Figura 3.6 – Aplicando o algoritmo Quickhull.

A analogia entre os algoritmos Quicksort e Quickhull se manifesta também nos resultados de complexidade computacional. Como Quicksort, Quickhull é um algoritmo simples de implementar e que funciona razoavelmente bem caso os pontos de C sejam escolhidos aleatoriamente no plano. No entanto, no pior caso, seu desempenho é apenas $O(n^2)$. É possível que, em cada nível de recursão, Quickhull divida C em dois conjuntos de forma extremamente desiquilibrada: um dos conjuntos possui apenas os pontos d e h (onde h é o ponto à distância máxima de ed), estando todos os demais no outro conjunto (exercício 3). Em consequência, o algoritmo pode envolver um número linear de separações, cada uma delas requerendo tempo linear, resultando daí desempenho apenas $O(n^2)$.

Algoritmo Mergehull

Para garantir um desempenho $O(n \log n)$, um algoritmo de dividir-para-conquistar deve separar o conjunto C em dois conjuntos C_1 e C_2, cada um tendo a metade dos elementos. O algoritmo Mergehull utiliza esta estratégia.

Algoritmo 3.4: Mergehull(C)

```
(SEPARAÇÃO)   Divida C em subconjuntos C1 e C2 de mesmo tamanho.
(RECURSÃO)    Obtenha conv(C1) e conv(C2).
(COMBINAÇÃO)  Determine conv(C) a partir de conv(C1) e conv(C2).
```

Em contraste com Quickhull, onde a etapa de COMBINAÇÃO é trivial, a etapa crucial do algoritmo descrito acima é aquela em que os fechos convexos de C_1 e C_2 são combinados para produzir o fecho convexo de C. Isto é, necessitamos um algoritmo eficiente para resolver o seguinte problema:

FECHO CONVEXO DA UNIÃO DE POLÍGONOS CONVEXOS: *Dados polígonos convexos* P_1 *e* P_2, *obter o fecho convexo de* $P_1 \cup P_2$.

Como vimos na discussão sobre algoritmos do tipo dividir-para-conquistar (capítulo 1), para que possamos garantir que Mergehull opera em tempo $O(n \log n)$ é necessário que sejamos capazes de obter $\text{conv}(P_1 \cup P_2)$ em tempo linear no número total de vértices de P_1 e P_2.

É óbvio que $\text{conv}(P_1 \cup P_2)$ pode ser obtido em tempo $O(n \log n)$, onde n é o número total de vértices de P_1 e P_2. Basta aplicar ao conjunto obtido reunindo os vértices de cada

polígono qualquer dos algoritmos ótimos para fecho convexo (por exemplo, o algoritmo de Graham). No entanto, a etapa crítica do algoritmo de Graham (responsável pela complexidade $O(n \log n)$) é aquela em que os pontos do conjunto C são ordenados polarmente em torno de um ponto interior a conv(C). Como P_1 e P_2 são polígonos convexos, os vértices de cada um já estão ordenados em torno de pontos interiores a cada um dos polígonos. A idéia é utilizar esta informação adicional para, em tempo linear, obter uma lista em que todos os vértices estejam polarmente ordenados em torno de um ponto p_0 de conv($P_1 \cup P_2$)

Começamos tomando um ponto p_0 interior a P_1 (este ponto pode ser obtido, por exemplo, tomando o baricentro do triângulo determinado por três vértices quaisquer de P_1). Certamente p_0 é também interior a conv($P_1 \cup P_2$). Vamos admitir que os vértices de P_1 e P_2 estejam ordenados no sentido anti-horário em cada polígono e mostrar como ordenar todos eles em torno de p_0. Há dois casos a considerar:

i) p_0 também é interior a P_2 (figura 3.7)

Neste caso, ambos os polígonos têm seus vértices ordenados em torno de p_0. Portanto, para ordenar todos os vértices em uma lista comum, basta combinar as duas listas ordenadas, o que certamente pode ser feito em tempo linear no número total de vértices de P_1 e P_2.

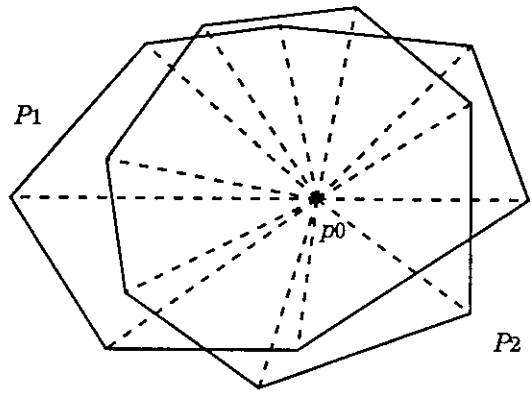


Figura 3.7 – P_1 e P_2 têm vértices em ordem polar em torno de um ponto interior comum.

ii) p_0 não é interior a P_2 (figura 3.8).

Agora P_2 está contido em um ângulo de vértice p_0 e definido pelas semi-retas p_0u e p_0v, onde u e v são vértices de P_2. Os vértices u e v separam a lista de vértices de P_2 em duas sublistas: uma ordenada segundo ângulos polares (em torno de p_0) crescentes e a outra ordenada segundo ângulos polares decrescentes. A segunda destas listas corresponde aos vértices de P_2 que são interiores ao triângulo p_0uv e pode, portanto, ser descartada. A primeira lista pode ser combinada com a lista dos vértices de P_1 em tempo $O(n)$, onde n é o número total de vértices de P_1 e P_2.

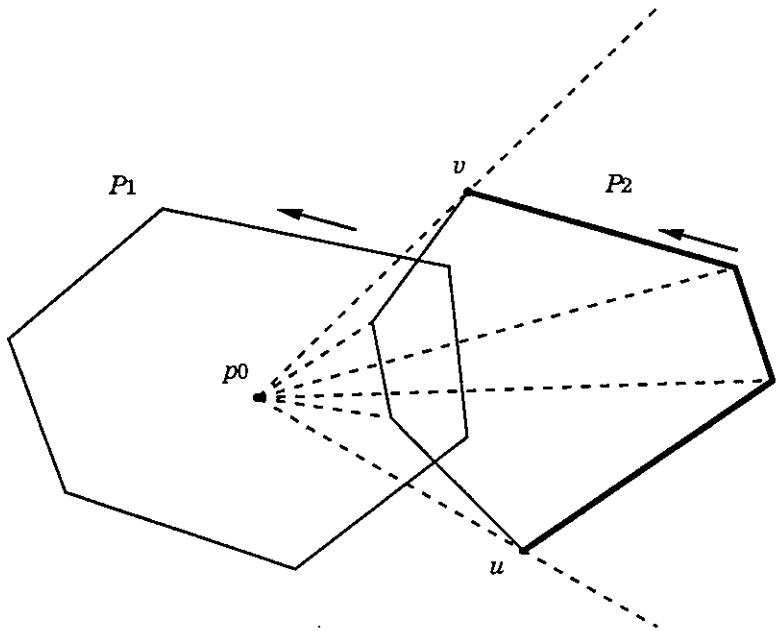


Figura 3.8 – Os vértices de P_2 entre u e v estão ordenados em torno de p_0.

Para obter os pontos u e v basta percorrer a lista de vértices de P_2, calculando o ângulo orientado em torno de p_0 correspondente a cada vértice e determinando os pontos u e v onde a monotonicidade da sequência de valores do ângulo orientado é alterada. Note que, se não

houver alteração de monotonicidade nesta sequência, podemos concluir que o ponto p_0 é interior a P_2 e aplicar a análise em (i).

Temos assim o seguinte algoritmo:

Algoritmo 3.5: Fecho convexo da união de polígonos convexos (P_1, P_2)

1. Determine um ponto p_0 interior a P_1.
2. Percorra a lista de vértices de P_2 determinando a sublista de extremos u e v em que os ângulos polares em torno de p_0 estão em ordem crescente (caso p_0 seja interior a P_2 todos os vértices de P_2 estarão nesta lista).
3. Combine a lista de vértices de P_1 e a lista obtida no passo 2 (ambas ordenadas polarmente em torno de p_0) em uma única lista ordenada em torno de p_0.
4. Aplique o último passo do algoritmo de Graham a esta lista ordenada.

Cada uma das etapas acima é executada em tempo $O(n)$. Portanto, temos o seguinte:

Teorema 3.4: *O fecho convexo da união de dois polígonos convexos P_1 e P_2 pode ser determinado em tempo proporcional ao número total de vértices de P_1 e P_2. Em consequência, o algoritmo Mergehull determina o fecho convexo de um conjunto de pontos do plano em tempo ótimo $\theta(n \log n)$.* ■

3.4 Fecho convexo no R^3

Consideramos agora o seguinte problema:

FC3D: *Dado um conjunto C de pontos $p_1, p_2, ..., p_n$ do R^3, obter o fecho convexo de C.*

Inicialmente, é necessário estabelecer, com precisão, o que entendemos por "determinar" conv(C). No caso plano, conv(C) é descrito simplesmente através de uma lista ordenada (isto é, unidimensional) de seus vértices. Esta descrição imediatamente fornece a lista de lados de conv(C) e permite determinar que pares de vértices e arestas são adjancentes.

No caso de FC3D, não nos contentamos em determinar quais dos pontos de C são vértices de conv(C). Desejamos obter uma quantidade de informação a respeito de conv(C) que permita determinar imediatamente como os vértices de conv(C) estão organizados. Isto é, como eles

estão agrupados determinando arestas e faces de conv(C). Na próxima seção estudamos estruturas de dados capazes de prover este tipo de informação.

Estruturas de dados para a fronteira de poliedros tridimensionais

Embora estejamos diretamente interessados na obtenção de estruturas capazes de representar poliedros tridimensionais convexos, a discussão a seguir vale para poliedros mais gerais. Especificamente, englobaremos em nossa discussão quaisquer poliedros conexos de faces planas, que sejam topologicamente equivalentes a uma esfera e tais que a fronteira de cada face seja conexa. A maior parte das abordagens aqui descritas podem ser estendidas para poliedros mais gerais, mas esta discussão está fora do escopo deste trabalho. O leitor deve consultar um livro sobre Modelagem de Sólidos (por exemplo, [Ma]), para um tratamento mais completo do assunto.

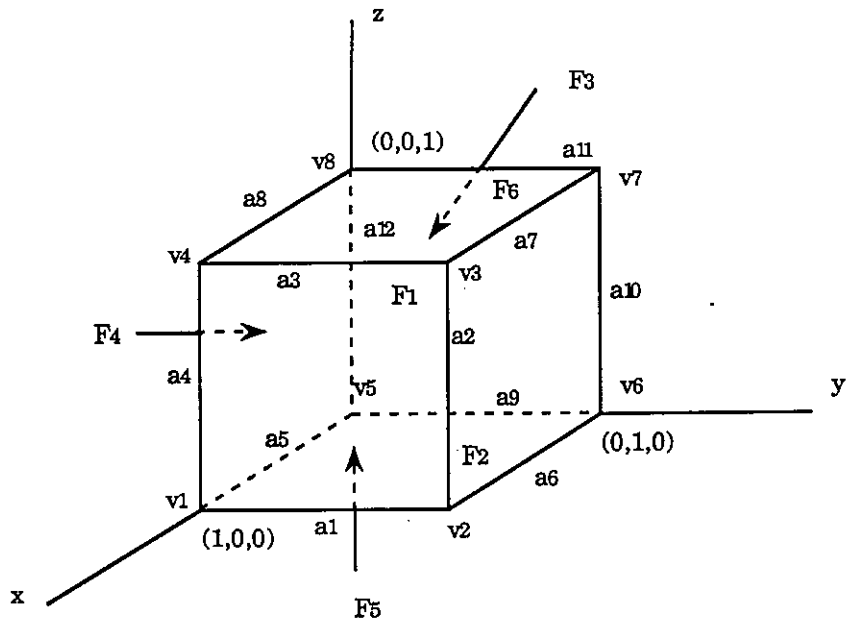
Lista de faces

A forma mais primitiva de descrever a fronteira de um poliedro tridimensional consiste em listar os vértices de cada face. Exigimos que estas listas sejam orientadas de modo consistente. Em geral, optamos por orientá-las no sentido anti-horário quando observadas de fora do poliedro (isto é, de modo que o vetor área de cada face (introduzido no capítulo 1) esteja orientado de dentro para fora do poliedro).

Ao invés de listar os vértices de cada face, muitas vezes se fornece uma lista dos vértices, com suas coordenadas e, ao listar as faces, são dados apenas os nomes dos vértices respectivos. Por exemplo, a figura 3.9 mostra um cubo unitário e sua representação através de listas de vértices e faces.

Estruturas de Adjacência

A grande desvantagem da estrutura descrita anteriormente é que ela só contém um tipo de informação de adjacência entre vértices, arestas e faces de um poliedro: dada uma face, consegue-se listar os vértices desta face em tempo proporcional ao número de vértices. Outras questões de adjacência, no entanto, requerem uma busca exaustiva na estrutura. Por exemplo,



Vértices:

v_1: (1,0,0) v_5: (0,0,0)
v_2: (1,1,0) v_6: (0,1,0)
v_3: (1,1,1) v_7: (0,1,1)
v_4: (1,0,1) v_8: (0,0,1)

Faces:

f1: $v_1v_2v_3v_4$
f_2: $v_2v_6v_7v_3$
f_3: $v_6v_5v_8v_7$
f_3: $v_5v_1v_4v_8$
f_5: $v_1v_5v_6v_2$
f_6: $v_4v_3v_7v_8$

Figura 3.9 – Um cubo e suas listas de vértices e faces.

para determinar as faces que são adjacentes a uma dada face F, é necessário percorrer cada face em busca de uma aresta comum com F.

Para obter estas informações de adjacência da melhor forma possível é necessário reconhecer que a fronteira de um poliedro convexo tem a mesma estrutura de um **diagrama planar**, isto é, da realização no plano de um grafo planar. A estrutura do diagrama planar

associado a um poliedro define certas propriedades topológicas do poliedro (ver, por exemplo, [He]). Por esta razão, estruturas de dados que descrevem tal diagrama são chamadas de **estruturas de dados topológicas.**

Um diagrama planar divide o plano em faces, que são limitadas por arestas, que por sua vez são limitadas por vértices. Por exemplo, a estrutura facial do cubo unitário da figura 3.9 pode ser representado através do diagrama da figura 3.10. O número F de faces, A de arestas e V de vértices de um diagrama planar determinado por um grafo conexo, estão relacionados segundo a chamada **fórmula de Euler**:

$$V + F = A + 2.$$

(veja, por exemplo, [Ha] para um estudo mais aprofundado sobre grafos planares e suas realizações no plano.)

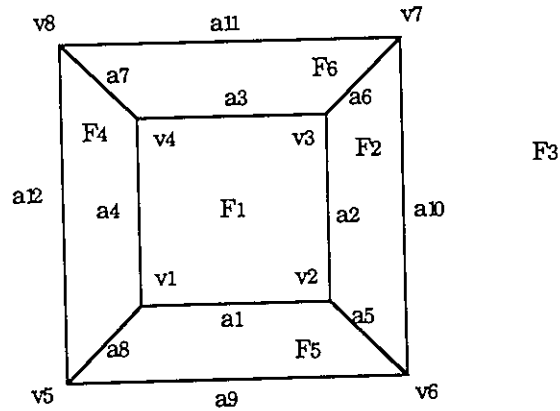


Figura 3.10 – Diagrama planar representando um cubo.

Uma representação eficiente de um diagrama planar deve ser capaz de produzir todas as possíveis relações de adjacência entre vértices, arestas e faces de um diagrama em tempo ótimo. Por exemplo, dadas duas faces do diagrama, deve-se poder determinar se elas são adjacentes em tempo linear no número total de vértices destas faces. Dado um vértice, deve-se poder listar as

faces que lhe são adjacentes em tempo proporcional ao seu número. A chave para tal eficiência é a representação adequada dos ciclos de arestas em torno de cada face e de cada vértice.

Diversas estruturas de dados foram propostas na literatura para esta finalidade (ver, por exemplo, [Ma, GS, We]). Todas estas estruturas de dados tiram partido do fato que um diagrama planar é **orientável**. Ou seja, ao percorrer as fronteiras das faces do diagrama no sentido anti-horário[2], cada aresta é percorrida duas vezes, uma vez em cada sentido.

Neste trabalho, utilizaremos a estrutura ***winged-edge*** (ou de "arestas aladas")—a mais clássica delas. Introduzida por Baumgart [Ba], a estrutura *winged-edge* possui uma lista de vértices, uma lista de faces e uma lista de arestas, organizadas da seguinte maneira :

Vértices: para cada vértice, são armazenados suas coordenadas (x_1, x_2, x_3) e uma das arestas (**av**) que lhe são incidentes:

Faces: para cada face, é armazenada uma das arestas (**af**) que lhe são incidentes.

Arestas: a lista de arestas concentra a maior parte da informação contida na estrutura. Cada aresta a é representada por um par de vértices (v_1 e v_2), dados por seus nomes. A ordem em que estes vértices são dados fornece uma orientação para esta aresta. Baseado nesta orientação, a face adjacente à aresta a e situada à sua esquerda será representada por **fccw** (*counterclockwise*, já que a orientação induzida por a nesta face é anti-horária), enquanto a face situada à direita de a é identificada por **fcw** (*clockwise*). Observe que a face **fccw** é aquela na qual a induz uma orientação positiva. Para permitir a obtenção eficiente dos ciclos de arestas adjacentes a cada vértice, a estrutura *winged-edge* contém os nomes das arestas que antecedem e sucedem a nas faces **fccw** e **fcw**. Assim, **pccw** (*previous counterclockwise*) e **nccw** (*next counterclockwise*) representam as arestas anterior e seguinte a a no ciclo da face **fccw** (orientado de acordo com a orientação de a, isto é, no sentido horário), enquanto **pcw** e **ncw** indicam as arestas anterior e seguinte à aresta a na face **fcw** (considerada orientada no sentido positivo, ou seja, de modo oposto ao induzido por a). A figura 3.11 ilustra o significado dos campos **v_1, v_2, fccw, fcw, pccw, nccw, pcw e ncw**.

---

[2] Para a face ilimitada do diagrama, a fronteira deve ser percorrida no sentido *horário*; observe que, de forma consistente com o que ocorre nas faces limitadas, o interior da face ilimitada estará sempre à esquerda de cada aresta.

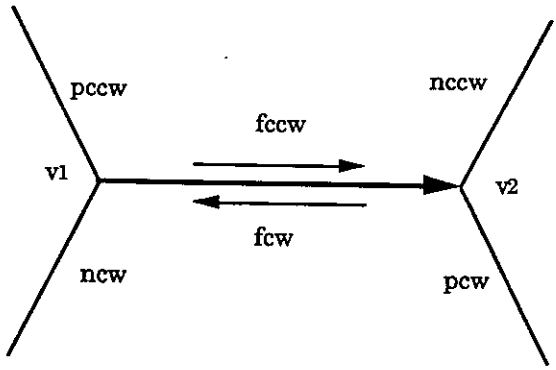
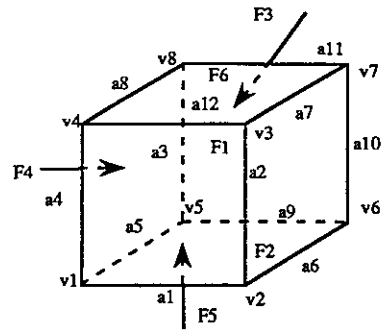


Figura 3.11 – Informação associada a cada aresta na estrutura *winged-edge*.

A estrutura *winged-edge* correspondente ao cubo da figura 3.9 é dada na figura 3.12. No exercício 11, o leitor é convidado a verificar que a estrutura *winged-edge* permite a obtenção eficiente de informações relativas à adjacência de vértices, arestas e faces. Esta propriedade será essencial para alguns algoritmos que manipulam poliedros tridimensionais ou subdivisões do plano.

3.5 Algoritmos para fecho convexo no R^3

Como vimos na seção 3.2, a cota inferior $\Omega(n \log n)$ que foi obtida para FC2D é trivialmente válida para dimensões $d > 2$. Nesta seção examinaremos um algoritmo simples para FC3D, que guarda um paralelo com o algoritmo de Jarvis e que resolve FC3D em tempo $O(n^2)$. Um outro algoritmo, bem mais complexo, que atinge a complexidade ótima $O(n \log n)$ será descrito em linhas gerais. A discussão nesta seção assume que os poliedros convexos a serem gerados serão armazenados em uma estrutura tipo *winged-edge*.



Vértices	(x_1, x_2, x_3)	av
v_1	(1,0,0)	a_1
v_2	(1,1,0)	a_2
v_3	(1,1,1)	a_3
v_4	(1,0,1)	a_4
v_5	(0,0,0)	a_9
v_6	(0,1,0)	a_{10}
v_7	(0,1,1)	a_{11}
v_8	(0,0,1)	a_{12}

Faces	af
F_1	a_1
F_2	a_2
F_3	a_{10}
F_4	a_{12}
F_5	a_1
F_6	a_8

Arestas	v_1	v_2	fccw	fcw	pccw	nccw	pcw	ncw
a_1	v_1	v_2	F_1	F_5	a_4	a_2	a_6	a_5
a_2	v_2	v_3	F_1	F_2	a_1	a_3	a_7	a_6
a_3	v_3	v_4	F_1	F_6	a_2	a_4	a_8	a_7
a_4	v_4	v_1	F_1	F_4	a_3	a_1	a_5	a_8
a_5	v_1	v_5	F_5	F_4	a_1	a_9	a_{12}	a_4
a_6	v_2	v_6	F_2	F_5	a_2	a_{10}	a_9	a_1
a_7	v_3	v_7	F_6	F_2	a_3	a_{11}	a_{10}	a_2
a_8	v_4	v_8	F_4	F_6	a_4	a_{12}	a_{11}	a_3
a_9	v_5	v_6	F_5	F_3	a_5	a_6	a_{10}	a_{12}
a_{10}	v_6	v_7	F_2	F_3	a_6	a_7	a_{11}	a_9
a_{11}	v_7	v_8	F_6	F_3	a_7	a_8	a_{12}	a_{10}
a_{12}	v_8	v_5	F_4	F_3	a_8	a_5	a_9	a_{11}

Figura 3.12 – Estrutura *winged-edge* correspondente ao cubo unitário.

Algoritmo de Chand e Kapur ("Embrulho-para-Presente")

A idéia básica deste algoritmo pode ser vista como a extensão para dimensões $d > 2$ do passo fundamental do algoritmo de Jarvis. Embora a idéia funcione para poliedros de dimensão arbitrária, limitaremos nossa discussão ao caso $d = 3$ (veja [PS] para o caso geral).

Suponhamos que, de alguma forma, tenhamos determinado uma face F de $P = \text{conv}(C)$. Se todas as faces adjacentes a F já tiverem sido determinadas, então F já está explorada. Caso contrário, veremos a seguir como obter a face adjacente a F em uma de suas arestas. O algoritmo de *gift-wrapping* constrói a fronteira de P gerando as faces adjacentes às já criadas, até que todas as faces tenham sido exploradas.

Seja $a = p_1p_2$ uma aresta de F tal que a face adjacente a F em a ainda é desconhecida. Como F é uma face de P, todos os pontos de C estão situados em um dos semi-espaços definidos por F. A face F' adjacente a F em a é tal que todos os pontos de C estão situados em um ângulo diedral definido pelos planos de F e F'. Desta forma, dentre todos os semiplanos determinados por a e por pontos de C, a face F' está contida naquele que forma ângulo máximo com o semiplano conduzido por a e contendo F (veja a figura 3.13).

Para cada ponto p de C, precisamos determinar o ângulo $\theta_p \in [0, \pi]$ que o semiplano $\alpha(a, p)$ definido por a e p forma com o semiplano F. Para tal, observamos que o ângulo entre os vetores n e n_p, normais a F e $\alpha(a, p)$, respectivamente, é igual a $\pi - \theta_p$. O vetor n_p pode ser calculado através da expressão

$$n_p = p_1p \times p_1p_2$$

e portanto

$$\cos\theta_p = -\frac{n_p.n}{\|n_p\|\,\|n\|}.$$

A fim de maximizar o ângulo entre F e $\alpha(a, p)$ devemos minimizar o seu cosseno. Isto é o plano da face F' adjacente a F em a é determinado por a e pelo ponto p tal que $\cos\theta_p$ mínimo.

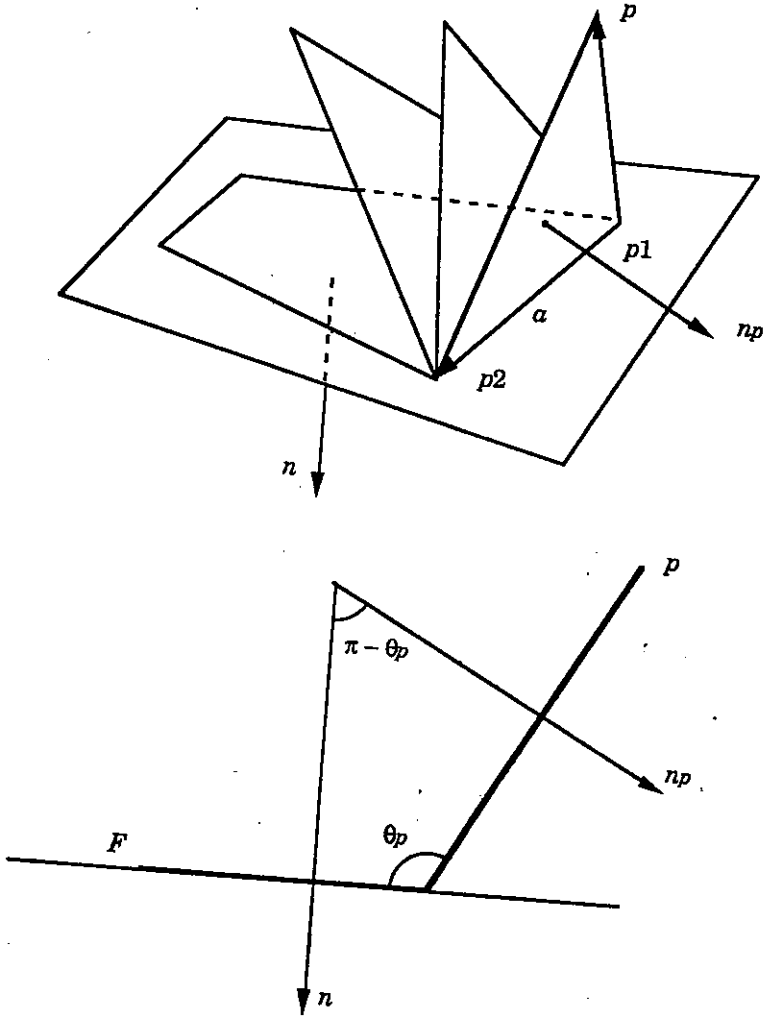


Figura 3.13 – Fazendo o "embrulho-para-presente".

Caso este mínimo seja atingido em um único ponto p, a face F' é simplesmente o ngulo de vértices p_1, p_2 e p. Caso contrário, é necessário obter o fecho convexo do junto $C_{F'}$ formado por p_1, p_2 e todos os pontos p para os quais cos θ_p é mínimo. Este é

um problema de determinação de fecho convexo em duas dimensões e pode ser resolvido aplicando-se qualquer dos algoritmos para FC2D ao conjunto C_F, (talvez após projetar os pontos de C_F, em um dos planos xy, xz ou yz).

O passo descrito acima (que chamaremos de passo **embrulho-para-presente**) mostra como expandir a descrição do fecho convexo a partir de uma face cujas faces vizinhas não estiverem ainda determinadas. Isto é, mostramos como implementar uma função "embrulho-para-presente" (F, a) que, dadas uma face F e uma de suas arestas a, retorna a face F' adjacente a F em a.

Para que possamos escrever o algoritmo que gera conv(C), é necessário discutir um detalhe: como obter uma primeira face F de conv(C) para dar início ao processo. Isto é análogo ao processo de obtenção do primeiro vértice para o algoritmo de Jarvis.

Como no algoritmo de Jarvis, começamos por determinar o ponto p_1 de coordenada z mínima. Tal ponto certamente é um vértice de conv(C). Consideremos, agora, a reta r, paralela ao eixo x e contendo p_1, e o semi-plano horizontal α limitado pela reta r e orientado na direção positiva do eixo y. Giramos este semi-plano no sentido de y para z até encontrar um segundo ponto p_2 de C. Nesta posição, α é um plano suporte de conv(C). Em geral, teremos então determinado uma aresta p_1p_2 de P. (É fácil lidar com os casos degenerados que podem ocorrer: se mais de um ponto de C for atingido por α nesta posição e se todos estiverem sobre a mesma semi-reta, basta tomar como p_2 o que estiver mais distante de p_1; se os pontos obtidos não estiverem sobre a mesma semi-reta, o fecho convexo destes pontos determina a face inicial.) Agora giramos o novo semi-plano α em torno da aresta p_1p_2 até encontrar um ponto p_3 de C, que determina a face inicial desejada.

É interessante observar que o método descrito acima para a obtenção da face inicial pode ser implementado empregando o passo de "embrulho-para-presente" para faces provisórias obtidas através da introdução de vértices artificiais. O leitor pode verificar que obtenção do ponto p_2 a partir de p_1 pode ser feita chamando o passo de "embrulho-para-presente" para a face F em que os vértices são, nesta ordem, p_1, $p_1 + (1,0,0)$ e $p_1 + (0,-1,0)$ e a aresta a é definida por p_1 e $p_1 + (1,0,0)$. Uma vez obtido p_2, o ponto p_3 é obtido empregando o mesmo passo para a face de vértices p_2, p_1 e $p_1 + (-1,0,0)$ e para a aresta p_2p_1.

Estamos prontos, então, para escrever o algoritmo. Vamos utilizar uma estrutura do tipo *winged-edge* para armazenar conv(C). A rigor, isto não é necessário para que o algoritmo possa produzir a lista de todas as faces de conv(C). Na realidade, basta armazenar a lista de faces e arestas já criadas (em cada caso, como listas de vértices). No entanto, a estrutura *winged-edge* fornece uma forma natural de fazer esta armazenagem, com a vantagem de que, ao final do processo, uma descrição topologicamente completa da fronteira de conv(C) estará disponível.

Quando uma nova face F de P é gerada pelo passo "embrulho-para-presente", suas arestas são examinadas para verificar se elas pertencem a uma face já criada. Neste caso, a informação correspondente àquela aresta fica completamente conhecida (já que as duas faces a ela adjacentes já foram determinadas). Caso contrário, dizemos que a aresta é **livre**. Cada face recém-criada é colocada em uma fila $\mathcal{F}$. A cada passo do algoritmo, uma face é retirada da fila e a função "embrulho-para-presente" é aplicada para cada uma de suas faces livres. O processo termina quando a fila $\mathcal{F}$ fica vazia. O algoritmo é [CK]:

Algoritmo 3.6: Embrulho-para-Presente para FC3D

```
1. Inicialização:
   Obtenha uma face inicial F para conv(C) aplicando a técnica descrita
   acima. Coloque F em 𝓕 e na estrutura winged-edge, com todas as suas
   arestas livres.
2. Enquanto 𝓕 ≠ ∅ repita
      remova uma face F de 𝓕;
      para cada aresta livre de F
            determine a face adjacente F', usando "embrulho-para-presente";
            coloque F' assim gerada na fila 𝓕;
            coloque F' na estrutura winged-edge, conectando-a com as faces
            já geradas que lhe são adjacentes e determinando suas arestas
            livres.
```

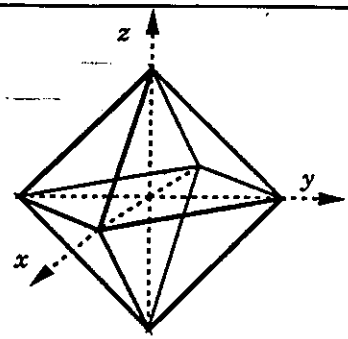
O algoritmo acima pára somente quando todas as faces de conv(C) tiverem sido erminadas. Para analisar sua complexidade, observamos que os passos 5 e 7 dominam a plexidade do algoritmo. O passo "embrulho-para-presente", executado em 5, é executado náximo uma vez para cada aresta de conv(C). Como cada execução demanda um tempo) e o número de arestas também é $O(n)$, o trabalho total executado em 5 é $O(n^2)$. (Estamos

omitindo o tempo necessário para a obtenção dos fechos convexos bidimensionais a serem executados caso a função "embrulho-para-presente" encontre mais de um vértice determinando a próxima face. Mas o trabalho total envolvido na determinação destes fechos convexos é $O(n \log n)$, o que não altera nossa análise.) Cada vez que uma face é criada, é necessário, no passo 7, percorrer a estrutura de dados para verificar se suas arestas já foram criadas. Para testar a já ocorrência de uma aresta, é necessário tempo linear (já que o número de arestas é $O(n)$). Como cada aresta é gerada exatamente duas vezes, o tempo total dispendido analisando a já existência de arestas é também $O(n^2)$. Logo

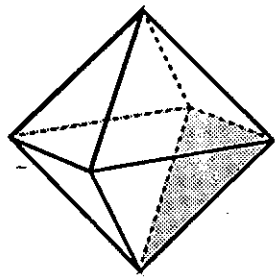
Teorema 3.5: *O algoritmo "embrulho-para-presente" corretamente determina o fecho convexo de um conjunto de n pontos do* $\mathbf{R}^3$ *em tempo* $O(n^2)$. ■

Na figura 3.14, indicamos diversos passos na obtenção do fecho convexo do conjunto de pontos C = {(1,0,0), (–1,0,0), (0,1,0), (0,–1,0), (0,0,1), (0,0,–1)} através do algoritmo de "embrulho-para-presente" (C é o conjunto de vértices de um octaedro regular).

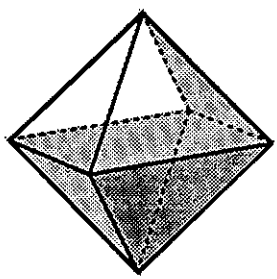
Na descrição do algoritmo "embrulho-para-presente" acima, recorremos a um algoritmo para FC2D para lidar com o caso em que o plano da nova face a ser gerada contém 2 ou mais pontos de C (além dos vértices da aresta que lhe dá origem). Há uma outra estratégia a ser adotada neste caso: escolhe-se qualquer destes pontos (desde que determine um triângulo não degenerado com a aresta em questão) e gera-se a face triangular correspondente. O resultado é que, ao final do processo, o poliedro formado possivelmente possuirá pares de faces contíguas coplanares. Basta, então, percorrer cada aresta do poliedro e aglutinar as faces que lhe são adjacentes, caso elas sejam coplanares. O trabalho total envolvido nesta tarefa é $O(n)$ e não afeta a complexidade do algoritmo. Deve-se frisar que, em muitas aplicações, este passo final é desnecessário ou envolve a aglutinação de um número muito pequeno de faces. Por exemplo, caso os pontos de C sejam gerados independentemente de acordo com alguma distribuição contínua de probabilidade, a probabilidade de que quatro ou mais pontos sejam coplanares é igual a zero.



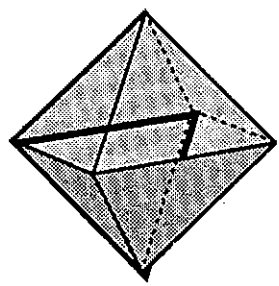
(a) os pontos dados são os vértices de um octaedro regular.



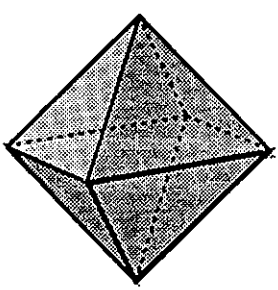
(b) a primeira face é encontrada pelo algoritmo.



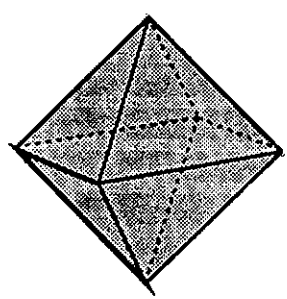
(c) as três faces vizinhas à primeira são acrescentadas pelo algoritmo.



(d) o algoritmo prossegue, determinando as duas novas vizinhas da face assinalada.



(e) a penúltima face é gerada, vizinha à face assinalada.



(f) o algoritmo gera a última face do octaedro, ao explorar a face assinalada.

Figura 3.14 – O algoritmo "embrulho-para-presente".

Algoritmo de Preparata e Hong

Se a utilização de uma estrutura topológica para armazenar conv(C) não é essencial para se obter um algoritmo $O(n^2)$ para FC3D, ela parece ser essencial para um algoritmo capaz de funcionar em tempo $O(n \log n)$.

O algoritmo de Preparata e Hong é um algoritmo do tipo dividir-para-conquistar para FC3D. A estratégia é semelhante à empregada para o algoritmo Mergehull para FC2D, isto é, segue a seguinte linha geral:

Algoritmo 3.7: Mergehull3D(C)

1. Se C possui k ou menos elementos, determine conv(C) diretamente (k é uma constante arbitrária; por exemplo, podemos tomar k=4).
2. Senão divida o conjunto C em conjuntos C_1 e C_2, cada um com metade dos elementos.
3. Recursivamente aplique Mergehull3D(C_1) e Mergehull3D(C_2) para obter conv(C_1) e conv(C_2).
4. Combine os resultados obtidos em (3) de modo a obter conv(C).

No caso de Mergehull para FC2D, o algoritmo de Graham nos permitiu não lidar diretamente com o problema de combinar conv(C_1) e conv(C_2). Apenas usamos a informação dada por conv(C_1) e conv(C_2) a respeito da ordem polar dos vértices de cada polígono em torno de um ponto interior para acelerar a primeira fase do algoritmo de Graham. Infelizmente, este recurso não é generalizável para três dimensões.

Alternativamente, poderíamos ter empregado um método que usa de modo mais direto as estruturas de conv(C_1) e conv(C_2) para obter conv(C). Para tal, incluiremos a hipótese adicional de que P_1 = conv(C_1) e P_2 = conv(C_2) sejam disjuntos. (Isto pode ser garantido em Mergehull se o conjunto C é preprocessado de modo que seus elementos estejam lexicograficamente ordenados[3] com respeito às suas coordenadas x e y e cada etapa de subdivisão é feita com base nesta ordenação.) Então conv(C) é da forma $u_p u_{p+1} \ldots u_q v_r v_{r+1} \ldots v_s$, onde os u_i's são vértices consecutivos de P_1, os v_j's são vértices

---

[3] A ordem lexicográfica é uma ordem completa $<$ em $\mathbf{R}^n$ definida do seguinte modo:
$(x_1, x_2, \ldots, x_n) < (y_1, y_2, \ldots, y_n)$ se e só se existe i, $1 \leq i \leq n$ tal que $x_i < y_i$ e $x_j = y_j$ para $1 \leq j < 1$ (isto é, se na primeira coordenada i em que x e y diferem tem-se $x_i < y_i$).

consecutivos de P_2 e os segmentos $u_q v_r$ e $v_s u_p$ determinam as retas de suporte comum a P_1 e P_2 (figura 3.15).

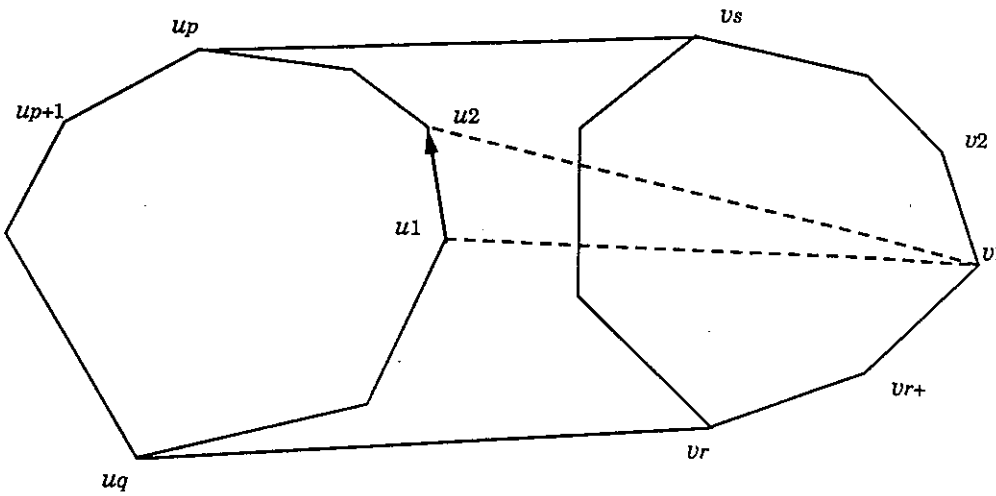


Figura 3.15 – Determinando as retas de suporte comum.

Preparata e Hong mostraram que os pontos u_p, u_q, v_r e v_s podem ser obtidos em tempo linear [PH]. Para tal, suponhamos (sem perda de generalidade) que $x(u_i) < x(v_j)$, para quaisquer vértices u_i e v_j de P_1 e P_2 e que u_1 e v_1 são os vértices de abscissa máxima em P_1 e P_2. Consideremos os lados u_1u_2 e v_1v_2. Se cada um destes lados está abaixo da reta u_1v_1, então u_1v_1 é necessariamente a reta de suporte comum superior aos dois polígonos. Senão, avançamos em um polígono que viola esta condição (figura 3.15). No exercício 12, pedimos que o leitor transforme esta idéia em um algoritmo e que verifique que o mesmo obtém a reta de suporte comum superior a P_1 e P_2 em tempo linear no número total de vértices de P_1 e P_2.

O algoritmo de Preparata e Hong para FC3D generaliza este argumento para o caso tridimensional (bem mais complexo). A subdivisão de C em C_1 e C_2 é feita com base na ordem lexicográfica dos pontos e o passo crucial é a obtenção do fecho convexo da união de dois poliedros tridimensionais disjuntos. Analogamente ao caso bidimensional, a fronteira de conv(C) é constituída por uma porção conexa da fronteira de conv(C_1), uma porção conexa da

fronteira de conv(C_2) e faces adicionais, que estão contidas em planos de suporte comum a conv(C_1) e conv(C_2). A obtenção destas faces pode ser obtida pelo processo de "embrulho-para-presente" descrito anteriormente, mas isto resultaria em um tempo de até $O(n^2)$ para reunir conv(C_1) e conv(C_2). No algoritmo de Preparata e Hong esta complexidade é reduzida para $O(n)$, com auxílio de testes que permitem percorrer as estruturas *winged-edge* que armazenam conv(C_1) e conv(C_2) sem jamais precisar recuar. Omitimos os detalhes e o leitor deve consultar [PS] ou [PH]. No próximo capítulo descreveremos um algoritmo similar ao de Preparata e Hong, para a obtenção da triangulação de Delaunay de um conjunto de pontos (na realidade, o algoritmo que obtém a triangulação pode ser obtido como uma consequência do algoritmo de Preparata e Hong; preferimos descrevê-lo com mais detalhes por ser mais simples entender seu comportamento, devido a operar no plano).

Exercícios

1. O fecho convexo de um subconjunto qualquer C do $\mathbf{R}^n$ é definido como o menor conjunto convexo contendo C. Escreva precisamente o que isto quer dizer e mostre que o fecho convexo existe e coincide com o conjunto de combinações convexas (finitas) de elementos de C.

2. Seja C um conjunto finito de pontos do $\mathbf{R}^n$ e seja C' o conjunto de pontos extremos de conv(C). Mostre que conv(C') = conv(C).

3. Seja $P = p_1p_2...p_n$ um polígono simples, orientado no sentido anti-horário. Suponha que o ângulo orientado de $p_{i-1}p_i$ para p_ip_{i+1} é negativo. Mostre que p_i não é vértice de conv(P). A recíproca é verdadeira?

4. Mostre que o algoritmo Quickhull pode precisar de um número quadrático de passos para obter o fecho convexo de um conjunto de pontos.

5. Suponha que $P = p_1p_2...p_n$ é um polígono convexo. Considere o problema de localizar um ponto p em relação ao polígono P (isto é, de determinar se p é interior, exterior ou pertence à fronteira de P).

a) Mostre que o problema pode ser resolvido em tempo $O(\log n)$, caso os vértices de P estejam armazenados em um vetor.

b) E se os vértices estão armazenados segundo uma lista encadeada?

6. a) Mostre que o fecho convexo de um conjunto de pontos do plano é o polígono simples de menor área que contém o conjunto.

b) Mostre que o fecho convexo de um conjunto de pontos do plano é o polígono simples de menor perímetro que contém o conjunto.

c) Os resultados acima valem para outras dimensões?

7. a) Mostre como construir um polígono simples passando por um conjunto de pontos no plano. [Sugestão: reveja a descrição do algoritmo de Graham.]

b) Considere as linhas poligonais fechadas com vértices em um conjunto fixo de pontos do plano. Mostre que as poligonais de comprimento mínimo são simples.

8. O que acontece quando se aplica o algoritmo de Graham a polígonos simples não necessariamente estrelados?

9. A **profundidade convexa** de um ponto em relação a um conjunto é definida da seguinte forma:

– os pontos extremos tem profundidade 0;

– um ponto tem profundidade p se ele é ponto extremo do conjunto restante após a retirada dos pontos de profundidade 0, 1, ..., p–1.

Adapte o algoritmo de Jarvis para listar os pontos de um conjunto em ordem crescente de profundidade convexa. Este algoritmo é ótimo?

10. Sejam $p_1, p_2, ..., p_n$ pontos do plano. Descreva um algoritmo linear que, determina se p_1 é um vértice do fecho convexo de $\{p_1, p_2, ..., p_n\}$.

[Sugestão: P_1 é vértice de conv$\{p_1, p_2, ..., p_n\}$ se e só se existe uma reta r contendo p_1 tal que $p_2, ..., p_n$ estejam no mesmo semiplano determinado por r.]

11. Considere a estrutura *winged-edge* apresentada na seção 3.4.

a) Escreva um algoritmo que, dada uma face F, obtém seus vértices em tempo linear.

b) Escreva um algoritmo que, dado um vértice v, obtém todos os vértices adjacentes a v em tempo linear.

12. Sejam P_1 e P_2 dois polígonos convexos tais que os vértices de P_1 têm abscissas menores que as abscissas de P_2. Seguindo as idéias da seção 3.4, escreva um algoritmo que obtenha sua linha de suporte comum inferior (superior). Mostre que seu algoritmo é linear no número total de vértices dos polígonos.

4

Triangulações

Neste capítulo estudamos como dar a um conjunto de vetores $\{x_1, x_2, ..., x_n\}$ do R^d uma estrutura análoga à obtida ao ordenarmos um conjunto de números reais. Na realidade, trataremos apenas do caso bidimensional, mas a maior parte das idéias deste capítulo podem ser generalizadas para dimensões superiores.

4.1 Introdução

O tipo de estrutura que pretendemos generalizar é o que permite aproximar uma função dada por uma série de observações esparsas. Sejam $x_1, x_2, ..., x_n$ números reais e vamos supor que os valores de uma certa função f são conhecidos nestes pontos. A partir desta informação, queremos estender f ao intervalo $[m, M]$, onde m e M são, respectivamente, o mínimo e o máximo dos x_i's.

Triangulações

O primeiro caso que consideramos é aquele em que os valores tomados por f são expressos por números reais. Podemos, por exemplo, supor que $f(x_i)$ exprime a temperatura observada no ponto x_i. Neste caso, é usual recorrermos a um esquema de interpolação para estender f ao intervalo $[m, M]$. O esquema de interpolação mais simples é **interpolação linear**, que estende f para uma função F, linear por partes, cujo valor em cada ponto x_i é igual a $f(x_i)$. Para aplicar este método de interpolação, os valores $x_1, x_2, ..., x_n$ são ordenados, de modo a delimitar os intervalos sobre os quais F será dado por uma função do primeiro grau (figura 4.1). Então, representando por $x_{(i)}$, $i = 1, ..., n$, os valores ordenados dos x_i's, a obtenção de $F(x)$ para $x \in [m, M]$ envolve determinar qual dos intervalos $[x_{(i)}, x_{(i+1)}]$ contém x, exprimir x na forma $\lambda x_{(i)} + (1-\lambda)x_{(i+1)}$ (onde $0 \leq \lambda \leq 1$), e então calcular $F(x) = \lambda f(x_{(i)}) + (1-\lambda)f(x_{(i+1)})$.

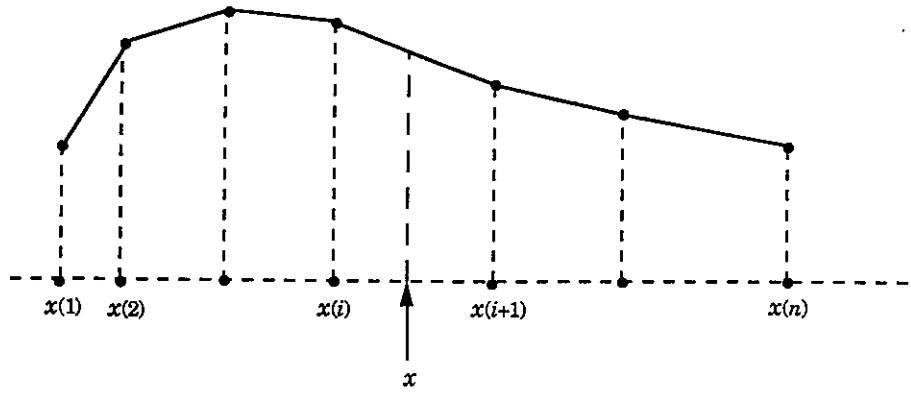


Figura 4.1 – Ordenando para interpolar em $\mathbf{R}^1$.

Consideremos agora o problema análogo no $\mathbf{R}^2$. Isto é, dado um conjunto $C = \{x_1, x_2, ..., x_n\}$ de pontos de $\mathbf{R}^2$ e valores reais $f(x_i)$, $i = 1,..., n$, obter uma função F, linear por partes, definida sobre um domínio adequado D, e tal que $F(x_i) = f(x_i)$, $i = 1,..., n$. Há uma escolha natural para o domínio D: o fecho convexo de C. No entanto, dado um ponto $x \in \text{conv}(C)$, não é tão óbvio como calcular $F(x)$. Mantendo a analogia com o caso anterior, a idéia é exprimir conv(C) como a união de um conjunto de triângulos que formem um **complexo simplicial** de dimensão 2. Isto é: tais que a interseção de dois triângulos não disjuntos seja um vértice ou um lado comum a eles (figura 4.2). Uma vez obtidos estes triângulos, dado $x \in \text{conv}(C)$, determinamos um triângulo $x_i x_j x_k$ contendo x, exprimimos $x = \lambda_1 x_i + \lambda_2 x_j + \lambda_3 x_k$ (onde os números não negativos λ_1, λ_2 e λ_3 são as coordenadas baricêntricas de x em relação ao triângulo $x_i x_j x_k$) e obtemos $F(x) = \lambda_1 f(x_i) + \lambda_2 f(x_j) + \lambda_3 f(x_k)$.

O passo fundamental no esquema de interpolação descrito acima consiste em resolver o seguinte problema:

TRIANGULAÇÃO: *Dado um conjunto C de pontos do plano, obter uma triangulação de conv(C) (isto é, um complexo simplicial cuja união seja conv(C)), cujo conjunto de vértices seja C.*[1]

[1] Para as aplicações, também é relevante o problema em que é permitido acrescentar novos pontos, conhecidos como **pontos de Steiner**, a conv(C).

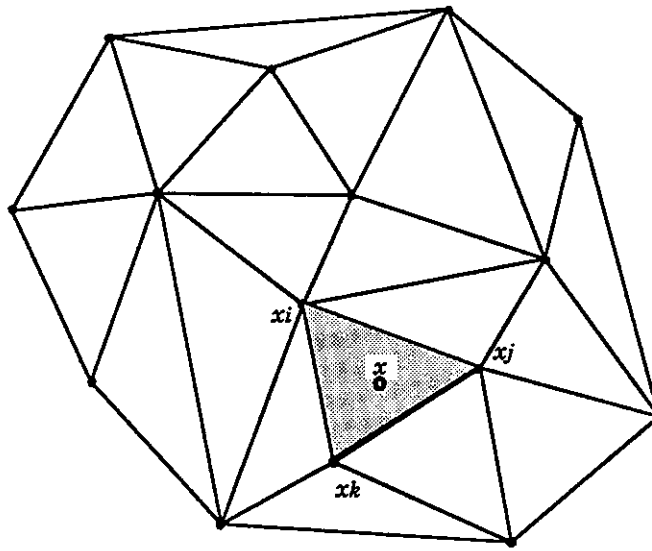


Figura 4.2 – Triangulando o fecho convexo de um conjunto de pontos do plano.

Ao contrário do caso unidimensional, não existe uma solução única para este problema. No entanto, é possível demonstrar algumas propriedades gerais sobre as triangulações de conv(C). Temos o seguinte:

Teorema 4.1: *Dado um conjunto $C = \{x_1, x_2, ..., x_n\}$ de pontos do plano, toda triangulação de conv(C) possui exatamente $(2n - v - 2)$ triângulos e $(3n - v - 3)$ arestas, onde v é o número de pontos de C que estão na fronteira de conv(C).*

Prova: Uma triangulação de conv(C) determina uma subdivisão do plano em T+1 faces, onde T é o número de triângulos na triangulação. Portanto, pela fórmula de Euler (seção 3.4) temos:

$$n + (T + 1) = a + 2,$$

onde a é o número de arestas.

As v arestas da fronteira de conv(C) são comuns a um triângulo e à face externa. Todas as demais figuram em exatamente dois triângulos. Logo, somando o número de arestas de todas as faces, processo no qual cada aresta é contada duas vezes, encontramos $3T + v = 2a$,

ou seja:

$$a = \frac{3T + v}{2}.$$

Substituindo e resolvendo para T, encontramos $T = 2n - v - 2$ e $a = 3n - v - 3$. ■

Uma consequência deste teorema é que o tamanho da solução para um problema de triangulação é linear no número de elementos de C.

Embora todas as triangulações de conv(C) tenham o mesmo número de triângulos, para certos problemas práticos é importante obter uma triangulação que apresente triângulos relativamente "gordos", isto é, que evite triângulos com ângulos muito pequenos. Uma das técnicas que examinaremos neste capítulo produz uma triangulação de conv(C) (a triangulação de Delaunay) que, dentre todas as triangulações de conv(C), maximiza o menor de todos os ângulos internos dos triângulos que compõem a triangulação [Si].

Diagrama de Voronoi

O outro caso de aproximação é aquele em f toma valores em um conjunto discreto, de modo que um processo de interpolação não faz sentido. Por exemplo, suponhamos que $f(x_i)$ indica o tipo de rocha (considerado como um elemento de um conjunto discreto) encontrado ao se perfurar um terreno no ponto x_i. Uma forma razoável de se estender esta informação a valores de x que não estejam em $C = \{x_1, x_2, ..., x_n\}$ é obter o ponto x_i de C que esteja mais próximo de x e tomar $F(x) = f(x_i)$.

No caso unidimensional, os conjuntos para os quais F é constante são intervalos delimitados pelos pontos médios dos intervalos obtidos ao se ordenar os x_i. No caso bidimensional, os pontos x para os quais $F(x) = f(x_i)$, onde x_i é um certo elemento de C, determinam uma região V_i. Na realidade, V_i é um polígono convexo e é chamado de **polígono de Voronoi** relativo a x_i. Observe que $x \in V_i$ se e somente se $d(x, x_i) \leq d(x, x_j)$, para todo $j \neq i$. Cada uma destas desigualdades representa o semiplano contendo x_i determinado pela mediatriz do segmento $x_i x_j$. Assim, V_i é um polígono convexo (possivelmente ilimitado), por ser a interseção de $n-1$ semiplanos.

Os polígonos de Voronoi relativos aos elementos de C decompõem o R^2 numa união de polígonos convexos de interiores disjuntos. Esta decomposição é chamada de **tesselação de**

Dirichlet do plano determinada por C ou simplesmente de **diagrama de Voronoi** de C e representado por Vor(C) (figura 4.3).

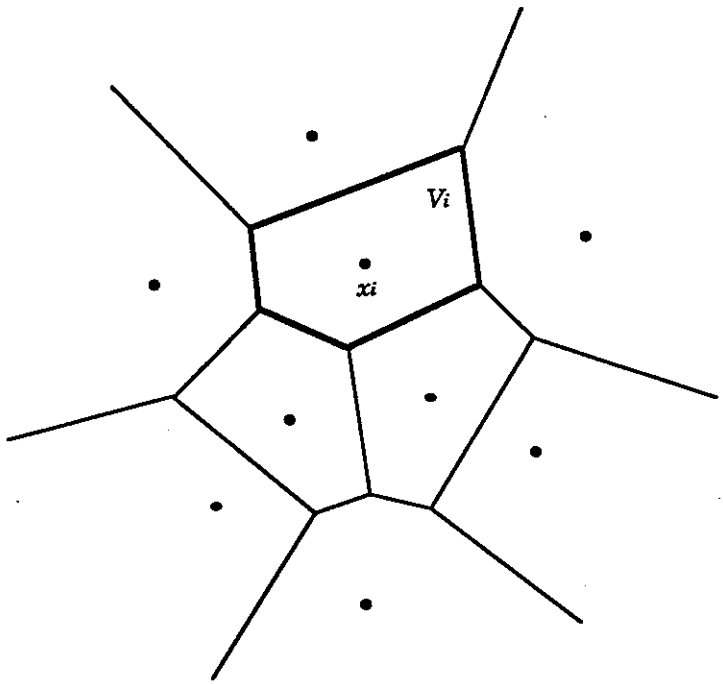


Figura 4.3 – Diagrama de Voronoi de um conjunto de pontos do plano.

Um dos problemas fundamentais que examinamos neste capítulo é a obtenção eficiente do diagrama de Voronoi. Isto é, a resolução do seguinte problema:

VORONOI: *Obter o diagrama de Voronoi de um conjunto C de pontos do plano.*

Não é difícil conceber algoritmos de força bruta para resolver os problemas mencionados acima. Uma triangulação de conv(C) pode ser obtida, por exemplo, da seguinte maneira. Começamos por obter conv(C) e por traçar as diagonais que partem de um vértice (x_1, por exemplo). A seguir, cada ponto que não esteja na fronteira de conv(C) é "lançado" em conv(C) e o triângulo em que ele "cai" é subdividido. Este algoritmo requer tempo $O(n \log n)$ para obter

conv(C) e a triangulação inicial por diagonais. A inclusão de cada ponto x_i requer tempo proporcional ao número de triângulos que, como vimos, é O(n). O algoritmo completo é, pois, O(n^2).

Para obter o diagrama de Voronoi de C, podemos obter cada polígono de Voronoi separadamente. A obtenção de V_i consiste em encontrar a interseção de n semiplanos. É fácil de ver que tal interseção pode ser feita em tempo O(n^2), acrescentando um semiplano de cada vez (no entanto, há um algoritmo O(n log n) para este problema; veja o capítulo 7 de [PS]). Logo, os n polígonos de Voronoi podem ser obtidos em tempo O(n^3) (ou O(n^2 log n), se a interseção de n semiespaços for feita em tempo O(n log n)).

Como veremos a seguir, o diagrama de Voronoi de C está intimamente relacionado à triangulação de Delaunay, que mencionamos na subseção anterior. Eles são realizações no plano de grafos planares que são duais um em relação ao outro. A idéia central deste capítulo é exatamente a relação entre estas duas estruturas. Explorando esta relação, poderemos obter algoritmos O(n log n) para cada um destes problemas e mostrar que tais algoritmos são ótimos.

4.2 Propriedades do diagrama de Voronoi

Para reduzir a complexidade do problema de obter o diagrama de Voronoi de um conjunto C é necessário encará-lo como a realização no plano de um grafo planar e obter propriedades relativas a este grafo planar e a esta sua realização.

A primeira observação é a que cada aresta do grafo representa um conjunto de pontos comum a dois polígonos de Voronoi V_i e V_j e, portanto, equidistantes de dois pontos x_i e x_j de C. Logo, cada aresta do grafo é um segmento de uma mediatriz definida por dois pontos de C. Em consequência, os vértices do diagrama de Voronoi são pontos de encontro de mediatrizes de segmentos determinados por pares de pontos de C. A partir daí, é possível obter os seguintes resultados:

Teorema 4.2: *Os polígonos de Voronoi correspondentes a um par de pontos x_i e x_j de C possuem uma aresta comum se e somente se existe um círculo contendo x_i e x_j e tal que todos os demais pontos sejam exteriores a este círculo.*

Prova: Dois polígonos de Voronoi V_i e V_j possuem uma aresta comum se e só se existem pontos (exatamente os pontos pertencentes a esta aresta comum) que são equidistantes dos

pontos correspondentes x_i e x_j e mais próximos deles que de qualquer outro ponto de C. Ou seja, V_i e V_j possuem aresta comum se e só se existe um círculo (tendo centro no interior da aresta comum) que contém x_i e x_j e exclui todos os demais elementos de C. ■

Teorema 4.3: *Um polígono de Voronoi V_i é ilimitado se e só se o ponto correspondente x_i pertence à fronteira de conv(C).*

Prova: Suponhamos que x_i não é da fronteira de conv(C). Então existem pontos x_1, x_2 e x_3 em C tais que x_i seja interior ao triângulo $x_1x_2x_3$. Consideremos os círculos K_{12}, K_{23} e K_{31}, circunscritos respectivamente aos triângulos $x_ix_1x_2$, $x_ix_2x_3$ e $x_ix_3x_1$ e tomemos um círculo K contendo K_{12}, K_{23} e K_{31} (figura 4.4). Seja p um ponto exterior a K e tomemos o segmento px_i, que intercepta um dos círculos K_{12}, K_{23} e K_{31} (digamos K_{12}) em um ponto u de seu arco externo. Como a corda ux_i de K_{12} está entre as cordas ux_1 e ux_2, necessariamente tem-se que pelo menos uma destas duas últimas cordas (digamos ux_1) é menor que ux_i (exercício 1).

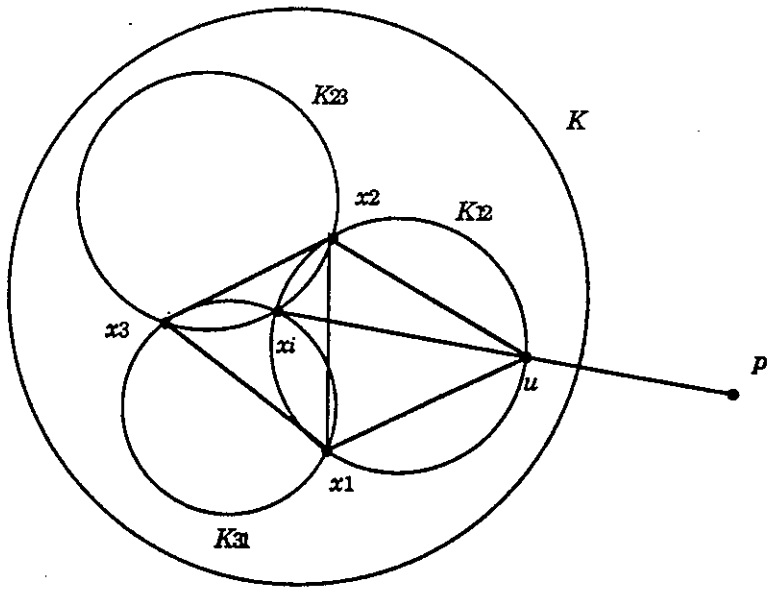


Figura 4.4 – x_i interior a conv(C) $\Rightarrow$ V_i limitado.

Logo, temos

$$px_i = pu + ux_i < pu + ux_1 \leq px_1,$$

o que mostra que p não pertence a V_i. Logo, V_i está contido em K e é, portanto, limitado.

Reciprocamente, se x_i é da fronteira de conv(C), então existe uma reta suporte L de conv(C) passando por x_i. Considere a semi-reta perpendicular a L, orientada para fora de conv(C), e tome um ponto p qualquer sobre ela. Como todo ponto de C está além de L em relação a p, concluímos que p está mais próximo de x_i que de qualquer outro ponto de C; isto é, $p \in V_i$ e, portanto, V_i é ilimitado. ■

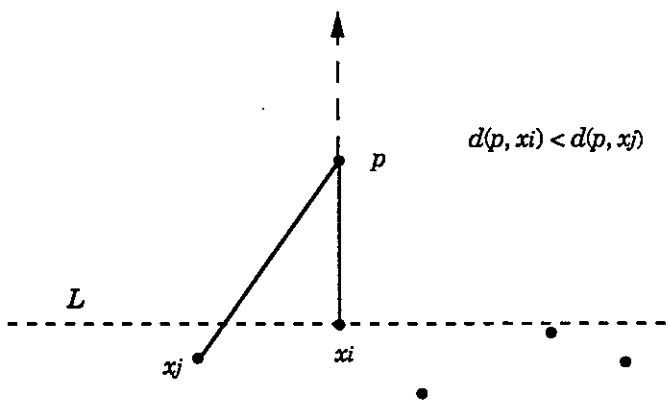


Figura 4.5 – x_i na fronteira de conv(C) $\Rightarrow$ V_i ilimitado.

Teorema 4.4: *Todo vértice v de Vor(C) é comum a pelo menos três polígonos de Voronoi e é centro de um círculo $C(v)$ definido pelos pontos de C correspondentes aos polígonos que se encontram em v. Além disso, $C(v)$ não contém nenhum outro ponto de C.*

Prova: Primeiro, vamos mostrar que no mínimo três polígonos de Voronoi se encontram em cada vértice v. Suponhamos que v fosse comum a apenas dois polígonos de Voronoi V_i e V_j. Então v seria a interseção de apenas duas arestas a_1 e a_2. Cada uma delas estaria contida na mesma mediatriz (a determinada por x_i e x_j) e portanto não se interceptariam em um único ponto. Logo v deve ser comum a pelo menos três polígonos de Voronoi.

Seja então C' o conjunto de pontos de C correspondentes a estes polígonos. O vértice v é equidistante dos pontos de C'. Logo, estes pontos determinam um círculo de centro v. Além disso, nenhum outro ponto pode estar contido em $C(v)$, já que, se algum outro ponto $x_k \in C \setminus C'$ estivesse contido em $C(v)$, então v estaria pelo menos tão próximo de x_k do que dos elementos de C', o que contradiz o fato de v ser comum apenas aos polígonos de Voronoi correspondentes a pontos de C'. ■

De acordo com o Teorema acima, uma possível estratégia para a obtenção eficiente do diagrama de Voronoi de C consiste em obter os subconjuntos $T_1, T_2, ..., T_t$ de C que determinam "círculos vazios". Isto é, cada T_k é formado por três ou mais pontos cocirculares de C tais que o círculo correspondente $C(T_k)$ não contém nenhum outro elemento de C. Um caso particular importante é aquele em que os pontos de C são tais que quatro pontos quaisquer não são cocirculares. Neste caso, cada um dos conjuntos T_k acima descritos contém exatamente três elementos de C.

Note que as arestas de Vor(C) são exatamente os segmentos de mediatrizes correspondentes a pontos consecutivos em algum dos T_k. Portanto, uma vez conhecidos os conjuntos T_k, o diagrama de Voronoi pode ser obtido em tempo linear no seu número de arestas. Veremos, mais tarde, que, de fato, esta estratégia pode ser usada para obter um algoritmo ótimo.

O teorema a seguir estabelece a ligação fundamental entre os problemas VORONOI e TRIANGULAÇÃO. Dada uma realização de um grafo planar G, o **dual** de G (com relação a esta realização) é o grafo em que os vértices correspondem a faces determinadas pela realização de G e tal que as arestas são determinadas por faces adjacentes nesta realização.

No diagrama de Voronoi, cada elemento de C está associado a uma face de Vor(C). O grafo dual de Vor(C) tem por vértices os elementos de C e por arestas os pares de elementos de C cujos polígonos de Voronoi são vizinhos; isto é, que são consecutivos em algum T_k. Considere agora o diagrama obtido representando estas arestas pelos segmentos de reta que ligam os elementos respectivos de C (figura 4.6). Tal diagrama será chamado de **diagrama de Delaunay** de C e denotado por Del(C). Note que dois pontos x_i e x_j de C determinam uma aresta de Del(C) se e só se existe um círculo contendo x_i e x_j tal que todos os demais pontos de C sejam exteriores a este círculo.

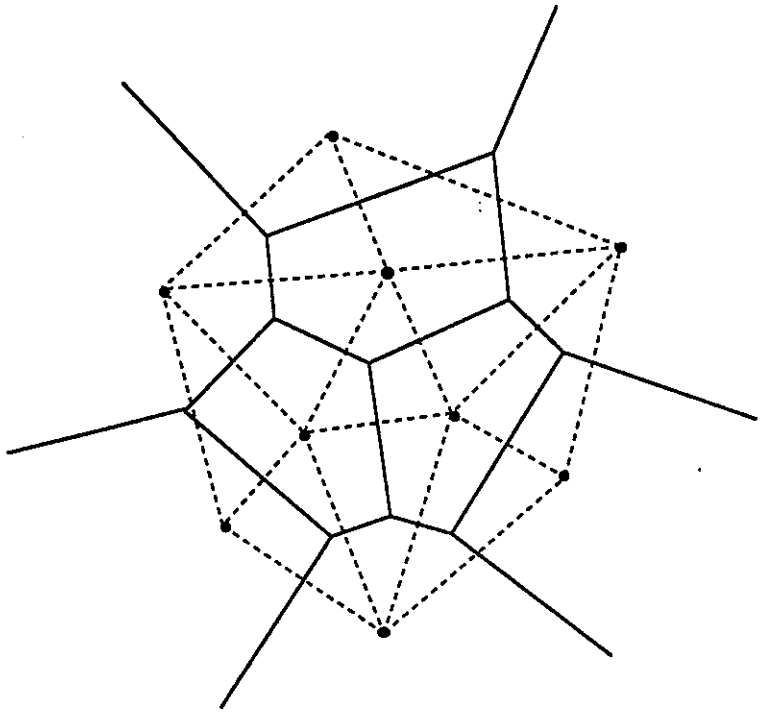


Figura 4.6 – Os diagramas de Voronoi (—) e Delaunay (•••) de um conjunto C.

O grafo dual de Vor(C) é planar (duais correspondentes a realizações de grafos planares são sempre planares). Mas não é nada óbvio que a construção acima forneça uma realização deste dual. O teorema a seguir garante que isto é sempre verdade.

Teorema 4.5: *Seja $C = \{x_1, x_2, ..., x_n\}$ um conjunto de pontos do plano e seja $\{T_k\}$ a família de subconjuntos de C que determinam círculos vazios. Suponhamos que os vértices de cada T_k sejam ordenado segundo este círculo. Então:*

a) o diagrama de Delaunay, obtido ligando os pontos consecutivos em algum T_k, é uma realização de um grafo planar (portanto, é uma realização do grafo dual de Vor(C)).

b) as arestas correspondentes a cada T_k delimitam uma região convexa R_k ; estas regiões possuem interiores disjuntos dois a dois e sua união é o fecho convexo de C.

c) as regiões R_k são exatamente as faces limitadas do diagrama planar determinado por Del(C).

d) finalmente, se C satisfaz a condição de quatro pontos quaisquer nunca serem cocirculares, então as regiões R_k determinam uma triangulação de conv(C), conhecida como a triangulação de Delaunay determinada por C.

Prova:

a) Basta provar que as arestas deste diagrama não se interceptam a não ser nos vértices. Para tal, demonstraremos primeiro o seguinte lema:

Lema 4.1: *Sejam pq e rs segmentos do plano que se interceptam em o. Então, para existir um círculo passando por p e q e tal que r e s sejam exteriores ao círculo, é necessário e suficiente que os ângulos do quadrilátero prqs sejam tais que $p + q > \pi$ (equivalentemente $r + s < \pi$).*

Prova do Lema: Suponhamos que exista tal círculo (figura 4.7). Sejam r' e s' as interseções de rs com o círculo. Então $r + s < r' + s' = \pi$ (já que o quadrilátero $pr'qs'$ é inscritível em um círculo). Reciprocamente, se $r + s < \pi$, existem pontos r' e s' tomados sobre o segmento rs de tal modo que $r' + s' = \pi$. Estes pontos, juntamente com p e q determinam um círculo que exclui r e s. ■

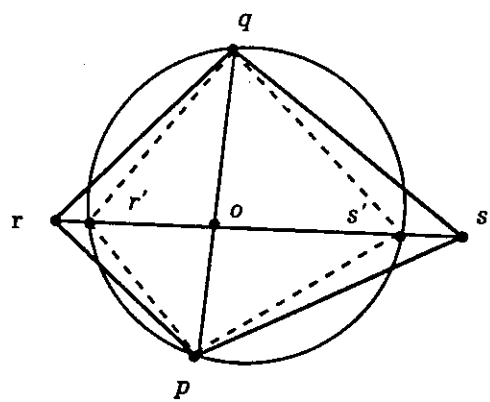


Figura 4.7 – Condição de existência de círculo por p e q excluindo r e s.

Suponhamos então que pq e rs sejam duas arestas de Del(C), que se interceptam em um ponto o. Como V_p e V_q possuem uma aresta comum, existe um círculo contendo p e q e tal que r e s são exteriores a ele. Mas, pelo Lema, isto indica que, no quadrilátero $prqs$, $r + s < \pi$. Ainda pelo Lema, isto implica que não existe círculo por r e s que exclua p e q. Portanto, V_r e V_s não possuem aresta comum, o que contradiz o fato de que $rs \in$ Del(C).

b) As arestas correspondentes a cada T_k são lados de um polígono inscrito em um círculo e, portanto, determinam uma região convexa R_k. Como o círculo contendo os elementos de T_k não contém nenhum outro ponto de C e as arestas de Del(C) não se cruzam a não ser nos vértices, concluímos que a única forma pela qual o interior de R_k poderia interceptar o interior de uma outra região seria se dois vértices não consecutivos p e q de T_k determinassem uma aresta de Del(C) (figura 4.8). Mas, se isto ocorresse, haveria um círculo contendo p e q e excluindo os demais elementos de T_k. Sejam r e s elementos de T_k que separam p de q. Como os ângulos no quadrilátero $prqs$ satisfazem $p + q = \pi$, pelo Lema 4.1 tal círculo não pode existir e, portanto, vértices não consecutivos de algum T_k não determinam arestas de Del(C). Logo, as regiões R_k têm interiores disjuntos dois a dois.

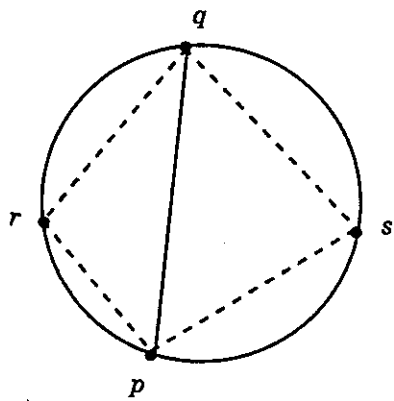


Figura 4.8 – Vértices não consecutivos em algum T_k não definem arestas de Del(C).

Falta ainda mostrar que a união das regiões R_k é igual a conv(C). Note, primeiro, que as regiões R_k são regiões convexas delimitadas por segmentos contidos em conv(C), o que implica que a união destas regiões certamente está contida em conv(C).

Por outro lado, os segmentos x_ix_j determinados por pontos consecutivos da fronteira de conv(C) certamente estão em Del(C), já que certamente existe um círculo contendo x_i e x_j que exclui os demais pontos de C (basta tomar como centro um ponto suficientemente distante sobre a mediatriz de p_ip_j). Em segundo lugar, observe que uma aresta qualquer $a = x_ix_j \in$ Del(C) delimita uma ou duas regiões R_k. Se a fizer parte da fronteira de conv(C), existe exatamente uma região R_k adjacente a a; esta região é definida por x_i, x_j e pelos pontos x_k tais que o ângulo $\angle(x_ix_kx_j)$ seja máximo (note que o círculo contendo tais pontos certamente é "vazio"). Se a não fizer parte da fronteira de conv(C), haverá duas regiões adjacentes a a, uma em cada semiplano, novamente determinadas pelos pontos determinando ângulos máximos com a. (Esta idéia será explorada em um dos algoritmos para obtenção de Del(C), a ser descrito na seção 4.4.)

Seja então x um ponto arbitrário de conv(C). Se x pertencer a alguma aresta de Del(C), então trivialmente x pertence a alguma região R_k. Caso contrário, tomemos por x uma semi-reta qualquer L que não contenha nenhum ponto de C. Como $x \in$ conv(C), necessariamente L intercepta pelo menos uma aresta de Del(C). Seja a a *primeira* aresta interceptada por L e seja R_k a região adjacente a a no mesmo semiplano de x (figura 4.9). Afirmamos, então, que $x \in R_k$. De fato, se x não pertencesse a R_k, então certamente a semi-reta L deveria interceptar alguma outra aresta de R_k. Este ponto de interseção seria necessariamente anterior ao ponto de interseção com a, o que contradiz o fato de a ser a primeira aresta interceptada. Deste modo, provamos que as regiões R_k cobrem conv(C) (e que, portanto, sua união é conv(C)).

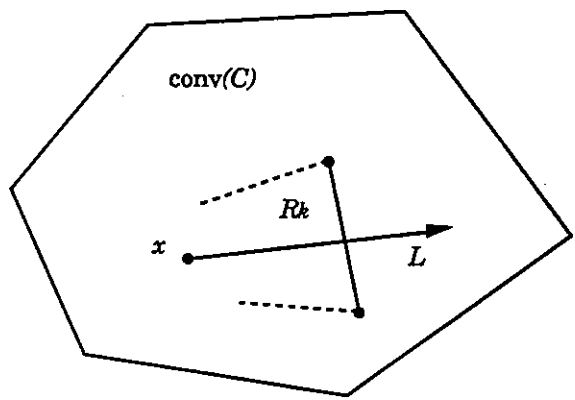


Figura 4.9 – Mostrando que as regiões de Delaunay cobrem conv(C).

c) Como cada região R_k é delimitada por arestas de Del(C) e seu interior não intercepta outras arestas de Del(C), certamente podemos afirmar que cada R_k é uma das regiões limitadas determinadas por Del(C). Por outro lado, a união de todas as regiões R_k é conv(C); mas a união de todas as faces limitadas determinadas por Del(C) está certamente contida em conv(C), já que cada uma de suas arestas está contida em conv(C). Logo não é possível que alguma das faces determinadas por Del(C) não corresponda a alguma região R_k.

d) No caso do conjunto C satisfazer a condição de que 4 pontos nunca são cocirculares, cada uma das regiões R_k é um triângulo e portanto Del(C) determina uma triangulação de conv(C).■

Assim, no caso do conjunto C satisfazer a restrição de que 4 pontos nunca são cocirculares, o diagrama de Delaunay é necessariamente uma triangulação de conv(C).[2] De qualquer modo, é sempre possível transformar Del(C) em uma triangulação, bastando para isto acrescentar diagonais às regiões (convexas) de Del(C). Qualquer uma destas triangulações será chamada de **triangulação de Delaunay** de C.

Utilizando o resultado do Teorema 4.1, que conta o número de triângulos e arestas em uma triangulação de conv(C), é possível estabelecer o seguinte:

Teorema 4.6: *O diagrama de Voronoi de* $C = \{x_1, x_2, ..., x_n\}$ *tem no máximo* $2n - 5$ *vértices e* $3n - 6$ *arestas.*

Prova: As arestas de Vor(C) correspondem a arestas de Del(C). O maior número possível de arestas de Del(C) ocorre quando todas as suas faces são triângulos e, além disso, conv(C) também é um triângulo. Neste caso, pelo Teorema 4.1 (com $v = 3$), temos que o número de arestas é $a = 3n - 3 - 3 = 3n - 6$. O número máximo de vértices é igual ao número máximo de triângulos em Del(C), que, ainda pelo Teorema 4.1 , é dado por $v = 2n - 2 - 3 = 2n - 5$. ■

4.3 Cotas inferiores

Uma das consequências da relação estabelecida no teorema 4.6 é que os problemas de obter o diagrama de Voronoi e o diagrama de Delaunay de um conjunto C são redutíveis um ao outro em tempo linear. Isto é, uma vez obtido um algoritmo para qualquer um dos dois

[2] Esta condição é suficiente mas não necessária para Del(C) determinar uma triangulação. Pode-se ter mais de 4 pontos no mesmo círculo, desde que o raio do círculo seja suficientemente grande.

problemas, automaticamente teremos um algoritmo de mesma complexidade para o outro problema.

Além disso, esta relação nos permite determinar cotas inferiores para os problemas TRIANGULAÇÃO e VORONOI. Observe que, embora o diagrama de Delaunay não produza sempre uma triangulação de conv(C), a obtenção de uma triangulação de Delaunay é trivial a partir de Del(C): basta triangular cada região convexa R_k, o que pode ser feito através de $m-3$ diagonais de R_k, onde m é o número de vértices de R_k.

Desta forma, o diagrama de Voronoi fornece, em tempo linear, uma triangulação de conv(C), o que mostra que

TRIANGULAÇÃO $\propto$ VORONOI.

Por outro lado, é fácil mostrar que ORDENAÇÃO pode ser reduzida a TRIANGULAÇÃO. Dados números reais $x_1, x_2, ..., x_n$, construa o conjunto $C = \{(0,0), p_1, p_2, ..., p_n\}$, onde cada p_i é dado por $p_i = (x_i, 1)$. Então existe uma única triangulação de conv(C), obtida tomando-se os p_i de acordo com a ordem de suas abscissas x_i. Deste modo, um algoritmo capaz de obter triangulações é também capaz de ordenar. Logo, temos:

ORDENAÇÃO $\propto$ TRIANGULAÇÃO.

Em consequência, temos o seguinte teorema:

Teorema 4.7: *O problema ORDENAÇÃO pode ser reduzido aos problemas* VORONOI *e* TRIANGULAÇÃO. *Em consequência, qualquer algoritmo capaz de triangular conv(C) ou obter Vor(C) requer tempo $\Omega(n \log n)$ no pior caso.* ■

Na próxima seção descrevemos um algoritmo O(n log n) (portanto, ótimo) para obter a triangulação de Delaunay de conv(C).

4.4 Algoritmos para triangulação de Delaunay

Um algoritmo quadrático

Começamos por exibir um algoritmo capaz de obter uma triangulação de Delaunay de C em tempo O(n^2). No caso de C não possuir 4 pontos cocirculares, a triangulação de Delaunay

é única. No caso geral, a triangulação não é única, mas cada triângulo satisfaz a propriedade de que seu círculo circunscrito é vazio, isto é, não contém pontos de C em seu interior.

Para desenvolver o algoritmo, precisamos do seguinte fato:

Teorema 4.8: *Se pqr é um triângulo que ocorre numa triangulação de Delaunay de conv(C), então o ângulo prq é máximo dentre todos os ângulos da forma psq, onde s pertence a C e está no mesmo semiplano de r em relação a pq.*

Prova: Suponhamos que s está no mesmo semiplano de r e que $\angle(psq) > \angle(prq)$. Então s é interior ao círculo de vértices p, q e r, o que contradiz o fato de pqr ser um triângulo de Delaunay (figura 4.10). ■

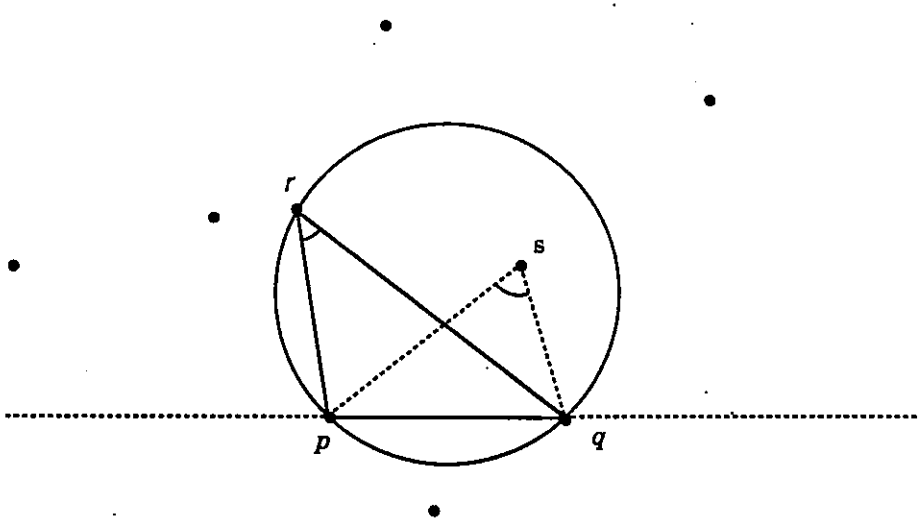


Figura 4.10 – Determinando o triângulo de base pq.

Desta forma, se pq é uma aresta da triangulação de Delaunay, podemos obter os triângulos que lhe são adjacentes determinando os pontos que, em cada semiplano, maximizam o ângulo que formam com o segmento pq. (Note que, caso pq faça parte da fronteira de conv(C), apenas um dos semi-planos contém pontos de C.)

Este processo pode ser visto como análogo ao utilizado pelo algoritmo "embrulho-para-presente" para determinar a face adjacente a uma aresta do fecho convexo tridimensional (isto é não é apenas uma analogia, como mostra o exercício 2). Como no caso de FC3D, uma estrutura de adjacência (como a *winged-edge*) pode ser utilizada para armazenar a triangulação. Também como no caso de FC3D, faces são introduzidas na triangulação e suas arestas testadas para verificar se já foram introduzidas anteriormente. Em caso negativo, um de seus semi-planos fica livre e a aresta deve ser explorada mais tarde. A única diferença é que, ao explorar o semi-plano livre de uma aresta, pode-se chegar à conclusão de que este semi-plano não contém pontos de C. Isto significa que a aresta está na fronteira de conv(C) e que, portanto, uma das faces a ela incidentes é a face ilimitada correspondente ao exterior de conv(C).

A face inicial pode ser obtida determinando uma aresta de conv(C) (pelo passo inicial do algoritmo de Jarvis) e aplicando o passo de "embrulho-para-presente" a esta aresta inicial. O algoritmo, que, como em FC3D, emprega uma fila $\mathcal{F}$ para armazenar as faces já geradas é:

Algoritmo 4.1: Determinação de Triangulação de Delaunay

```
1. Inicialização:
   Obtenha um triângulo inicial T, aplicando a técnica descrita acima.
   Coloque T em F e na estrutura winged-edge, com todas as suas arestas
   livres (exceto a correspondente à fronteira de conv (C)).
2. Enquanto F ≠ ∅ repita
3.     remova um triângulo T de F;
4.     para cada aresta livre de T
5.         determine a face adjacente T', usando "embrulho-para-presente";
6.         coloque T' assim gerada na fila F;
7.         coloque T' na estrutura winged-edge, conectando-a com as faces
           já geradas que lhe são adjacentes e determinando suas arestas
           livres.
```

A análise da correção e complexidade do algoritmo acima é análoga a que fizemos para o algoritmo embrulho para presente. Como no caso de FC3D, o algoritmo pára somente quando toda a triangulação tiver sido gerada. Observe que se as faces geradas (incluindo a face externa) não cobrissem o plano, alguma aresta de uma das faces estaria livre, o que não ocorre, já que o

algoritmo só termina quando todas as arestas livres tiverem sido exploradas. O tempo de execução é quadrático, pelas mesmas razões vistas no caso de FC3D. Assim, temos o seguinte:

Teorema 4.9: *O algoritmo 4.1 obtém uma triangulação de Delaunay para conv(C) em tempo* $O(n^2)$. ■

Um algoritmo $O(n \log n)$

A estratégia a ser usada para obter um algoritmo $O(n \log n)$ para a obtenção de uma triangulação de Delaunay consiste em empregar um algoritmo do tipo "Dividir para Conquistar". Ou seja, dado um conjunto C de pontos do plano, dividiremos este conjunto em dois subconjuntos de igual tamanho C_1 e C_2, triangularemos o fecho convexo de cada um destes subconjuntos e combinaremos os resultados para obter uma triangulação de C.

Suponhamos que as triangulações de Delaunay de $\text{conv}(C_1)$ e $\text{conv}(C_2)$ sejam conhecidas e consideremos a triangulação de Delaunay de $\text{conv}(C_1 \cup C_2)$. A primeira observação importante é dada pelo:

Lema 4.2: *Se uma aresta de* $Del(C_1 \cup C_2)$ *tem ambos os extremos em* C_1 *(resp.* C_2*) então ela é uma aresta de* $Del(C_1)$ *(resp.* $Del(C_2)$*).*

Prova: Se a é uma aresta de $\text{Del}(C_1 \cup C_2)$ então existe um círculo contendo seus extremos que exclui todos os elementos de $C_1 \cup C_2$. Mas se ambos os extremos estão em C_1 então esta condição implica que a é uma aresta de $\text{Del}(C_1)$. ■

Uma consequência deste lema é que, ao criar $\text{Del}(C_1 \cup C_2)$ a partir de $\text{Del}(C_1)$ e $\text{Del}(C_2)$, nunca inserimos arestas tendo ambos os extremos em C_1 ou C_2. Para obter $\text{Del}(C_1 \cup C_2)$ devemos determinar que arestas devem ser eliminadas em $\text{Del}(C_1)$ e $\text{Del}(C_2)$ e criar as arestas necessárias tendo um extremo em cada conjunto.

Para facilitar a execução da etapa do algoritmo encarregada de combinar $\text{Del}(C_1)$ e $\text{Del}(C_2)$, a etapa de separação será feita de tal modo que $\text{conv}(C_1)$ e $\text{conv}(C_2)$ sejam disjuntos. Uma forma de garantir que isto ocorra é ordenar os elementos de C de acordo com suas abscissas e, em caso de empate, conforme suas ordenadas e efetuar todos os passos de separação com base nesta ordenação. Neste caso, podemos pensar em C_1 e C_2 como os

conjuntos da esquerda e da direita, respectivamente e nosso problema é identificar quais arestas do tipo E–E e D–D devem ser eliminadas e quais arestas do tipo E–D devem ser criadas.

Lee e Schachter [LS] desenvolveram um algoritmo capaz de realizar esta tarefa de combinar os diagramas de Delaunay da esquerda e da direita em tempo linear, o que conduz a um algoritmo $O(n \log n)$ para obter triangulações de Delaunay. Uma exposição extremamente clara do algoritmo acompanhada de uma demonstração precisa de sua correção pode ser encontrada em [GS]. Por brevidade, omitiremos aqui muitos dos detalhes, para os quais sugeriremos que o leitor consulte [GS] que, além disso, descreve uma estrutura de dados topológica alternativa, a *quad-edge*.

Tomemos uma reta separando $\text{conv}(C_1)$ e $\text{conv}(C_2)$. Cada aresta de $\text{Del}(C_1 \cup C_2)$ corta esta reta e estas interseções determinam uma ordem vertical para tais arestas. As arestas mais baixa e mais alta são as tangente comuns a $\text{conv}(C_1)$ e $\text{conv}(C_2)$, que são parte da fronteira de $\text{conv}(C_1 \cup C_2)$ e, por esta razão, estão necessariamente em $\text{Del}(C_1 \cup C_2)$. O algoritmo de Lee e Schachter parte da tangente inferior (que pode ser obtida em tempo linear como vimos no capítulo 3) e encontra sucessivamente as demais arestas, até chegar à tangente superior. À medida que tais arestas são encontradas, as arestas de $\text{Del}(C_1)$ e $\text{Del}(C_2)$ que não fazem parte de $\text{Del}(C_1 \cup C_2)$ são determinadas e eliminadas do diagrama.

Suponha que tenhamos encontrado corretamente todas as arestas de $\text{Del}(C_1 \cup C_2)$, da tangente inferior até uma certa aresta pq (figura 4.11). A próxima aresta será incidente a pq em um dos extremos. Para determinar esta aresta, percorremos as arestas de $\text{Del}(C_1)$ incidentes a p e as arestas de $\text{Del}(C_2)$ incidentes a q, nos sentidos anti-horário e horário, respectivamente.

Sejam $p_1, p_2, ..., pk$ os vértices que determinam arestas incidentes a p, no sentido anti-horário. Se o círculo circunscrito ao triângulo pqp_1 contiver o ponto p_2, imediatamente concluímos que a aresta pp_1 não pertence a $\text{Del}(C_1 \cup C_2)$. Neste caso, eliminamos tal aresta do diagrama e repetimos o teste para o ponto p_2 (isto é, determinamos se p_3 está no círculo definido por p, q e p_2). Seja p_e o primeiro dos vizinhos de p a passar no teste acima. Seja q_d o primeiro ponto a passar no teste análogo aplicado aos vizinhos de q.

Entre p_e e q_d escolhemos o que determina o maior ângulo com pq (ou seja, tal que o círculo definido por p, q e o ponto escolhido exclua o outro ponto). Se p_e é o ponto escolhido, a aresta p_eq é acrescentada a $\text{Del}(C_1 \cup C_2)$; senão, pq_d é a aresta escolhida.

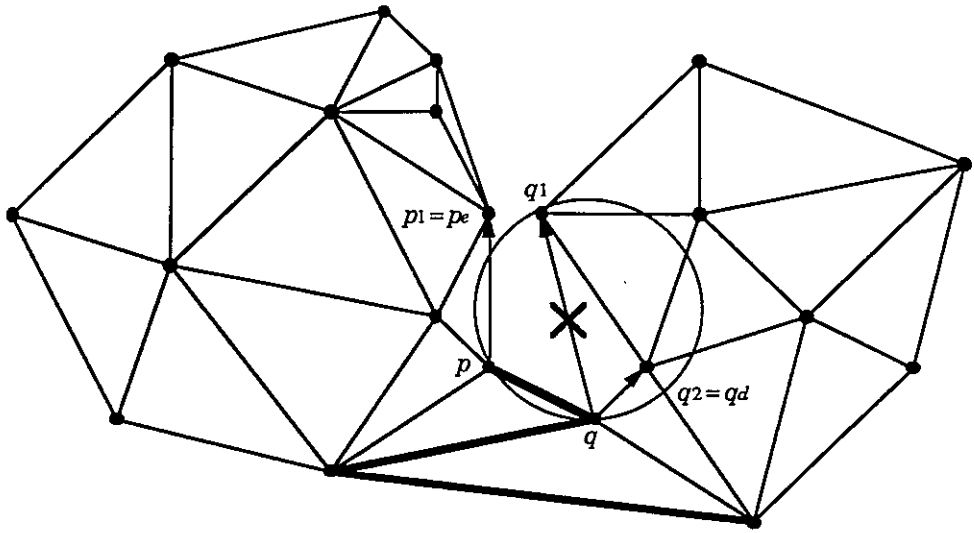


Figura 4.11 – Determinando a próxima aresta de Del($C_1 \cup C_2$).

No caso da figura 4.11, o ponto p_1 passa no teste e é, portanto, o candidato p_e a definir a próxima aresta da triangulação. O ponto q_1 *não* passa no teste (observe que o círculo determinado por p, q e q_1 contém o próximo vizinho q_2 de q); por esta razão, a aresta qq_1, que fazia parte de Del(C_2), deve ser eliminada. O primeiro ponto q_d a passar no teste é q_2. Como o ângulo $\angle(pq_dq)$ é menor que $\angle(pp_eq)$, a aresta pq_d é acrescentada ao diagrama.

Descrevemos abaixo, de maneira mais formal, o procedimento acima. Note que o algoritmo requer a determinação das diversas arestas incidentes aos extremos da aresta pq. Para garantir que esta determinação seja feita em tempo constante por aresta incidente, admitiremos que Del(C_1) e Del(C_2) sejam dadas em uma estrutura do tipo *winged-edge*. Desta maneira, a obtenção da aresta imediatamente à esquerda ou à direita de uma aresta orientada uv (representadas por esq(uv) e dir(uv)) pode ser feita em tempo constante (veja o exercício 5).

Algoritmo 4.2 - Obtenção de Del($C_1 \cup C_2$) a partir de Del(C_1) e Del(C_2)

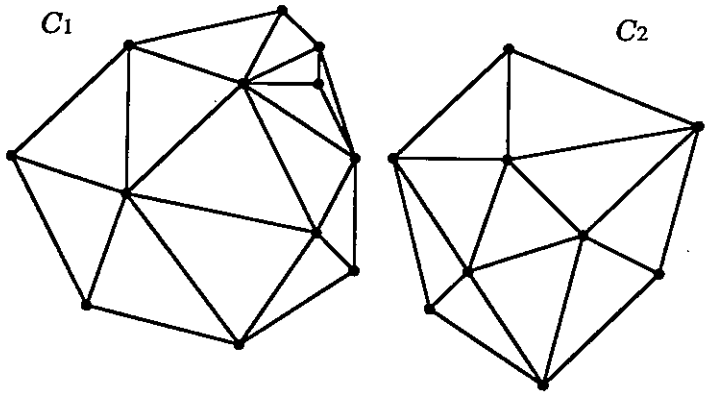
(C_1 e C_2 são tais que conv(C_1) e conv(C_2) são disjuntos)

```
0. Faça Del(C1∪C2) = Del(C1) ∪ Del(C2)
1. Determine a tangente comum inferior pq e acrescente-a a Del(C1∪C2).
```

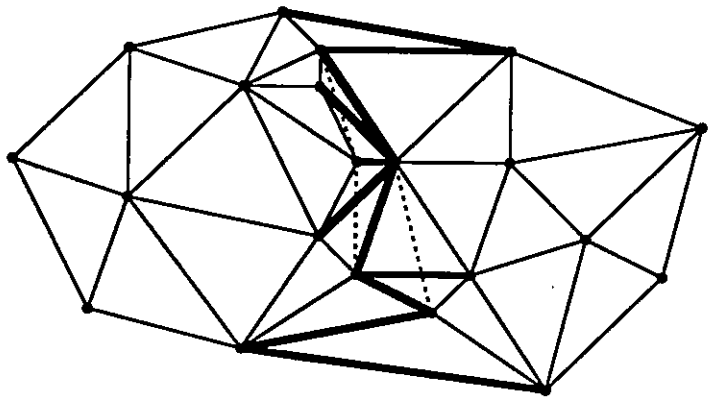
```
2. Enquanto pq não é a tangente comum superior:
3.    Determinação do candidato em C1:
          pe = esq (pq) , pps = esq (ppe)
          Se pe está acima de pq então faça:
               Enquanto ps pertence ao círculo definido por p, q e pe:
                    Elimine a aresta ppe e faça pe = ps, pps = esq(ppe)
          Senão, não há candidato em C1
4.    Determinação do candidato em C2:
          qqd = dir (qp) , qqs = dir (qqd)
          Se qd está acima de pq então faça:
               Enquanto qs pertence ao círculo definido por p, q e qd:
                    Elimine a aresta qqd e faça qd = qs, qqs = dir(qqd)
          Senão, não há candidato em C2
5.    Escolha da nova aresta
      Se apenas pe é candidato
          insira peq em Del(C1∪C2) e faça pq = peq
      Senão, se apenas qd é candidato
          insira pqd em Del(C1∪C2) e faça pq = pqd
      Senão, se ambos são candidatos
          insira o que determina o maior ângulo com pq
      (Senão, se não há candidatos, então pq é a tangente comum superior)
```

A figura 4.12 mostra diversas etapas da construção de Del($C_1 \cup C_2$), a partir de Del(C_1) e Del(C_2). As arestas eliminadas de Del(C_1) e Del(C_2) são representadas em tracejado, enquanto as arestas acrescentadas pelo algoritmo são mostradas em negrito.

Note que Del(C_1) e Del(C_2) têm ambos um número linear de arestas. Por outro lado, em cada passo do cálculo de p_e e q_d uma aresta é eliminada, o que demonstra que o algoritmo é certamente linear nos tamanhos de C_1 e C_2. No entanto, a prova da correção do algoritmo é sutil. Não é óbvio que a aresta pq_d ou p_eq escolhida pelo algoritmo seja de fato uma aresta de Del($C_1 \cup C_2$), nem que as arestas eliminadas pelo algoritmo o sejam corretamente. Sugerimos que o leitor consulte [GS] para se convencer da validade do teorema a seguir.



(a) Triangulações de Delaunay de C_1 e C_2.



(b) Triangulação de Delaunay de $C_1 \cup C_2$.

Figura 4.12 – Combinando as triangulações de C_1 e C_2

Teorema 4.9: *O algoritmo 4.2 corretamente obtém Del($C_1 \cup C_2$) em tempo linear nos números de elementos de C_1 e C_2. Logo, é possível obter uma triangulação de Delaunay de n pontos do plano em tempo O(n log n).*

Relação com os algoritmos para FC3D

A similaridade entre os algoritmos para a obtenção de fecho convexo no $\mathbf{R}^3$ e os algoritmos para obter a triangulação de Delaunay no plano não é casual. No exercício 2, pedimos que o leitor demonstre que o problema de obter a triangulação de Delaunay de conv(C) pode ser reduzido à determinação do fecho convexo da imagem C' de C no parabolóide que é o gráfico da função f: $\mathbf{R}^2 \rightarrow \mathbf{R}$ definida por $f(x, y) = x^2 + y^2$. As faces da triangulação de Delaunay de conv(C) estão em correspondência biunívoca com as faces de conv(C') que compõem o seu fecho convexo inferior.

4.5 Problemas resolvidos pela triangulação de Delaunay

A existência de um algoritmo O(n log n) para determinar uma triangulação de Delaunay de conv(C) acarreta a existência de algoritmos de mesma complexidade para diversos problemas importantes. O primeiro deles, naturalmente, é o que permite obter o diagrama de Voronoi de C.

Uma questão a ser levantada neste ponto é como representar os polígonos ilimitados que ocorrem no diagrama de Voronoi . Uma forma simples de lidar com este problema é admitir que as arestas ilimitadas conectam vértices do diagrama de Voronoi com um vértice no "infinito". Este vértice no infinito pode ser visualizado de maneira mais concreta através de uma projeção estereográfica do diagrama de Voronoi numa esfera tangente ao plano. As projeções das arestas ilimitadas convergem no polo da esfera. Feita esta convenção, o diagrama de Voronoi pode ser representado por uma estrutura tipo *winged-edge*. Os vértices no infinito podem ser representados juntamente com os demais vértices usando, por exemplo, coordenadas homogêneas (ver, por exemplo, [GV]).

Teorema 4.10: *O diagrama de Voronoi determinado por n pontos do plano pode ser obtido em tempo O(n log n).*

Prova: Pelo algoritmo 4.2, o diagrama de Delaunay de conv(C) pode ser obtido em tempo O(n log n). Para os pontos x_i de C que não estão na fronteira de conv(C), o polígono de Voronoi correspondente tem por vértices os circuncentros das faces de Del(C) que se encontram em x_i. Quando x_i está na fronteira de conv(C), o polígono de Voronoi correspondente é

ilimitado e é obtido acrescentando as arestas ilimitadas, que são mediatrizes das arestas adjacentes a x_i na fronteira de conv(C). O número total de circuncentros a determinar é $O(n)$ e o número total de arestas em Vor(C) também é $O(n)$. Logo, a obtenção dos polígonos de Voronoi (ou sua organização em uma estrutura tipo *winged-edge*) é feita em tempo linear, a partir de Del(C). O tempo total para a obtenção de Vor(C) é, então, $O(n \log n)$. ■

Consideremos agora os seguintes problemas:

PAR MAIS PRÓXIMO: *Dados pontos $x_1, x_2, ..., x_n$ do $\mathbf{R}^2$ obter o par (x_i, x_j) tal que a distância $d(x_i, x_j)$ seja mínima.*

TODOS OS VIZINHOS MAIS PRÓXIMOS: *Dados pontos $x_1, x_2, ..., x_n$ do R^2 obter, para cada x_i, o ponto x_j tal que $d(x_i, x_j)$ seja mínima.*

É claro que

PAR MAIS PRÓXIMO $\propto$ TODOS OS VIZINHOS MAIS PRÓXIMOS,

já que, uma vez obtido o vizinho mais próximo de cada ponto, um passo linear determina qual destas n distâncias é a menor de todas. Os dois problemas podem ser resolvidos eficientemente com auxílio da triangulação de Delaunay, graças ao seguinte:

Teorema 4.11: *Seja $C = \{x_1, x_2, ..., x_n\}$. Se x_ix_j é um segmento de comprimento mínimo dentre todos os segmentos obtidos ligando x_i a outros pontos de C, então x_ix_j é necessariamente uma aresta de Del(C).*

Prova: Basta considerar o círculo de diâmetro x_ix_j. Se este círculo contivesse algum outro ponto x_k de C em seu interior, então teríamos $d(x_i, x_j) > d(x_i, x_k)$, o que contradiz x_ix_j ser de comprimento mínimo. Logo, existe um círculo passando por x_i e x_j que não contém outros pontos de C, o que mostra que x_ix_j é uma aresta de Del(C). ■

Em virtude deste teorema, para buscar o elemento de C mais próximo de x_i basta procurar entre seus vizinhos em Del(C). Como o número total de arestas é linear em n, a determinação de todos os vizinhos mais próximos é feita em tempo $O(n)$. Isto é, demonstramos que:

Teorema 4.12: *Os problemas* PAR MAIS PRÓXIMO *e* TODOS OS VIZINHOS MAIS PRÓXIMOS *podem ser resolvidos em tempo $O(n \log n)$.* ■

Pode-se demonstrar que estes algoritmos são ótimos. Isto é, que $\Omega(n \log n)$ é uma cota inferior para a complexidade de um algoritmo que resolva os problemas acima (sob o modelo de árvores algébricas de decisão). No entanto, não podemos usar o problema de ordenação para estabelecer este resultado. O leitor deve consultar [Me] ou [PS] para uma demonstração, que se baseia em um resultado de Ben-Or ([BO]), que por sua vez utiliza resultados de Geometria Algébrica.

Um outro problema que admite um algoritmo $O(n \log n)$ devido ao teorema 4.11 é a versão euclidiana do problema de encontrar a árvore varredora mínima de um conjunto de n pontos do plano (ver o exercício 3 e [PS]). Na verdade, diversos grafos importantes associados a um conjunto de pontos do plano são subgrafos do diagrama de Delaunay e podem ser obtidos eficientemente a partir dele [T].

4.6 Outros problemas de triangulação

O problema de determinar uma triangulação do fecho convexo de um conjunto de pontos do plano pode ser visto como um caso particular do seguinte problema mais geral, de grande interesse prático:

TRIANGULAÇÃO DE UM DOMÍNIO: *Dado um domínio D do $\mathbf{R}^2$, delimitado por linhas poligonais, obter uma triangulação desta região (isto é, exprimir D como a união de um conjunto de triângulos que formem um complexo simplicial) (figura 4.13).*

O problema acima possui muitas variantes. Da forma como enunciado acima, não se estabeleceu nenhuma condição à respeito do conjunto de vértices e arestas a serem empregados na triangulação. Dependendo das condições impostas sobre estes vértices e arestas, obtém-se vários problemas específicos de triangulação.

Triangulação de Polígono Simples

Consideremos o caso em que D é um polígono simples (isto é, caracterizado por uma única linha poligonal simples e fechada) e não é permitido utilizar vértices adicionais. Ou seja, desejamos resolver o seguinte problema:

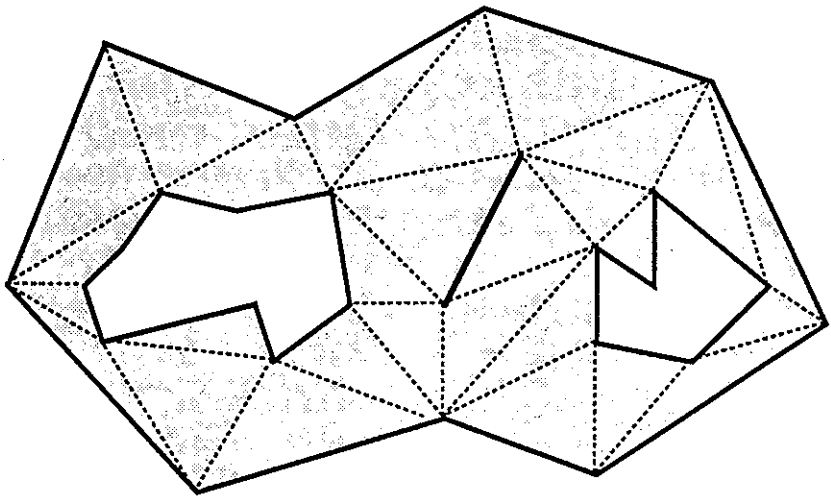


Figura 4.13 – Triangulando uma região do plano.

TRIANGULAÇÃO DE UM POLÍGONO SIMPLES: *Dado um polígonos simples P, obter uma triangulação do polígono que utilize apenas lados e diagonais de P.*

Não é inteiramente óbvio que exista uma triangulação de P satisfazendo a estas condições. Mas é fácil mostrar que, se existir, ela será determinada por n–3 diagonais de P. Abaixo, apresentamos uma prova algorítmica de que tal triangulação de fato existe.

Para mostrar que é possível decompor um polígono simples $P = p_1p_2...p_n$ em triângulos, basta mostrar que é possível obter uma diagonal inteiramente contida em P. O traçado desta diagonal decompõe P em dois polígonos simples, cada um deles tendo menos vértices que o polígono primitivo, aos quais o mesmo processo pode ser aplicado de forma recursiva.

Tomemos um vértice de P tal que o ângulo correspondente seja convexo. Certamente existe um tal vértice (por exemplo, o vértice de P que possui abscissa mínima), que pode ser encontrado em tempo linear. Sem perda de generalidade, suponhamos que p_1 satisfaz a esta condição. Consideremos a diagonal p_2p_n determinada pelos vértices adjacentes a p_1. Se esta diagonal estiver inteiramente contida em P (o que pode ser verificado em tempo linear, bastando para isto testar se o triângulo $p_1p_2p_n$ contém algum outro vértice de P), teremos decomposto P em um triângulo e um polígono P' com n–1 vértices (figura 4.14). Senão, seja V' o conjunto de vértices de P, distintos de p_1, p_2 e p_n e contidos no triângulo $p_1p_2p_n$. Tomemos por p_1

uma reta paralela a p_2p_n e deslizemos esta reta até encontrar algum dos vértices em V'. Esta operação equivale a encontrar em V' o vértice p_k cuja coordenada baricêntica relativa a p_1 no triângulo $p_1p_2p_n$ é máxima (figura 4.15).

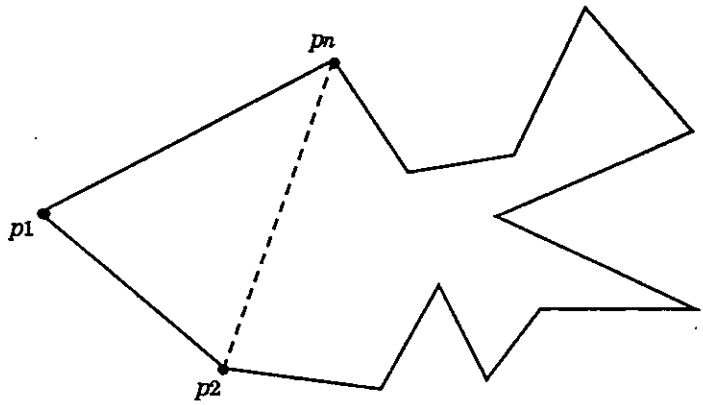


Figura 4.14 – p_2p_n é uma diagonal válida.

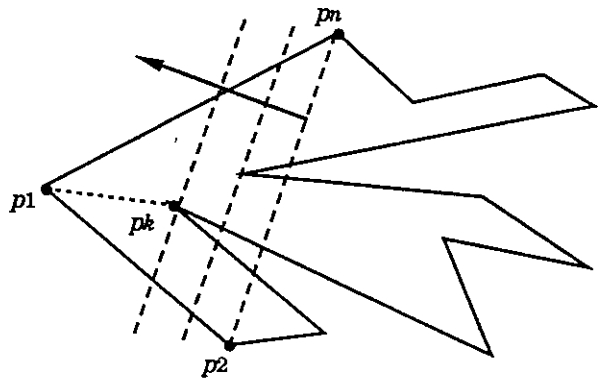


Figura 4.15 – p_2p_n não é uma diagonal válida.

Temos, então o seguinte:

Lema 4.3: *A diagonal p_1p_k, onde p_k é determinado da forma descrita acima, está inteiramente contida em P.*

Prova: O segmento p_1p_k está contido no triângulo $p_1p_2p_n$. Como p_1 determina um ângulo convexo, existe pelo menos um trecho de p_1p_k, adjacente a p_1, que está contido em P. Para

mostrar que ele está inteiramente contido em P, basta então mostrar que ele não corta nenhum lado p_ip_j de P.

Se p_i e p_j estão em V' (isto é, estão no triângulo $p_1p_2p_n$), certamente o lado determinado por eles não pode cortar p_1p_k, já que p_i e p_j estão ambos pelo menos tão afastados de p_1 quanto p_k.

Suponhamos agora que p_i e p_j não estão ambos em V'. Se estão ambos fora de V' (isto é, são exteriores a $p_1p_2p_n$), o segmento p_ip_j não intercepta o triângulo $p_1p_2p_n$; note que, para fazê-lo, teria que interceptar duas vezes a fronteira de $p_1p_2p_n$ e pelo menos uma destas interseções seria ao longo de p_1p_2 ou p_1p_n, o que contradiz a hipótese do polígono ser simples. Por outro lado, se exatamente um dos extremos está em V', então p_ip_j necessariamente corta p_2p_n, o que indica que todos os pontos de p_ip_j estão pelo menos tão afastados de p_1 do que p_k. Portanto, p_ip_j não pode cortar p_1p_k. ■

O algoritmo correspondente ao processo descrito acima é:

Algoritmo 4.3: Triangulação de Polígono Simples

```
0. Se P for um triângulo, pare (não há o que triangular).
1. Senão, determine um vértice de P tal que o ângulo correspondente seja
   convexo (por exemplo, tome o vértice de abscissa mínima). Sem perda de
   generalidade, suponha que p1 seja um vértice com esta propriedade.
2. Se o conjunto V' dos vértices de P que estão contidos no triângulo
   P1P2Pn é vazio então
3.    Faça P1 = P1P2Pn e P2 = P2P3···Pn.
   Senão,
4.    determine entre os vértices em V' o vértice Pk cuja coordenada
      baricêntrica em relação a P1 no triângulo P1P2Pn é mínima.
5.    faça P1 = P1P2 ···Pk e P2 = P1PkPk+1 ···Pn.
6. Recursivamente, aplique o algoritmo a P1 e P2.
```

Note que cada execução do passo 2 requer tempo $O(n)$ e gera uma das $n-3$ diagonais necessárias para triangular P. Logo, demonstramos o seguinte:

Teorema: *Todo polígono simples P admite uma triangulação que usa apenas lados e diagonais de P. O algoritmo 4.3 determina uma tal triangulação em tempo $O(n^2)$.* ■

A busca de algoritmos eficientes para triangular polígonos simples possui uma longa e ilustre história em Geometria Computacional. Um algoritmo $O(n \log n)$ foi dado em 1978 por Garey, Johnson, Preparata e Tarjan [GJPT]. Esta complexidade só foi melhorada em 1987, quando Tarjan e Van Wyk [TW] exibiram um algoritmo $O(n \log \log n)$ para o mesmo problema. Finalmente, em 1990, Chazelle [Cz] obteve um algoritmo $O(n)$ para o problema.

Esta sequência de resultados mostra bem a utilidade de se obter cotas inferiores para problemas. Quando o melhor algoritmo existente para um problema tem complexidade maior que a melhor cota inferior obtida para este problema, um problema em aberto fica imediatamente caracterizado: o objetivo é diminuir a complexidade do algoritmo ou obter uma melhor cota inferior para o problema, de modo que a otimalidade do algoritmo possa ser demonstrada.

Triangulação Com Restrições

Examinamos agora um algoritmo capaz de resolver o seguinte problema:

TRIANGULAÇÃO COM RESTRIÇÕES: *Dado um conjunto C de pontos do plano e um conjunto G de segmentos com extremos em C (tais que dois elementos quaisquer de G não se interceptam a não ser em seus extremos), obter uma triangulação do fecho convexo de C, cujo conjunto de vértices seja C e que inclua todos os segmentos em G.*

Em outras palavras, desejamos obter uma triangulação de conv(C) satisfazendo a restrição de que o grafo planar determinado por G seja um subgrafo desta triangulação. É claro, portanto, que este problema generaliza o problema de triangular conv(C) estudado nas seções anteriores deste capítulo. Por outro lado, a solução deste problema resolve automaticamente o problema de triangular um domínio do plano introduzido no início desta seção (incluindo triangulação de polígonos simples): uma vez triangulado o fecho convexo do domínio, basta colecionar as faces da triangulação que estão contidas no domínio a ser triangulado.

É possível introduzir o conceito de triangulação de Delaunay com restrições, que possui propriedades análogas à triangulação de Delaunay estudada nas seções iniciais deste capítulo. Tais triangulações também podem ser obtidas em tempo $O(n \log n)$, utilizando um método que generaliza o algoritmo de Lee e Schachter (ver [Ch]). No entanto, preferimos apresentar um outro método para obter uma triangulação de conv(C) (com ou sem restrições), que também pode ser implementado de modo a operar em tempo $O(n \log n)$. O método utiliza uma técnica

bastante comum em Geometria Computacional, conhecida como **varredura** (*sweeping*). A triangulação gerada por este método, porém, não tem as propriedades que caracterizam a triangulação de Delaunay: ela tende a gerar triângulos finos, o que pode ser desvantajoso para certas aplicações.

Começamos com o caso em que não há restrições, isto é, simplesmente desejamos triangular conv(C). A idéia básica é tomar uma reta de direção fixa e varrer o plano com esta reta, parando cada vez que ocorrer um evento relevante ao nosso processo. Neste caso, utilizaremos uma reta vertical, e pararemos cada vez que esta reta encontrar um dos pontos de C. A idéia básica do algoritmo é, ao encontrar um novo ponto p_i, uni-lo a todos os pontos anteriores que são visíveis de p_i (isto é, a todos os pontos p_j tais que o interior de p_ip_j não intercepta o interior de nenhum dos segmentos já introduzidos). O algoritmo, cujo comportamento é ilustrado na figura 4.16, é simplesmente:

Algoritmo 4.4: Algoritmo de varredura para triangular conv(C)

```
1. Ordene os pontos de C segundo suas abscissas crescentes
2. Examine, em ordem, um ponto por vez, unindo-o a todos os pontos
   anteriores que podem ser vistos a partir dele.
```

Teorema 4.13: *O algoritmo 4.4 obtém uma triangulação de conv(C) em tempo O(n log n).*

Prova: Para provar a correção do algoritmo, empregamos indução no número de pontos já visitados pelo algoritmo. Para $n \leq 3$ é óbvio que o algoritmo funciona corretamente. Suponha que, ao chegar ao ponto p_{k+1}, o algoritmo tenha já determinado uma triangulação válida para P_k = conv($\{p_1, p_2, ..., p_k\}$). O ponto p_{k+1} é externo a P_k. Logo, entre os pontos $p_1, p_2, ..., p_k$, aqueles que são visíveis de p_{k+1} são os vértices consecutivos de P_k situados entre as linhas de suporte a P_k conduzidas por p_k. A inclusão destes segmentos estende a triangulação de P_k a uma triangulação de P_{k+1} = conv($\{p_1, p_2, ..., p_{k+1}\}$).

O passo 1 do algoritmo, que consiste na etapa de ordenação, pode ser executado em tempo O(n log n). Para determinar o tempo necessário para a inclusão de um novo ponto, no passo 2, é crucial observar que o ponto p_k é necessariamente visível a partir de p_{k+1}. Portanto, para determinar todos os vértices de P_k que são visíveis de p_{k+1}, basta partir de p_k e percorrer a fronteira de P_k, em cada sentido, até chegar ao primeiro vértice que não seja mais visível a partir de p_{k+1} (veja o exercício 6). Logo, o tempo consumido determinando todas as arestas

que terminam em p_{k+1} é proporcional ao seu número. Em consequência, o tempo total gasto no passo 2 é proporcional ao número total de arestas da triangulação. Isto é, o passo 2 é executado em tempo $O(n)$, o que faz com que a complexidade total do algoritmo 4.4 seja $O(n \log n)$. ■

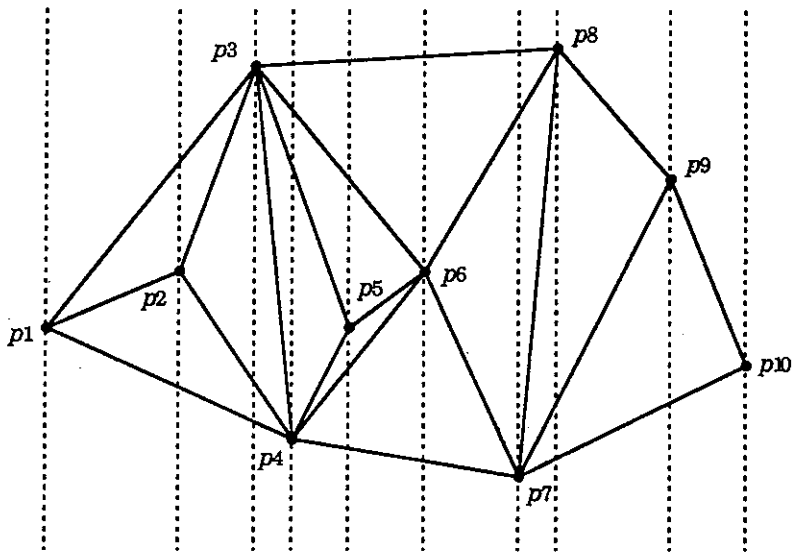


Figura 4.16 – Triangulação por varredura.

Vejamos agora como estender o algoritmo 4.4 para triangulações com restrição. A descrição do algoritmo ainda é a mesma dada acima. A diferença ocorre nos testes a serem feitos para acrescentar os segmentos da triangulação correspondentes a um novo ponto p_{k+1}, devido à presença dos segmentos obrigatórios. Quando o algoritmo encontra o ponto inicial de um dos segmentos s de G, este segmento não é introduzido imediatamente na triangulação (já que o algoritmo 4.4 acrescenta segmentos sempre a partir de seu vértice final); no entanto, ele deve ser considerado nos testes de visibilidade. Dizemos, então, que s está **em aberto**. Quando o ponto final de s é atingido, aí sim ele é introduzido na triangulação.

Para a eficiência do algoritmo 4.4, no caso sem restrições, foi fundamental que os segmentos acrescentados nos k primeiros passos determinassem uma triangulação de P_k =

conv($\{p_1, p_2, ..., p_k\}$). Isto permitiu que os segmentos a serem acrescentados no passo $k+1$ fossem determinados pelos vértices consecutivos da fronteira de P_k situados entre as tangentes conduzidas por p_{k+1}, que podiam ser obtidos de modo eficiente. Agora, a união P_k de todas as regiões determinadas pelos segmentos acrescentados até o passo k não é mais um polígono convexo. No entanto, podemos demonstrar, por indução, que sua fronteira é tal que cada trecho situado entre dois segmentos em aberto consecutivos é uma linha poligonal convexa (veja a figura 4.17 e o exercício 7). Logo, para achar os pontos visíveis a partir do ponto p_{k+1}, determinamos entre que segmentos em aberto ele se situa e obtemos os pontos visíveis (necessariamente consecutivos) na linha poligonal convexa correspondente. Como o último ponto acrescentado a esta poligonal é certamente visível a partir de p_{k+1}, podemos novamente obter todas as arestas que terminam em p_{k+1} simplesmente partindo deste último ponto e marchando sobre a linha poligonal, em cada direção, até encontrar o primeiro ponto que não seja visível. Se p_{k+1} for um ponto onde um segmento em aberto s termina, ele "verá" pontos de duas regiões. Neste caso, o segmento em aberto é introduzido na triangulação, juntamente com as arestas correspondentes aos pontos visíveis a partir de p_{k+1} em cada região.

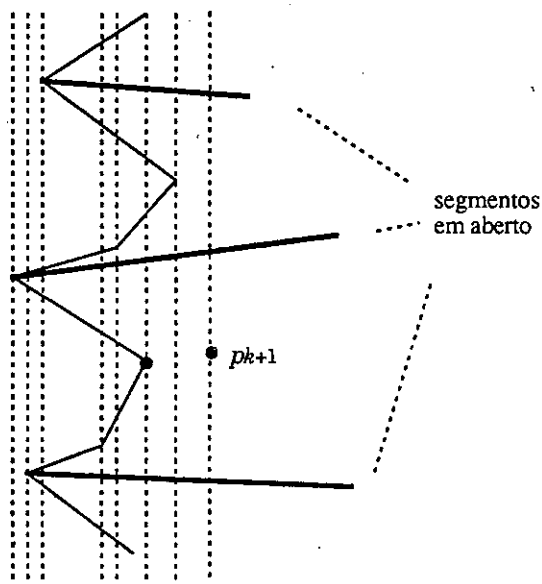


Figura 4.17 - As linhas poligonais entre segmentos em aberto são convexas.

Como no caso sem restrições, o tempo total consumido encontrando pontos visíveis é proporcional ao número total de arestas, ou seja, $O(n)$. Isto não inclui, no entanto, o tempo gasto para determinar a poligonal à qual um novo ponto p_{k+1} deve ser conectado (isto é, os segmentos em aberto entre os quais o novo ponto p_{k+1} se encontra). Para que a complexidade do algoritmo permaneça $O(n \log n)$ é necessário que esta determinação seja feita em tempo logarítmico. Para tal, basta que os segmentos em aberto sejam mantidos em uma árvore balanceada (veja o capítulo 1). Deste modo, a inserção e eliminação de segmentos em aberto, bem como a determinação dos segmentos em aberto adjacentes a um novo ponto a ser considerado, podem ser executadas em tempo $O(\log n)$ por iteração. Portanto:

Teorema 4.14: *O algoritmo 4.4 resolve o problema de triangulação com restrições em tempo $O(n \log n)$.* ■

A figura 4.18 mostra o resultado de se aplicar o algoritmo 4.4 ao mesmo conjunto de pontos da figura 4.16, mas sob a restrição de que os segmentos assinalados em negrito devem obrigatoriamente estar entre as arestas da triangulação.

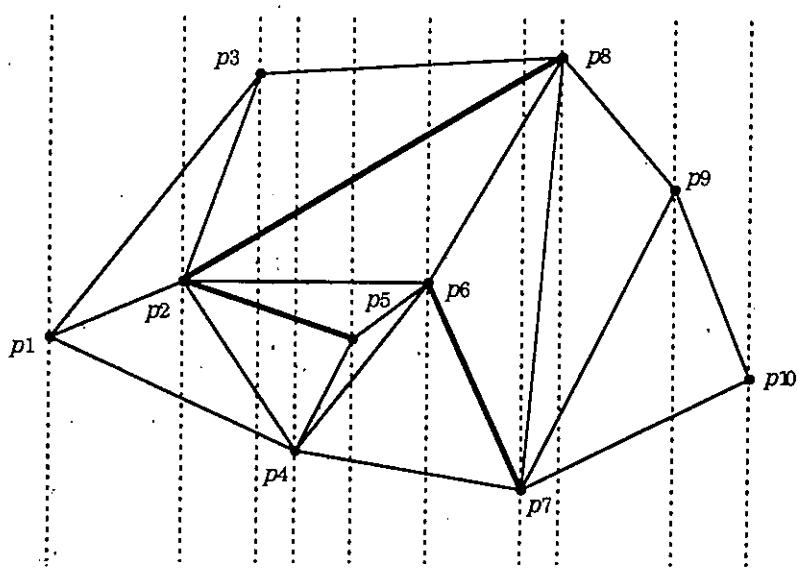


Figura 4.18 – Triangulação com restrições por varredura.

Note que o resultado anterior fornece um método capaz de triangular um polígono simples em tempo $O(n \log n)$. Para tal, podemos aplicar o algoritmo 4.4 ao conjunto de vértices do polígono simples, tomando seus lados como os segmentos obrigatórios da triangulação. Ao final, basta determinar que porções da triangulação obtida estão efetivamente no polígono (o que pode ser feito em tempo linear). É, porém, possível adaptar o algoritmo 4.4 de modo a só gerar segmentos que sejam interiores ao polígono simples (veja [HM] e exercício 8).

4.7 Localização de pontos em subdivisões planares

Na seção 4.1, justificamos nosso interesse em desenvolver algoritmos capazes de determinar diagramas de Voronoi e triangulações utilizando o problema de estender uma função definida sobre um conjunto C. Por exemplo, conhecendo uma triangulação de $\text{conv}(C)$, podemos, dado um novo ponto x, determinar o triângulo contendo x e, a partir daí, calcular $f(x)$ baseados nos valores de f nos vértices de T. Para que isto possa ser feito de modo eficiente, é necessário que possamos, eficientemente, determinar qual das regiões da triangulação contém o ponto x.

De um modo geral, desejamos resolver, de modo eficiente, o seguinte problema:

LOCALIZAÇÃO EM SUBDIVISÃO PLANAR: *Dada uma subdivisão planar determinada por segmentos de reta, representada por uma estrutura de adjacências, e um ponto x, determinar uma face da subdivisão que contenha x.*

Antes de tudo, devemos ser mais precisos sobre o que entendemos por determinação "eficiente". Como cada aresta em uma subdivisão planar ocorre em exatamente duas faces e como podemos determinar se um ponto pertence a um polígono simples em tempo linear no número de arestas desta face, em tempo linear no número de arestas podemos encontrar uma face contendo x: basta examinar cada face e testar se x pertence a ela.

Uma razão para não ficarmos satisfeitos com esta complexidade $O(n)$ é que, na versão unidimensional, em que as regiões são simplesmente intervalos consecutivos, a localização de um ponto pode ser feita em $O(\log n)$. E, de fato, há algoritmos capazes de executar a localização de pontos em subdivisões planares em tempo $O(\log n)$. Para tal, esses algoritmos criam uma estrutura auxiliar de busca. Os melhores algoritmos são capazes de fazer localização

em tempo O(log n) com uma estrutura auxiliar que utiliza armazenagem adicional O(n) e que pode ser construída em tempo O(n log n) (ver o capítulo 2 de [PS]).

No caso de localização de pontos em diagramas de Voronoi e triangulações de Delaunay, o tempo e a memória necessárias a esta estrutura adicional são da mesma ordem de complexidade que as requeridas pelo algoritmo que encontra os diagramas. Portanto, é possível se executar localização de pontos em tempo O(log n) sem pagar qualquer custo adicional do ponto de vista assintótico.

Exercícios

1. Mostre que se up, uq e ur são cordas distintas e consecutivas de um círculo então up ou ur tem comprimento menor que uq (utilizado na prova do teorema 4.3).

2. Seja $C = \{p_1, p_2, ..., p_n\}$ um conjunto de pontos do plano. Seja $C' = \{q_1, q_2, ..., q_n\}$ um conjunto de pontos do espaço, obtido associando a cada ponto $p_i = (x_i, y_i)$ o ponto $q_i = (x_i, y_i, x_i^2 + y_i^2)$.

a) Mostre que o círculo definido por p_i, p_j e p_k não contém outros pontos de C se e somente se nenhum ponto de C' está abaixo do plano definido por q_i, q_j e q_k.

b) Mostre que as faces da triangulação de Delaunay de conv(C) correspondem às faces do fecho convexo inferior de C'. Isto é, as faces de conv(C') que satisfazem a propriedade de que nenhum ponto de conv(C') está abaixo do seu plano.

3. Seja C um conjunto finito de pontos do plano e sejam S e T subconjuntos disjuntos de C tais que $S \cup T = C$. Mostre que se pq é o menor segmento entre pontos de S e pontos de T então pq é necessariamente uma aresta do diagrama de Delaunay de C. (Este resultado pode ser utilizado para obter um algoritmo O(n log n) que determine a árvore de comprimento total mínimo cujos vértices são os pontos de C; veja o capítulo 6 de [PS]).

4. Suponha que uma subdivisão planar esteja armazenada em uma estrutura *winged-edge*. Seja $a = uv$ uma aresta desta subdivisão, considerada orientada de u para v. Escreva as funções esq(uv) e dir(uv), que retornam as arestas imediatamente à esquerda e à direita de uv. (Lembre-se que, na estrutura de dados, a pode estar orientada de v para u.)

5. Mostre que o dual de uma triangulação de um polígono simples é uma árvore. Conclua que todo polígono simples tem pelo menos duas orelhas disjuntas.

6. Escreva um algoritmo que dado um polígono convexo P e um ponto exterior p_0, determina se um vértice p_i é visível a partir de p_0.

7. No algoritmo de varredura para triangulações com restrições, mostre, por indução, que a fronteira do polígono obtido até o passo k é dividido em trechos convexos pelos segmentos em aberto.

8. Modifique o algoritmo de varredura para triangulações com restrição de modo a obter um algoritmo capaz de triangular um polígono simples. Isto é, o algoritmo deve gerar apenas as diagonais efetivamente contidas no polígono. [Sugestão: para cada faixa vertical, mantenha informação sobre as regiões internas e externas ao polígono].

Referências

[Ba] B. Baumgart, A polyhedron representation for computer vision, *AFIPS Conf. Proc*, 589–596 (1975).

[BO] M. Ben-Or, Lower bounds for algebraic computation trees, *Proc. 15th ACM Annual Symp. on Theory of Comput.*, 80–86 (1983).

[Bl] L. Blum, *Theory of Computation over the Reals*, 18º Colóquio Brasileiro de Matemática, IMPA (1991).

[BSS] L. Blum, M. Shub e S. Smale, On a theory of computation and complexity over the real numbers: NP-completeness, recursive functions and universal Machines, *Bull. Amer. Math. Soc.* **21** (1), 1–46 (1989).

[Ch] P. Chew, Constrained Delaunay triangulations, *Algorithmica* **4**, 97–108 (1989).

[CK] D. R. Chand e S. S. Kapur, An algorithm for convex polytopes, *J. ACM* **17**(1), 78–86 (1970).

[Cz] B. Chazelle, Triangulating a simple polygon in linear time, Tech. Report CS-TR-264-90, Dept. of Comp. Science, Princeton University (maio de 1990).

[E] J. Edmonds, Paths, trees and flowers, *Canadian J. Math.* **17**, 449–467 (1965).

[G] R. L. Graham, An efficient algorithm for determining the convex hull of a finite planar set, *Info. Proc. Letters* **1**, 132–133 (1972).

[GKP] R. L. Graham, D. E. Knuth e O. Patashnik, *Concrete Mathematics: a Foundation for Computer Science*, Addison-Wesley, Reading, Mass. (1989).

[GJ] M. R. Garey, D. S. Johnson, *Computers and Intractability: a Guide to the Theoy of NP-Completeness*, W. H. Freeman, San Francisco (1979).

[GJPT] M. R. Garey, D. S. Johnson, F. P. Preparata e R. E. Tarjan, Triangulating a simple polygon, *Info. Proc. Letters* 7(4), 175–180 (1978).

[GS] L. J. Guibas e J. Stolfi, Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams, *ACM Trans. Graphics* 4, 141–155 (1985).

[GS2] L. J. Guibas e J. Stolfi, *Notes on Computational Geometry*, Notas de aula, Stanford University (1982).

[GV] J. M. Gomes e L. C. Velho, *Conceitos Básicos de Computação Gráfica*, VII Escola de Computação, São Paulo (1990).

[GY] R. L. Graham e F. F. Yao, Finding the convex hull of a simple polygon, *J. Algorithms* 4(4), 324–331 (1983).

[Ha] F. Harary, *Graph Theory*, Addison-Wesley, Reading, Mass. (1969).

[He] M. Henle, *A Combinatorial Introduction to Topology*, W. H. Freeman, San Francisco (1979).

[HM] S. Hertel e K. Melhorn, Fast triangulation of a simple polygon, Proc. Conf. Found. Comput. Theory, New York, *Lecture Notes on Computer Science* 158, 207–218 (1983).

[Ho] C. A. R. Hoare, Quicksort, *Computer Journal* 5, 10–15 (1962).

[J] R. A. Jarvis, On the identification of the convex hull of a finite set of points in the plane, *Info. Proc. Letters* 2, 18–21 (1973).

[Ma] M. Mäntylä, *An Introduction to Solid Modeling*, Comp. Science Press, Rockville (1988).

[Me] K. Melhorn, *Data Structures and Algorithms (vols. 1 e 3)*, Springer-Verlag, New York (1984).

[PH] F. P. Preparata e S. J. Hong, Convex hulls of finite sets of points in two and three dimensions, *Comm. ACM* 2(20), 87–93 (1977).

[PS] F. P. Preparata e M. I. Shamos, *Computational Geometry: an Introduction*, Springer-Verlag, New York (1985).

[R] R. T. Rockafellar, *Convex Analysis*, Princeton University Press, Princeton (1970).

[Se] R. Sedgewick, *Algorithms*, Addison-Wesley, Reading, Mass. (1988).

[Sh] M. I. Shamos, *Computational Geometry*, Tese de doutorado, Dept. of Comp. Science, Yale University (1978).

[Si] R. Sibson, Locally equiangular triangulations, *Computer Journal* **21**, 243–245 (1978).

[T] G. Toussaint, Pattern Recognition and Computational Geometry, *Proc. 5th Int. Conf. Pattern Recognition*, 1324–1347 (1980).

[TW] R. E. Tarjan e C. J. van Wyk, An O(n log log n)-time algorithm for triangulating a simple polygon, *SIAM J. Comput.* **17**, 143–178 (1988).

[We] K. Weiler, *Topological Structures for Geometric Modeling*, Tese de Doutorado, Rensselaer Polytechnic Institute, Troy, NY (1986).

Impresso na Gráfica do

impa

pelo Sistema Xerox / 5390