Capítulo 7

Convite à Geometria Computacional

Cristina G. Fernandes, José Coelho de Pina

Abstract

This text is a modest invitation to study computational geometry. We address four classical problems: closest pair, convex hull, segments intersection and polygon division. The closest pair problem consists of, given points in the plane, to find a closest pair of them. We present an elegant divide-and-conquer algorithm for this problem. There are several algorithms that find the convex hull of a set of points in the plane. We present four of them: an incremental algorithm, the gift wrapping, Graham scan and Quickhull. These algorithms are based on a variety of techniques, and show how a fundamental problem can be solved in several different ways. We apply the successful sweepline method to find segments intersection and to construct a certain polygon division.

Resumo

Este texto é um convite modesto ao estudo de geometria computacional, feito através de problemas clássicos: par mais próximo, fecho convexo, interseção de segmentos e divisão de polígono. No problema do par mais próximo, são dados pontos no plano, e deseja-se determinar um par mais próximo destes pontos. Apresentamos um elegante algoritmo de divisão e conquista para este problema. Existem vários algoritmos que determinam o fecho convexo de um conjunto de pontos no plano. Apresentamos quatro deles: um algoritmo incremental, o embrulho de presente, o de Graham e o Quickhull. Estes algoritmos usam-se de técnicas bem diferentes, e mostram como um problema fundamental pode ser atacado de diversas maneiras. O bem-sucedido método da linha de varredura é apresentado usando dois problemas: interseção de segmentos e divisão de polígono.

7.1. Introdução

A procura por algoritmos para resolver problemas geométricos vem desde a época da antiguidade. Algumas motivações práticas para a busca por tais algoritmos foram os impostos sobre o uso da terra e construções de edificações. São bem-conhecidas as construções geométricas de Euclides, que usavam como instrumentos régua e compasso e consistiam de algumas operações que podiam ser realizadas com esses instrumentos. Um problema clássico de construção geométrica através de régua e compasso é o chamado *Problema de Apollonius* (cerca de 200 A.C.), no qual três circunferências arbitrárias no

plano eram dadas e pedia-se uma quarta circunferência que fosse tangente às três circunferências dadas. Euclides apresentou um algoritmo que resolve este problema.

Dentre todos os problemas de construção geométrica usando as operações de Euclides, um que atraiu grande atenção foi o problema da construção de um polígono regular de n lados. Para $n = 3, 4, 5, 6$, a solução é conhecida desde a antiguidade. Entretanto, para heptágonos regulares, o problema não tem solução: aos 17 anos, Carl Friedrich Gauss (1777-1855) mostrou que *não* existe um algoritmo que, usando somente as operações de Euclides, constrói um heptágono regular. Gauss na realidade mostrou que existe um algoritmo para construir um polígono regular com p lados, para p primo, se e somente se p é um número de Fermat, ou seja, é da forma $2^{2^k} + 1$ para algum inteiro k não-negativo.

Em 1902, Emile Lemoine introduziu uma medida de *simplicidade* para os algoritmos que usam as operações de Euclides [Preparata and Shamos 1985]. Esta medida é baseada no número destas operações realizadas pelo algoritmo. Para Lemoine, o algoritmo mais simples é aquele que faz menos operações. A solução de Euclides para o Problema de Apollonius requer 508 operações enquanto que um algoritmo proposto por Lemoine requer menos de duzentas. Estava portanto introduzido em geometria um conceito que é, pelo menos em essência, o que hoje chamamos de complexidade de um algoritmo.

Em geometria computacional estamos interessados em projetar algoritmos eficientes para resolver problemas geométricos. Pelo que foi exposto acima, vemos que não é algo novo. A diferença é que as construções usam um instrumento diferente da régua e do compasso: usam um computador. Um pouco mais precisamente, em geometria computacional, estamos interessados em encontrar algoritmos eficientes, ou procedimentos computacionais, para resolver problemas geométricos. Muitos desses problemas têm sua origem em outras áreas, como computação gráfica, robótica e processamento de imagens. No projeto de tais algoritmos, são comumente utilizados resultados de geometria euclidiana, combinatória, teoria dos grafos, estruturas de dados e análise de algoritmos.

Geometria computacional é um termo usado por diversos grupos. Entretanto, o termo tem sido mais utilizado para descrever a subárea da teoria de algoritmos que trata do projeto e análise de algoritmos eficientes para problemas envolvendo objetos geométricos, principalmente, em espaços de dimensão 2, 3 ou, de uma maneira mais geral, de dimensão fixa. As entradas para os problemas são primordialmente objetos simples: pontos, retas, segmentos de retas, polígonos, planos e poliedros. É neste sentido que empregamos o termo geometria computacional neste curso.

Se a tese de doutorado de Michael Ian Shamos (1978) for aceita como o início da geometria computacional, pelo menos da maneira como ela será tratada aqui, então a área tem apenas cerca de 30 anos. Ela desenvolveu-se rapidamente nas décadas de 80 e 90, e continua a se desenvolver. Por causa

da área a partir da qual cresceu, algoritmos combinatórios, geometria computacional tem sempre enfatizado problemas de natureza matemática discreta. Na maioria dos problemas em geometria computacional, as instâncias são um conjunto finito de pontos ou de outros objetos geométricos, e a resposta é algum tipo de estrutura descrita por um conjunto finito de pontos ou segmentos de retas.

De acordo com Joseph O'Rourke, nem todos os problemas em aberto em geometria computacional são necessariamente difíceis; alguns estão simplesmente esperando a devida atenção [O'Rourke 1993]. Este pode ser um bom motivo para investigarmos problemas desta área.

Na próxima seção, especificamos o modelo de computação e a notação usada nas análises dos algoritmos. Na Seção 7.3, apresentamos o problema do par mais próximo, e alguns algoritmos para ele. Na Seção 7.4, apresentamos o clássico problema do fecho convexo de um conjunto de pontos no plano, e vários algoritmos para resolvê-lo. Na Seção 7.5, apresentamos o método da linha de varredura, clássica ferramenta para resolução de problemas em geometria computacional, e o aplicamos a um problema de interseção de segmentos. Na última seção, apresentamos um algoritmo para o problema de divisão de um polígono em polígonos monótonos, que também usa o método da linha de varredura.

Os problemas e algoritmos discutidos neste texto são tratados em vários livros [Cormen et al. 2001, de Berg et al. 1997, Kleinberg and Tardos 2006, Laszlo 1996, O'Rourke 1993, Preparata and Shamos 1985], alguns por autores brasileiros [Figueiredo and Carvalho 1991, de Rezende and Stolfi 1994].

7.2. Modelo de computação

Um *algoritmo* é uma sequência finita de instruções ou operações que resolve um problema. Um *modelo de computação* é uma descrição abstrata e conceitual, não necessariamente realista, de um computador que será usado para executar um algoritmo. Nele são especificadas as *operações elementares* que um algoritmo pode executar e o critério empregado para medir a quantidade de tempo que cada operação consome. Exemplos de operações elementares típicas são operações aritméticas entre números e comparações.

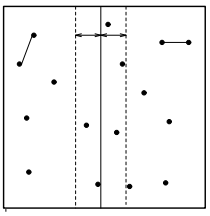
Um modelo de computação envolve um compromisso entre realidade e tratabilidade matemática e deve, portanto, capturar as características de um computador e ainda ser suficientemente simples para permitir uma análise completa dos algoritmos. O modelo de computação que é comumente usado em geometria computacional é o *real-RAM* (*real random access machine*) com custos uniformes [Preparata and Shamos 1985, Aho et al. 1974]. Esse modelo é capaz de manipular números reais arbitrários. Supõe-se que cada operação envolvendo números reais consome apenas uma unidade de tempo, mesmo uma operação como raiz quadrada.

Ao analisarmos um algoritmo em um determinado modelo de computação, queremos estimar o seu consumo de tempo como uma função do tamanho da

instância do problema. Nesta estimativa, ignoramos os fatores constantes e consideramos apenas o comportamento assintótico da função. Neste capítulo, empregamos a notação apresentada no capítulo 1 *Análise de Algoritmos* deste livro, bem como as ferramentas matemáticas lá discutidas.

Do ponto de vista prático, programas de geometria computacional lidam com cálculos numéricos que, ao contrário de métodos analíticos, fornecem soluções aproximadas e não exatas. A diferença entre tais soluções e os valores exatos é chamada de erro. Numa implementação, deve-se levar em consideração vários tipos de erros possíveis, como por exemplo erros de arredondamento devido à precisão adotada [de Berg et al. 1997, Sec. 1.2 e 1.4]. No modelo real-RAM, esses erros não existem, pois este manipula números reais arbitrários. Entretanto, em toda implementação de algum dos algoritmos aqui apresentados, esse ponto não deve ser ignorado [de Figueiredo and Stolfi 1997, Guibas et al. 1989].

7.3. Problema do par mais próximo



Esta seção estuda alguns algoritmos para um problema de proximidade. Trata-se do problema do par mais próximo (*closest pair problem*), onde queremos encontrar dois pontos de uma coleção dada tais que a distância entre eles seja a menor possível. Este problema tem aplicações em controle de tráfego aéreo ou marítimo [Cormen et al. 2001], onde pode-se querer saber quais são os objetos mais próximos para se detectar potenciais colisões.

7.3.1. O problema

Os pontos dados podem estar em um espaço de dimensão arbitrária, mas neste texto discutiremos o caso em que os pontos estão no plano e a distância entre eles é a euclidiana. Assim, se (x,y) e (x',y') são (as coordenadas de) dois pontos do plano, a distância entre eles, denotada por $\textsc{Dist}(x,y,x',y')$, é o número $\left((x-x')^2+(y-y')^2\right)^{1/2}$.

> **Problema do par mais próximo.** Dada uma coleção P de pontos do plano, encontrar dois pontos (distintos) de P tais que a distância entre eles seja a menor possível.

Para simplificar a exposição, os algoritmos descritos devolvem apenas a menor distância entre dois pontos de P. Se $|P|=1$, convencionamos que essa distância é infinita. É fácil modificar os algoritmos para que resolvam o problema acima (Exercício 2).

Uma coleção de n pontos do plano é representada por dois vetores de números, $X[1..n]$ e $Y[1..n]$. Especificamente, os pontos da coleção são $(X[1],Y[1]),\ldots,(X[n],Y[n])$.

7.3.2. Algoritmo elementar

A primeira ideia para a resolução do problema do par mais próximo é simplesmente calcularmos a distância entre cada par de pontos, enquanto mantemos a menor distância encontrada até o momento. O algoritmo abaixo, que implementa essa ideia, recebe dois vetores $X[1..n]$ e $Y[1..n]$ e devolve a menor distância entre dois pontos da coleção representada por $X[1..n], Y[1..n]$.

```
ELEMENTAR(X,Y,n)
1  d ← +∞
2  para i ← 2 até n faça
3      para j ← 1 até i−1 faça
4          se DIST(X[i],Y[i],X[j],Y[j]) < d
5              então d ← DIST(X[i],Y[i],X[j],Y[j])
6  devolva d
```

O algoritmo está correto. Para mostrar que o algoritmo está correto, basta verificar que a cada passagem pela linha 2, imediatamente antes da comparação de i com n,

> d é a menor distância entre dois pontos da coleção representada por $X[1..i-1], Y[1..i-1]$.

Este invariante mostra que o algoritmo está correto, pois na última passagem pela linha 2 temos $i = n+1$.

Consumo de tempo. Observe que o número de vezes que a linha 4 é executada é

$$\sum_{i=2}^{n}(i-1) \;=\; \sum_{i=1}^{n-1} i \;=\; \frac{n(n-1)}{2}$$

e esta função está em $\Theta(n^2)$. Já a linha 5 é executada $\mathrm{O}(n^2)$ vezes. Assim sendo, o algoritmo consome tempo $\Theta(n^2)$.

É possível projetar um algoritmo mais eficiente que este? Uma ideia que algumas vezes funciona em geometria computacional é considerar o problema em um espaço de dimensão menor e encontrar um algoritmo mais rápido que possa ser estendido para espaços de dimensão maior. Vamos então fazer um desvio, e pensar no problema quando os pontos dados estão em uma reta.

7.3.3. Par mais próximo na reta

Quando os pontos dados estão numa reta, podemos pensar neles simplesmente como uma coleção de números. Observe que, na reta, a distância entre dois números x e x' é $|x - x'|$. Neste caso, há um algoritmo muito simples que determina a menor distância entre dois números da coleção. Basta ordenar os números dados e determinar a menor distância entre dois consecutivos. O algoritmo a seguir recebe um vetor $X[1..n]$, representando uma coleção de pontos da reta, e devolve a menor distância entre dois deles.

```
ELEMENTARRETA(X,n)
 1  MERGESORT(X,1,n)
 2  d ← +∞
 3  para i ← 2 até n faça
 4       se X[i] − X[i−1] < d
 5            então d ← X[i] − X[i−1]
 6  devolva d
```

O algoritmo está correto. Veja o Exercício 3.

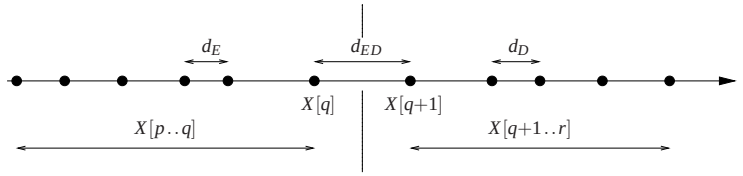
Consumo de tempo. Pela Seção Ordenação de Vetor do Capítulo 1 Análise de Algoritmos deste livro, o consumo de tempo da linha 1 é $\Theta(n \lg n)$. O número de vezes que o bloco de linhas 4-5 é executado é $n-1$. Logo, o consumo de tempo do algoritmo ELEMENTARRETA é $\Theta(n \lg n)$, que é substancialmente menor que o consumo do algoritmo ELEMENTAR.

Infelizmente não sabemos estender a ideia deste algoritmo para o plano. A seguir aplicamos a estratégia de divisão e conquista para resolver o problema do par mais próximo na reta, obtendo um algoritmo que também consome tempo $\Theta(n \lg n)$. O algoritmo obtido é um pouco mais complicado que o anterior, porém pode ser estendido para resolver o problema no plano.

O algoritmo DISTÂNCIARETAREC-SH, mostrado mais abaixo, é recursivo. Ele recebe como entrada um vetor crescente $X[p..r]$, com $p \leq r$, e devolve a menor distância entre dois números em $X[p..r]$. Assim, antes de invocá-lo, é necessário ordenar o vetor X:

```
DISTÂNCIARETA-SH(X,n)
 1  MERGESORT(X,1,n)
 2  devolva DISTÂNCIARETAREC-SH(X,1,n)
```

A estratégia é dividir o vetor $X[p..r]$ ao meio, uma metade com os números "mais à esquerda" e a outra com os números "mais à direita", resolver o problema recursivamente para os dois vetores resultantes e, de posse das soluções para estes vetores, determinar a solução para o vetor $X[p..r]$.



```
DISTÂNCIARETAREC-SH(X, p, r)
 1  se r = p
 2     então devolva +∞
 3     senão se r = p+1
 4              então devolva X[r] − X[p]
 5              senão q ← ⌊(p+r)/2⌋
 6                    d_E ← DISTÂNCIARETAREC-SH(X, p, q)
 7                    d_D ← DISTÂNCIARETAREC-SH(X, q+1, r)
 8                    d ← min{d_E, d_D, X[q+1] − X[q]}
 9                    devolva d
```

O algoritmo está correto. A menor distância entre dois números em $X[p..r]$ é a menor das distâncias entre dois números em $X[p..q]$, ou entre dois números em $X[q+1..r]$, ou entre um número em $X[p..q]$ e um em $X[q+1..r]$. Como $X[p..q]$ é crescente, a menor das distâncias entre um número em $X[p..q]$ e um em $X[q+1..r]$ é $X[q+1]-X[q]$. Desse argumento indutivo segue que o algoritmo está correto.

Consumo de tempo. O consumo de tempo de DISTÂNCIARETAREC-SH é medido em relação ao tamanho $n := r-p+1$ do vetor $X[p..r]$. Seja $T(n)$ o tempo consumido pelo algoritmo DISTÂNCIARETAREC-SH quando aplicado a um vetor X de tamanho n. O tempo consumido pelas linhas 5, 8 e 9 independe de n, ou seja, esse consumo de tempo é $\Theta(1)$. Portanto,

$$T(n) \;=\; T(\lceil \tfrac{n}{2} \rceil) + T(\lfloor \tfrac{n}{2} \rfloor) + \Theta(1). \qquad (1)$$

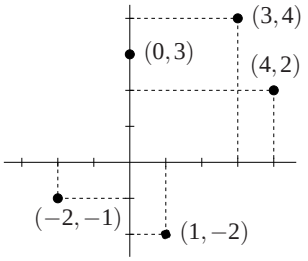
O termo $T(\lceil \frac{n}{2} \rceil)$ corresponde ao consumo de tempo da linha 6 e o termo $T(\lfloor \frac{n}{2} \rfloor)$, ao consumo da linha 7. A função $T(n)$ está em $\Theta(n)$ (Exercício 4). O algoritmo DISTÂNCIARETAREC-SH consome então tempo $\Theta(n)$. Como o MERGESORT consome tempo $\Theta(n \lg n)$ quando aplicado a um vetor de tamanho n, temos que o consumo de tempo do algoritmo DISTÂNCIARETA-SH é $\Theta(n \lg n)$.

7.3.4. Algoritmo de Shamos e Hoey

O algoritmo aqui descrito é uma generalização do algoritmo anterior para pontos do plano [Shamos and Hoey 1975]. Aqui ele é denominado de DISTÂNCIA-SH e, como o anterior, é uma mera casca para o algoritmo que faz o verdadeiro serviço, o DISTÂNCIAREC-SH.

O algoritmo DISTÂNCIAREC-SH recebe $X[p..r]$ e $Y[p..r]$, representando uma coleção de pontos do plano. Analogamente ao DISTÂNCIARETAREC-SH, o vetor X é crescente. Com isso, temos uma representação dos pontos ordenados em relação às suas coordenadas X, da esquerda para a direita.

Em uma de suas fases, DISTÂNCIAREC-SH necessita acessar os pontos em ordem de suas coordenadas Y. Por isso ele também recebe como entrada um vetor $a[p..r]$ com uma permutação de $[p..r]$ tal que $Y[a[p]] \le Y[a[p+1]] \le \cdots \le Y[a[r]]$. Isso nos fornece uma representação dos pontos ordenados em relação às suas coordenadas Y, de baixo para cima:



	1	2	3	4	5
X	−2	0	1	3	4
Y	−1	3	−2	4	2
a	3	1	5	2	4

Dizemos que X,Y,a é uma *representação ordenada* de pontos. Já que DISTÂNCIAREC-SH(X,Y,a,p,r) recebe uma representação ordenada $X[p..r]$, $Y[p..r],a[p..r]$ de pontos, então as linhas 1-4 de DISTÂNCIA-SH meramente rearranjam os vetores X e Y e criam o vetor a para que X,Y,a seja uma tal representação.

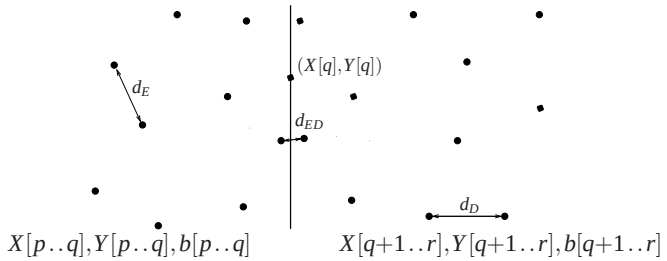
DISTÂNCIA-SH(X,Y,n)
1 MERGESORT$(X,Y,1,n)$ ▷ ordena X rearranjando Y simultaneamente
2 para $i \leftarrow 1$ até n faça
3 $\quad a[i] \leftarrow i$
4 MERGESORTIND$(Y,1,n,a)$ ▷ ordenação indireta
5 devolva DISTÂNCIAREC-SH$(X,Y,a,1,n)$

Passamos a descrever o algoritmo DISTÂNCIAREC-SH, que é recursivo. Se $r \leq p+2$ então o problema é resolvido diretamente. Caso contrário, a ideia é, em cada nível da recursão, executar as três fases a seguir:

Dividir: Seja $q := \lfloor (p+r)/2 \rfloor$. Obtenha um vetor $b[p..r]$ tal que $X[p..q],Y[p..q]$, $b[p..q]$ seja uma representação ordenada dos pontos mais à esquerda e $X[q+1..r],Y[q+1..r],b[q+1..r]$, uma representação ordenada dos pontos mais à direita.

Conquistar: Determine, recursivamente, a menor distância d_E entre dois pontos da esquerda e a menor distância d_D entre dois pontos da direita.

Combinar: Devolva o mínimo entre d_E, d_D e a menor distância d_{ED} entre um ponto da esquerda e um ponto da direita.



Veja o pseudocódigo a seguir. Nele, são usados dois algoritmos: DIVIDA e COMBINE. O algoritmo DIVIDA recebe uma representação ordenada $X[p..r]$, $Y[p..r], a[p..r]$ de pontos e devolve um vetor $b[p..r]$ como na descrição da fase dividir acima. O algoritmo COMBINE recebe $X[p..r], Y[p..r], a[p..r]$ e as distâncias d_E e d_D, e devolve a menor distância entre dois pontos da coleção representada por $X[p..r], Y[p..r], a[p..r]$.

```
DISTÂNCIAREC-SH(X,Y,a,p,r)
 1  se r ≤ p+2
 2     então ▷ resolva o problema diretamente
 3     senão q ← ⌊(p+r)/2⌋
 4           b ← DIVIDA(X,Y,a,p,r)
 5           d_E ← DISTÂNCIAREC-SH(X,Y,b,p,q)
 6           d_D ← DISTÂNCIAREC-SH(X,Y,b,q+1,r)
 7           devolva COMBINE(X,Y,a,p,r,d_E,d_D)
```

O algoritmo está correto. Desde que DIVIDA e COMBINE estejam corretamente implementados, DISTÂNCIAREC-SH está correto, e portanto DISTÂNCIA-SH também.

Consumo de tempo. O consumo de tempo de DISTÂNCIAREC-SH é medido em relação ao número de pontos $n := r - p + 1$. Este consumo depende de DIVIDA e COMBINE. O consumo de tempo da implementação destes exibida à frente é $\Theta(n)$. Assim, se $T(n)$ é o consumo de DISTÂNCIAREC-SH, então

$$T(n) \;=\; T(\lceil \tfrac{n}{2} \rceil) + T(\lfloor \tfrac{n}{2} \rfloor) + \Theta(n).$$

As parcelas $T(\lceil \frac{n}{2} \rceil)$ e $T(\lfloor \frac{n}{2} \rfloor)$ correspondem às linhas 5 e 6 respectivamente. Da Seção *Solução de Recorrências* do Capítulo *Análise de Algoritmos* deste livro, a função T está em $\Theta(n \lg n)$. Portanto DISTÂNCIAREC-SH consome tempo $\Theta(n \lg n)$.

No algoritmo DISTÂNCIA-SH, as linhas 1, 4 e 5 consomem tempo $\Theta(n \lg n)$ e as linhas 2 e 3 consomem $\Theta(n)$. Assim o seu consumo de tempo é $\Theta(n \lg n)$.

A seguir está uma implementação do algoritmo DIVIDA que, por simplicidade, supõe que na coleção não há dois pontos com a mesma coordenada X.

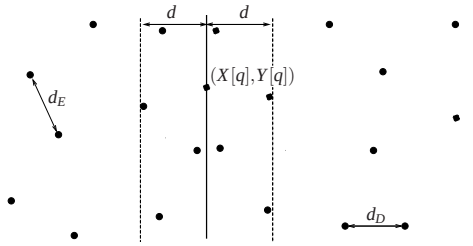
```
DIVIDA(X,Y,a,p,r)
 1  q ← ⌊(p+r)/2⌋
 2  i ← p−1    j ← q
 3  para k ← p até r faça
 4     se X[a[k]] ≤ X[q]    ▷ (X[a[k]],Y[a[k]]) está à esquerda da reta x = X[q]?
 5        então i ← i+1
 6              b[i] ← a[k]
 7        senão j ← j+1
 8              b[j] ← a[k]
 9  devolva b
```

O algoritmo DIVIDA está correto. Como X é crescente e não há pontos com a mesma coordenada X, então o número de pontos com coordenada X menor ou igual a $X[q]$ é exatamente $q-p+1$. Assim, ao final do DIVIDA, $i{=}q$ e $j{=}r$. Além disso, pelo teste da linha 4, os índices de $a[p..r]$ de pontos da esquerda estão sendo colocados em $b[p..q]$, em ordem crescente de suas coordenadas Y. Similarmente os índices de pontos da direita estão sendo colocados em $b[p..q]$, também em ordem crescente de suas coordenadas Y. Portanto, ao final, $X[p..q], Y[p..q], b[p..q]$ e $X[q{+}1..r], Y[q{+}1..r], b[q{+}1..r]$ são representações ordenadas dos pontos à esquerda e à direita respectivamente.

Consumo de tempo do algoritmo DIVIDA. É fácil ver que o DIVIDA consome tempo $\Theta(n)$, onde $n := r-p+1$.

O algoritmo COMBINE é o coração do DISTÂNCIAREC-SH. Uma implementação ingênua dele consumiria tempo quadrático: calcule a distância entre cada ponto da coleção representada por $X[p..q], Y[p..q], a[p..q]$ e cada ponto da coleção representada por $X[q{+}1..r], Y[q{+}1..r], a[p..r]$, determine d_{ED}, que é a menor destas distâncias, e devolva a menor entre d_{ED}, d_E e d_D. É preciso fazer algo mais refinado que isso.

A primeira observação importante é que não é necessário que COMBINE calcule a distância entre pontos que estão sabidamente a uma distância maior que $d := \min\{d_E, d_D\}$. Com isso, COMBINE precisa considerar apenas pontos que estão a uma distância menor que d da reta vertical $x = X[q]$.



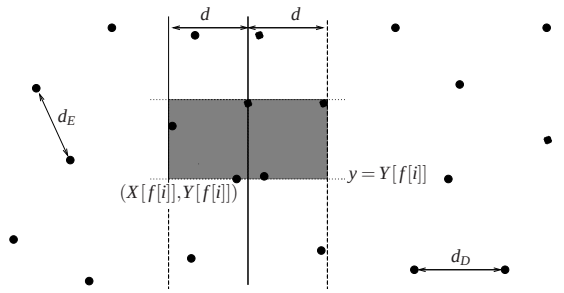
É fácil determinar quais pontos da coleção estão a uma distância menor que d da reta $x = X[q]$. Isso é feito pelo algoritmo CANDIDATOS, que recebe como parâmetros $X[p..r], a[p..r]$, parte de uma representação ordenada X, Y, a, e recebe também um número positivo d. O algoritmo CANDIDATOS devolve um vetor $f[1..t]$ indicando os pontos da coleção representada por X, Y, a que estão a uma distância menor que d da reta $x = X[q]$ em ordem de suas coordenadas Y.

```
CANDIDATOS (X, a, p, r, d)
 1  q ← ⌊(p+r)/2⌋    t ← 0
 2  para k ← p até r faça
 3      se |X[a[k]] − X[q]| < d
 4          então t ← t+1    f[t] ← a[k]
 5  devolva (f, t)
```

Chamemos de F a coleção dos pontos indicados pelo vetor f. Quantos pontos há em F? No pior caso, há n pontos em F. Logo, se calcularmos a distância entre quaisquer dois pontos de F, no pior caso calcularíamos $\Theta(n^2)$ distâncias nesta fase. Isso resultaria de novo em um consumo de tempo de $\Theta(n^2)$ para o COMBINE, o que não é suficiente.

A segunda observação crucial é que qualquer quadrado de lado d totalmente contido em um dos lados da reta $x = X[q]$ contém no máximo 4 pontos da coleção. Do contrário, haveria um par de pontos em um dos lados a uma distância menor que d.

Para cada ponto em F, ou seja, para cada i com $1 \le i \le t$, considere a reta horizontal $y = Y[f[i]]$ que passa pelo ponto $(X[f[i]], Y[f[i]])$. Considere os dois quadrados de lado d que têm sua base nesta reta: um com seu lado direito na reta $x = X[q]$ e o outro com seu lado esquerdo na reta $x = X[q]$.



Há no máximo 8 pontos da coleção nestes quadrados, incluindo o ponto $(X[f[i]], Y[f[i]])$. Dentre os pontos de F acima da reta $y = Y[f[i]]$, apenas pontos nestes quadrados têm chance de estar a uma distância de $(X[f[i]], Y[f[i]])$ menor do que d. Assim basta comparar $(X[f[i]], Y[f[i]])$ com os 7 próximos pontos de F, em ordem de suas coordenadas Y.

```
COMBINE (X, Y, a, p, r, d_E, d_D)
 1  d ← min{d_E, d_D}
 2  (f, t) ← CANDIDATOS (X, a, p, r, d)
 3  para i ← 1 até t − 1 faça
 4      para j ← i + 1 até min{i + 7, t} faça
 5          d′ ← DIST(X[f[i]], Y[f[i]], X[f[j]], Y[f[j]])
 6          se d′ < d
 7              então d ← d′
 8  devolva d
```

O algoritmo COMBINE está correto. Conforme discussão prévia, os pares de pontos para os quais a distância não foi calculada na linha 5 estão à distância pelo menos $\min\{d_E, d_D\}$.

Consumo de tempo do algoritmo COMBINE. O consumo de tempo do algoritmo COMBINE também é medido em relação ao número $n := r - p + 1$ de

pontos. CANDIDATOS consome tempo $\Theta(n)$ e o bloco de linhas 5-7 é executado não mais do que $7n$ vezes. Portanto o consumo de tempo do COMBINE é $\Theta(n)$.

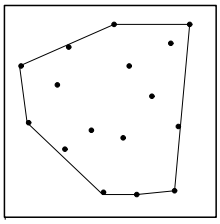
Com isso, concluímos a descrição do DISTÂNCIAREC-SH, pois mostramos os algoritmos DIVIDA e COMBINE. Da correção e do consumo de tempo de DIVIDA e COMBINE, segue, como já vimos, que o DISTÂNCIAREC-SH está correto e consome tempo $\Theta(n \lg n)$.

Exercícios

1. Considere o espaço euclidiano 3-dimensional (o $\mathbb{R}^3$, com a distância definida da maneira usual). Este espaço é usualmente denotado por $\mathbb{E}^3$. Ajuste o algoritmo ELEMENTAR para que ele aceite como dados uma coleção de pontos do $\mathbb{E}^3$.
2. Modifique os algoritmos ELEMENTAR e DISTÂNCIA-SH para que eles devolvam as coordenadas de dois pontos da coleção tais que a distância entre eles seja a menor possível.
3. Encontre um invariante apropriado que torne evidente a correção do algoritmo ELEMENTARRETA.
4. Seja F a função definida sobre os inteiros positivos satisfazendo $F(1) = 1$ e
$$F(n) \;=\; F(\lfloor \tfrac{n}{2} \rfloor) + F(\lceil \tfrac{n}{2} \rceil) + 1.$$
Demonstre por indução em n que $n - 1 \leq F(n) \leq 2n - 1$. Conclua que o consumo de tempo do DISTÂNCIARETAREC-SH, dado pela função $T(n)$ que satisfaz (1), está em $\Theta(n)$.
5. Simule a execução do algoritmo DIVIDA para a representação ordenada X, Y, a dada como exemplo na página 8.
6. Modifique o algoritmo DIVIDA para que ele funcione quando a coleção dada tem pontos com mesma coordenada X.
7. É possível trocar o número 7 na linha 8 do algoritmo COMBINE por um valor menor? Qual? Encontre um exemplo que mostra que o número que você escolheu não pode ser diminuído
8. A construção do vetor f é necessária no COMBINE? O algoritmo COMBINEESPERTO abaixo, que promete fazer o mesmo serviço que COMBINE, evita essa construção. Dê um exemplo que mostra que COMBINEESPERTO está errado.

```
COMBINEESPERTO (X, Y, a, p, r, d_E, d_D)
 1  d ← min{d_E, d_D}
 2  para i ← p até r − 1 faça
 3      para j ← i + 1 até min{i + 7, r} faça
 4          d' ← DIST(X[a[i]], Y[a[i]], X[a[j]], Y[a[j]])
 5          se d' < d
 6              então d ← d'
 7  devolva d
```

7.4. Fecho convexo de um conjunto de pontos



Dados pontos no plano, queremos encontrar o fecho convexo desses pontos. Uma aplicação deste problema se encontra em robótica, onde se deseja evitar que um robô em movimento colida com outros objetos [O'Rourke 1993].

Nos anos 60 uma aplicação da Bell Labs necessitava computar o fecho convexo de aproximadamente 10.000 pontos no plano e o consumo de tempo dos algoritmos disponíveis era $\Theta(n^2)$, onde n é o número de pontos dados. Tendo esse problema como motivação, foi projetado o primeiro algoritmo com consumo de tempo $O(n \lg n)$ [Graham 1972]. Aqui estão descritos vários algoritmos para esse problema.

7.4.1. O problema

Uma *combinação convexa* de uma coleção de pontos do plano representada por $X[1..n], Y[1..n]$ é uma soma da forma

$$\alpha_1(X[1], Y[1]) + \cdots + \alpha_n(X[n], Y[n]),$$

com $\alpha_i \geq 0$, para $i = 1, \ldots, k$, e $\alpha_1 + \cdots + \alpha_n = 1$.

O *fecho convexo* de uma coleção P de pontos do plano representada por $X[1..n], Y[1..n]$ são os pontos do plano que são combinação convexa de pontos de P, ou seja,

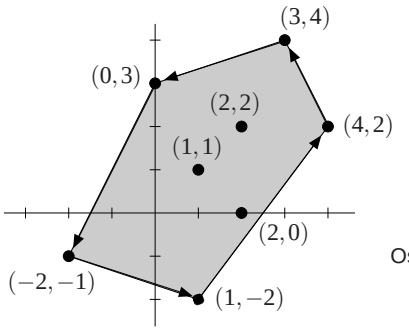
$$\text{conv}(P) := \{\alpha_1(X[1], Y[1]) + \cdots + \alpha_n(X[n], Y[n]) : \alpha_1 + \cdots + \alpha_n = 1, \text{ e } \alpha_i \geq 0 \ (i = 1, \ldots, n)\}.$$

> **Problema do fecho convexo.** Dada uma coleção P de pontos do plano, determinar o fecho convexo de P.

O próximo passo é definir como representar nos algoritmos o fecho convexo de uma coleção de pontos. A representação natural é pela coleção dos pontos "extremos" do fecho convexo.

Seja P uma coleção de pontos do plano. Um ponto (x, y) de P é *extremo* se (x, y) não é combinação convexa de pontos de $P - \{(x, y)\}$. Os pontos extremos da coleção P são os mesmos que os de $\text{conv}(P)$.

Os algoritmos que apresentamos a seguir recebem $X[1..n]$ e $Y[1..n]$, representando uma coleção de pontos do plano, e devolvem um vetor $H[1..h]$ com os índices dos pontos extremos do fecho convexo da coleção na ordem em que aparecem na fronteira do fecho convexo quando a percorremos no sentido anti-horário. O vetor $H[1..h]$ é uma *descrição combinatória* do fecho convexo da coleção representada por $X[1..n], Y[1..n]$.



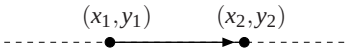
	1	2	3	4	5	6	7	8
X	1	3	2	−2	1	2	4	0
Y	1	4	0	−1	−2	2	2	3
H	2	8	4	5	7			

Os pontos de índice 2, 4, 5, 7 e 8 são extremos.

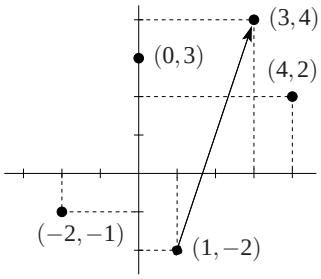
Uma coleção de pontos no plano está em *posição geral* se ela não possui três pontos colineares. Para simplificar, os algoritmos apresentados supõem que a coleção de pontos dada está em posição geral.

7.4.2. Predicados geométricos

Dados dois pontos (x_1,y_1) e (x_2,y_2), chamamos de *reta orientada* determinada por (x_1,y_1) e (x_2,y_2) a reta que passa por (x_1,y_1) e (x_2,y_2) e tem a orientação de (x_1,y_1) para (x_2,y_2).



Os predicados geométricos usados nos algoritmos desta seção são o ESQUERDA e o DIREITA. Dados três pontos (x_1,y_1), (x_2,y_2) e (x_3,y_3), o predicado ESQUERDA devolve VERDADEIRO se (x_3,y_3) está à esquerda da reta orientada determinada por (x_1,y_1) e (x_2,y_2), e FALSO caso contrário. Já o predicado DIREITA devolve VERDADEIRO se (x_3,y_3) está à direita da reta orientada determinada por (x_1,y_1) e (x_2,y_2), e FALSO caso contrário.



ESQUERDA$((1,-2),(3,4),(0,3))$ = VERDADEIRO
ESQUERDA$((1,-2),(3,4),(4,2))$ = FALSO
ESQUERDA$((3,4),(1,-2),(4,2))$ = VERDADEIRO
DIREITA$((1,-2),(3,4),(0,3))$ = FALSO
DIREITA$((1,-2),(3,4),(4,2))$ = VERDADEIRO
DIREITA$((3,4),(1,-2),(4,2))$ = FALSO

O valor de ESQUERDA$((x_1,y_1),(x_2,y_2),(x_3,y_3))$ é dado pelo sinal do determinante [Figueiredo and Carvalho 1991, Cap. 2]

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

bem como o valor de DIREITA$((x_1,y_1),(x_2,y_2),(x_3,y_3))$.

O valor deste determinante é $(x_2-x_1)(y_3-y_1)-(x_3-x_1)(y_2-y_1)$, e o denotamos por DET$(x_1,y_1,x_2,y_2,x_3,y_3)$. O seu valor absoluto é duas vezes a área do triângulo de extremos (x_1,y_1), (x_2,y_2) e (x_3,y_3).

ESQUERDA$((x_1,y_1),(x_2,y_2),(x_3,y_3))$ é VERDADEIRO se e somente se esse determinante é positivo, e DIREITA$((x_1,y_1),(x_2,y_2),(x_3,y_3))$ é VERDADEIRO se e somente se ele é negativo [Laszlo 1996]. Note que DET, ESQUERDA e DIREITA consomem tempo $\Theta(1)$.

Para que a apresentação seja mais leve, usamos uma variante do algoritmo ESQUERDA, que denotamos por ESQ, que recebe uma coleção de pontos do plano representada por $X[1..n],Y[1..n]$ e três índices, i, j e k entre 1 e n. Da mesma forma, temos um algoritmo DIR.

```
ESQ(X,Y,i,j,k)
 1  devolva ESQUERDA((X[i],Y[i]),(X[j],Y[j]),(X[k],Y[k]))

DIR(X,Y,i,j,k)
 1  devolva DIREITA((X[i],Y[i]),(X[j],Y[j]),(X[k],Y[k]))
```

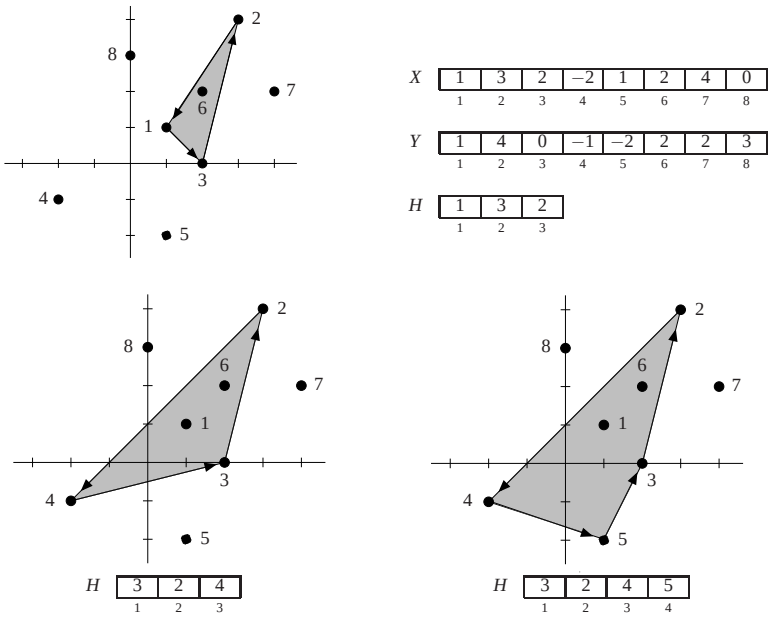
7.4.3. Um algoritmo incremental

Este primeiro algoritmo é iterativo e examina um ponto da coleção após o outro. Ele mantém o fecho convexo dos pontos já examinados. A cada iteração, de posse do fecho convexo corrente, ele constrói o fecho convexo incluindo o próximo ponto da coleção. O pseudocódigo a seguir, que implementa essa ideia, recebe $X[1..n]$ e $Y[1..n]$, representando uma coleção de pelo menos três pontos em posição geral, e devolve uma descrição combinatória $H[1..h]$ do fecho convexo da coleção.

As linhas de 1-3 do algoritmo INCREMENTAL constroem o fecho convexo dos três primeiros pontos da coleção. O algoritmo PERTENCE recebe a descrição combinatória $H[1..h]$ do fecho convexo de um subconjunto da coleção de pontos representada por X,Y e um ponto (x,y), e devolve VERDADEIRO se (x,y) está neste fecho e FALSO caso contrário. O algoritmo INSEREPONTO recebe a descrição combinatória $H[1..h]$ do fecho convexo da coleção representada por $X[1..k-1],Y[1..k-1]$ e devolve o fecho convexo da coleção representada por $X[1..k],Y[1..k]$.

```
INCREMENTAL(X,Y,n)
 1  se ESQ(X,Y,1,2,3)
 2     então H[1] ← 1   H[2] ← 2   H[3] ← 3   h ← 3
 3     senão H[1] ← 1   H[2] ← 3   H[3] ← 2   h ← 3
 4  para k ← 4 até n faça
 5     se não PERTENCE(H,h,X,Y,X[k],Y[k])
 6        então (H,h) ← INSEREPONTO(H,h,X,Y,k)
 7  devolva (H,h)
```

Abaixo está uma simulação das primeiras iterações do algoritmo.



O algoritmo está correto. Desde que PERTENCE e INSEREPONTO estejam corretamente implementados, na linha 4, imediatamente antes de k ser comparado com n,

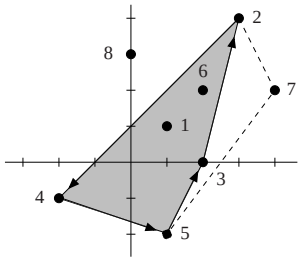
> $H[1..h]$ é uma descrição combinatória do fecho convexo da coleção representada por $X[1..k-1], Y[1..k-1]$.

Portanto INCREMENTAL está correto.

Consumo de tempo. O consumo de tempo do INCREMENTAL depende de PERTENCE e INSEREPONTO. As implementações destes que apresentamos à frente consomem tempo $\Theta(n)$ e $\mathrm{O}(h)$ respectivamente. Como $h \leq n$, temos que INCREMENTAL consome tempo $O(n^2)$. Se todos os n pontos são extremos do fecho convexo, INCREMENTAL consome tempo $\Theta(n^2)$.

A seguir, está uma implementação do algoritmo PERTENCE.

```
PERTENCE(H,h,X,Y,x,y)
1  H[h+1] ← H[1]    ▷ sentinela
2  para i ← 1 até h faça
3      se DIREITA(X[H[i]],Y[H[i]],X[H[i+1]],Y[H[i+1]],x,y)
4          então devolva FALSO
5  devolva VERDADEIRO
```



X	1	3	2	-2	1	2	4	0
	1	2	3	4	5	6	7	8
Y	1	4	0	-1	-2	2	2	3
	1	2	3	4	5	6	7	8

H	3	2	4	5
	1	2	3	4

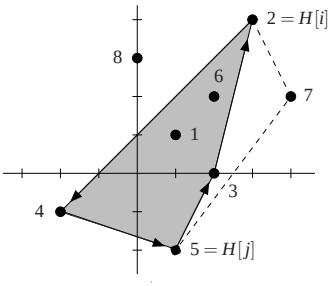
PERTENCE$(H,4,X,Y,2,2)$ = VERDADEIRO
PERTENCE$(H,4,X,Y,4,2)$ = FALSO

O algoritmo PERTENCE está correto. A correção do algoritmo é devida à convexidade. Se um ponto não pertence ao fecho convexo, então ele está à direita de alguma de suas 'arestas'.

Consumo de tempo do PERTENCE. Este consumo é $\Theta(h)$.

Agora, passemos à descrição do algoritmo INSEREPONTO. Como $(X[k],Y[k])$ não pertence ao fecho corrente, ele é ponto extremo do próximo fecho. Então, para atualizar o vetor H, precisamos determinar o sucessor e o predecessor de $(X[k],Y[k])$ na fronteira do próximo fecho.

Considere que $H[0] = H[h]$ e $H[h+1] = H[1]$. Seja i entre 1 e h tal que $(X[k],Y[k])$ fica à direita do vetor que vai do ponto indicado por $H[i-1]$ para o indicado por $H[i]$ e à esquerda do vetor que vai do ponto indicado por $H[i]$ para o ponto indicado por $H[i+1]$. O ponto de índice $H[i]$ é o sucessor de $(X[k],Y[k])$. Seja j definido similarmente com esquerda e direita trocadas. O ponto de índice $H[j]$ é o predecessor de $(X[k],Y[k])$. Com isso, o resultado de INSEREPONTO(H,h,X,Y,k) é o vetor F com o trecho de H de i até j 'circularmente', acrescido de k.



X	1	3	2	-2	1	2	4	0
	1	2	3	4	5	6	7	8
Y	1	4	0	-1	-2	2	2	3
	1	2	3	4	5	6	7	8

H	3	2	4	5
	1	2	3	4

$i = 2$ e $j = 4$ na linha 9. Ao final

F	2	4	5	7
	1	2	3	4

No bloco de linhas 1-8 do algoritmo INSEREPONTO, determinamos os índices i e j. No bloco de linhas 9-12, transcrevemos para F o trecho do fecho $H[i\,..\,j]$ circularmente e acrescentamos k.

```
INSEREPONTO(H,h,X,Y,k)
 1  H[0] ← H[h]    H[h+1] ← H[1]    ▷ sentinelas
 2  i ← 1
 3  enquanto ESQ(X,Y,H[i−1],H[i],k) = ESQ(X,Y,H[i],H[i+1],k) faça
 4      i ← i+1
 5  j ← i+1
 6  enquanto ESQ(X,Y,H[j−1],H[j],k) = ESQ(X,Y,H[j],H[j+1],k) faça
 7      j ← j+1
 8  se ESQ(X,Y,H[i−1],H[i],k) então i ↔ j
 9  t ← 1
10  enquanto i ≠ j faça
11      F[t] ← H[i]    t ← t+1    i ← (i mod h)+1
12  F[t] ← H[i]    t ← t+1    F[t] ← k
13  devolva (F,t)
```

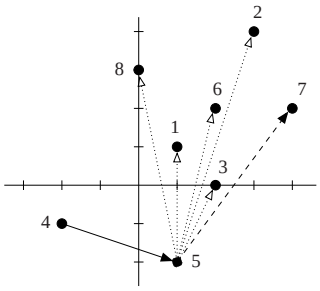
O algoritmo está correto. O ponto $(X[k],Y[k])$ não está no fecho e os pontos estão em posição geral. Assim, existem exatamente dois índices ℓ tais que $1 \leq \ell \leq h$ e $\text{ESQ}(X,Y,H[\ell-1],H[\ell],k) \neq \text{ESQ}(X,Y,H[\ell],H[\ell+1],k)$. Estes índices são o i e o j mencionados anteriormente, e são determinados pelo bloco de linhas 1-8. A linha 8 garante que $H[i]$ é o sucessor de k no sentido anti-horário na fronteira do novo fecho.

Consumo de tempo. O número de vezes que a linha 4 e a linha 7 são executadas no total é no máximo h. Também a linha 11 é executada no máximo h vezes. Assim o consumo de tempo do INSEREPONTO é $O(h)$.

7.4.4. Algoritmo do embrulho de presente

Este algoritmo é a versão para o plano de um método que determina o fecho convexo para pontos em um espaço de dimensão arbitrária [Chand and Kapur 1970, Jarvis 1973].

Intuitivamente, o algoritmo simula o enrolar da coleção de pontos por um barbante. A ideia é, a partir de um ponto extremo do fecho convexo, encontrar o próximo ponto extremo no sentido anti-horário. Ela se baseia no fato de que em uma descrição combinatória do fecho convexo temos os índices dos pontos extremos da coleção na ordem em que aparecem na fronteira do fecho convexo quando a percorremos no sentido anti-horário.



	1	2	3	4	5	6	7	8
X	1	3	2	−2	1	2	4	0
Y	1	4	0	−1	−2	2	2	3

	1	2	3	4
H	4	5	?	…

$\text{DIR}(X,Y,5,7,j) = \text{FALSO}$ para $j = 1,\ldots,8$

Considere a descrição combinatória $H[1..h]$ do fecho convexo de uma coleção de pontos $X[1..n]$ e $Y[1..n]$. Para cada k entre 1 e $h-1$, temos que $\textsc{Dir}(X,Y,H[k],H[k+1],j) = \textsc{falso}$ para $j = 1,\ldots,n$. Este fato nos fornece uma maneira para, a partir do ponto extremo $(X[H[k]],Y[H[k]])$, determinar o ponto $(X[H[k+1]],Y[H[k+1]])$. O algoritmo EMBRULHO a seguir, que se apóia nesta observação, recebe $X[1..n]$ e $Y[1..n]$, com $n \geq 2$, representando uma coleção de pontos em posição geral, e devolve uma descrição combinatória $H[1..h]$ do fecho convexo da coleção.

```
EMBRULHO(X,Y,n)
 1  h ← 0
 2  H[0] ← min{i ∈ [1..n] : X[i] ≤ X[j], 1 ≤ j ≤ n}
 3  repita
 4      i ← (H[h] mod h) + 1      ▷ qualquer ponto distinto de H[h]
 5      para j ← 1 até n faça
 6          se DIR(X,Y,H[h],i,j) então i ← j
 7      h ← h + 1
 8      H[h] ← i
 9  até que i = H[0]      ▷ fechou o polígono
10  devolva (H,h)
```

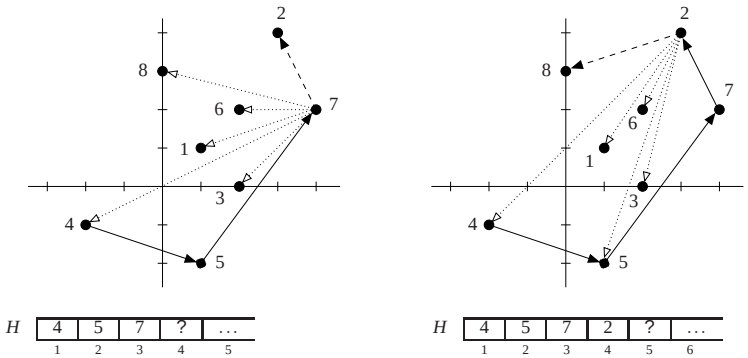
O algoritmo está correto. Como os pontos estão em posição geral, $H[0]$ calculado na linha 2 é o índice de um ponto extremo da coleção. Na linha 9, vale que

> $H[1..h]$ são os h primeiros índices que sucedem $H[0]$ numa descrição combinatória do fecho convexo de $X[1..n],Y[1..n]$.

Nas linhas 4-6, o algoritmo determina o sucessor do ponto de índice $H[h]$ na fronteira do fecho convexo no sentindo anti-horário. Quando este sucessor é o ponto de índice $H[0]$, significa que $H[1..h]$ está completo.

Consumo de tempo. Se h é o número de pontos extremos do fecho convexo dos pontos $X[1..n],Y[1..n]$, então o bloco de linhas 4-8 é executado h vezes. Para cada uma dessas h execuções, a linha 6 é executada n vezes. Portanto, o consumo de tempo do algoritmo é $\Theta(hn)$. Note que o consumo de tempo deste algoritmo depende não somente do número n de pontos dados, mas também do número de pontos h na descrição combinatória do fecho devolvida. Diz-se que um algoritmo como EMBRULHO para o qual o consumo de tempo depende do tamanho da resposta produzida é *output-sensitive*.

A seguir mostramos as duas próximas iterações de EMBRULHO.



7.4.5. Algoritmo de Graham

Segundo O'Rourke, é possível que o primeiro artigo em geometria computacional tenha sido o de Graham que apresenta um algoritmo $O(n \lg n)$ para encontrar o fecho convexo de um conjunto de n pontos no plano [Graham 1972, O'Rourke 1993].

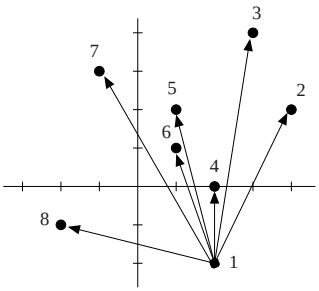
O algoritmo de Graham faz um certo pré-processamento e posteriormente é semelhante ao INCREMENTAL. Ele é iterativo, examinando um ponto da coleção após o outro, mantendo o fecho convexo dos pontos já examinados. O pré-processamento faz uma "ordenação angular" dos pontos em torno do ponto de menor coordenada Y. O efeito deste pré-processamento é que, em cada iteração, o ponto sendo examinado seja ponto extremo do novo fecho. Isso elimina a necessidade do PERTENCE. Com isso, a tarefa de cada iteração se restringe ao INSEREPONTO. Além disso, devido ao pré-processamento, o sucessor do ponto examinado é sempre o primeiro ponto que o algoritmo examinou.

O pré-processamento consiste no seguinte.

ORDENA-G(X,Y,n)
1 $i \leftarrow \min\{i \in [1..n] : Y[i] \leq Y[j], 1 \leq j \leq n\}$
2 $(X[1],Y[1]) \leftrightarrow (X[i],Y[i])$
3 MERGESORT-G$(X,Y,2,n)$

Depois da linha 2, o ponto $(X[1],Y[1])$ é extremo. Considere uma ordem total $\prec$ dos pontos $(X[2],Y[2]),\ldots,(X[n],Y[n])$ em que $(X[i],Y[i]) \prec (X[j],Y[j])$ se o ponto $(X[i],Y[i])$ está à direita da reta orientada determinada por $(X[1],Y[1])$ e $(X[j],Y[j])$. O MERGESORT-G rearranja os vetores $X[2..n]$ e $Y[2..n]$ de modo que $(X[2],Y[2]) \prec \cdots \prec (X[n],Y[n])$. Ele pode ser implementado com o MERGESORT usando como função de comparação a seguinte rotina, que recebe X, Y e dois índices i e j entre 2 e n, e devolve VERDADEIRO se o ponto $(X[i],Y[i])$ é considerado menor que $(X[j],Y[j])$:

MENOR-G(X,Y,i,j)
1 devolva DIR$(X,Y,1,j,i)$



X	1	3	2	-2	2	1	4	-1
	1	2	3	4	5	6	7	8

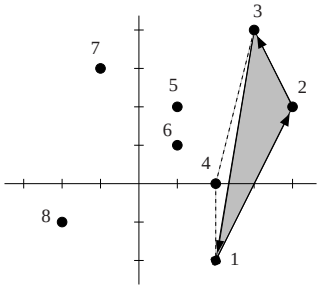
Y	1	4	0	-1	-2	2	2	3
	1	2	3	4	5	6	7	8

MENOR-G$(X,Y,2,j)$ = VERDADEIRO
para $j = 3,\ldots,8$

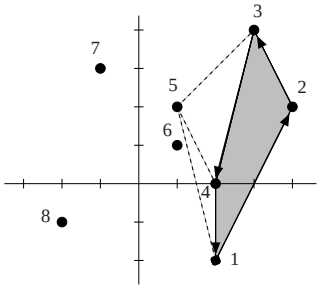
O algoritmo GRAHAM recebe $X[1..n]$ e $Y[1..n]$, com $n \geq 3$, representando uma coleção de pontos em posição geral, e devolve uma descrição combinatória $H[1..h]$ do fecho convexo da coleção.

```
GRAHAM(X,Y,n)
1  ORDENA-G(X,Y,n)
2  H[1] ← 1    H[2] ← 2    H[3] ← 3    h ← 3
3  para k ← 4 até n faça
4      j ← h
5      enquanto ESQ(X,Y,H[j],H[j−1],k) faça
6          j ← j − 1
7      h ← j + 1    H[h] ← k
8  devolva (H,h)
```

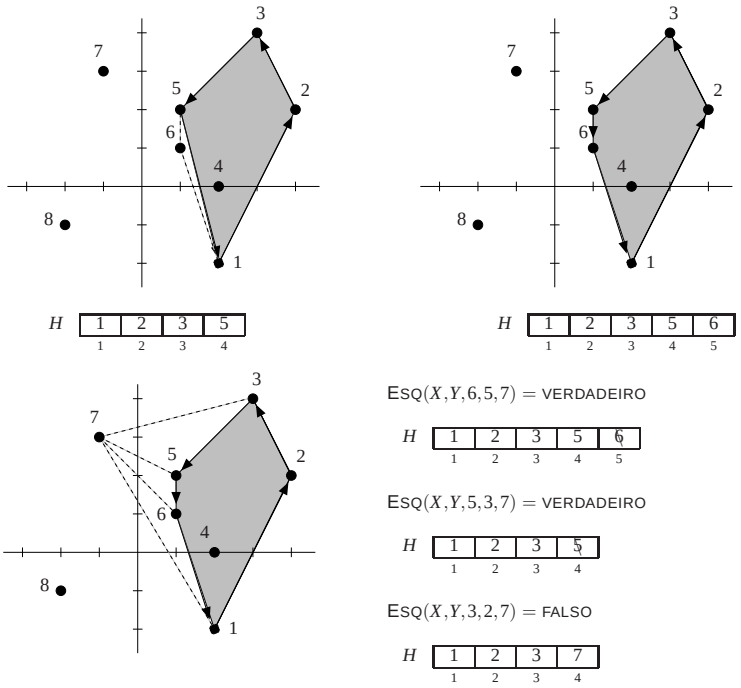
A seguir estão as primeiras iterações de GRAHAM para o exemplo anterior.



H	1	2	3
	1	2	3



H	1	2	3	4
	1	2	3	4



O algoritmo está correto. Note que o GRAHAM é uma implementação mais esperta do INCREMENTAL. Para mostrar que ele está correto, basta verificar que a cada passagem pela linha 3, imediatamente antes da comparação de k com n,

> $H[1..h]$ é uma descrição combinatória do fecho convexo da coleção representada por $X[1..k-1], Y[1..k-1]$.

Este invariante mostra que GRAHAM está correto, pois na última passagem pela linha 3 temos $k = n+1$.

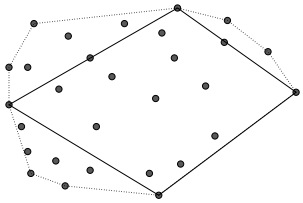
O invariante vale pela seguinte razão. Primeiro, $H[1..3]$ é uma descrição combinatória do fecho dos três primeiros pontos, já que estes estão ordenados conforme o pré-processamento. Agora suponha que $H[1..h]$ seja a descrição combinatória do fecho convexo de $X[1..k-1], Y[1..k-1]$ para $k \leq n$. Devido a ORDENA-G e à hipótese da coleção estar em posição geral, $(X[k], Y[k])$ é ponto extremo do novo fecho e $(X[1], Y[1])$ é o seu sucessor na fronteira do novo fecho convexo no sentindo anti-horário. O índice k é adicionado na posição correta de H na linha 7. Devido a ORDENA-G e à hipótese da coleção estar em posição geral, sabemos que $\text{ESQ}(X, Y, 1, H[h], k) = \text{VERDADEIRO}$. Assim, adaptando as linhas 3-8 do INCREMENTAL, o predecessor é dado pelo maior j tal que $\text{ESQ}(X, Y, H[j], H[j-1], k) = \text{FALSO}$. A tarefa de determinar esse j é feita pelas linhas 4-6.

Consumo de tempo. A linha 1 consome tempo $\Theta(n \lg n)$. Na linha 3 do algoritmo, $h \geq 3$. A linha 6 é executada $n-3$ vezes. Assim, o número de vezes que h é decrementado na linha 5 não é superior a $n-3$. Portanto o bloco de linhas 2-7 consome tempo $\Theta(n)$, e o consumo de tempo do GRAHAM é dominado pelo consumo de tempo da linha 1, que é $\Theta(n \lg n)$.

7.4.6. Quickhull

É possível tratar geometria computacional como o estudo de problemas de busca e ordenação em dimensões maiores [Mulmuley 1994]. De fato, em um certo sentido, problemas de ordenação e busca em uma lista de elementos de um universo ordenado podem ser vistos como versões unidimensionais de alguns problemas em geometria computacional em dimensões maiores. Deste ponto de vista, não é nenhuma surpresa o fato de que algoritmos de ordenação e busca sejam uma constante fonte de inspiração para o projeto de algoritmos em geometria computacional. O algoritmo apresentado nesta seção foi proposto independentemente, com algumas variações, por várias pessoas quase ao mesmo tempo [Eddy 1977, Bykat 1978, Green and Silverman 1979]. Devido à semelhança com o QUICKSORT, este algoritmo foi batizado de QUICKHULL [Preparata and Shamos 1985].

A ideia básica por trás do QUICKHULL é que, para a "maioria" dos conjuntos de pontos, é fácil descartar muitos pontos que estão no interior do fecho convexo e concentrar o trabalho nos pontos que estão próximos da fronteira.



A semelhança com o QUICKSORT vem do fato de que o QUICKHULL processa os pontos, montando duas coleções de pontos basicamente disjuntas, onde o problema é resolvido recursivamente. A partir dos fechos convexos destas duas coleções, o fecho convexo da coleção dada é obtido facilmente. A seguir, descrevemos uma rotina que chamamos de PARTICIONE, que faz esse trabalho de montar as duas coleções. Vale ressaltar que essas duas coleções não formam, como no caso do QUICKSORT, uma partição da coleção dada de pontos. A rotina detecta pontos da coleção que estão no interior do fecho, e não os inclui em nenhuma das duas coleções montadas. Antes de apresentar o PARTICIONE, exibimos duas rotinas usadas nele.

Lembre-se que DET$(x_1,y_1,x_2,y_2,x_3,y_3)$ é o valor do determinante da página 14, cujo valor absoluto é duas vezes a área do triângulo de extremos (x_1,y_1), (x_2,y_2) e (x_3,y_3). Considere a rotina

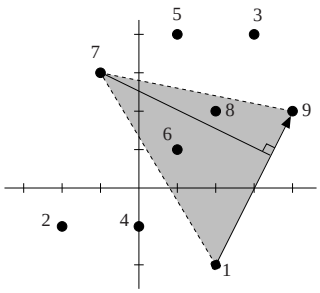
```
ÁREA (X,Y,i,j,k)
1  devolva |DET(X[i],Y[i],X[j],Y[j],X[k],Y[k])|/2
```

que devolve a área do triângulo cujos extremos são os pontos da coleção de índices i, j e k.

O algoritmo abaixo recebe uma coleção de pelo menos três pontos $X[p..r],Y[p..r]$ em posição geral e, usando ÁREA, devolve o índice de um ponto extremo da coleção distinto de p e r. Este algoritmo será usado no algoritmo PARTICIONE.

```
PONTOEXTREMO (X,Y,p,r)
1  q ← p+1    max ← ÁREA(X,Y,p,r,q)
2  para i ← p+2 até r−1 faça
3      se ÁREA(X,Y,p,r,i) > max
4          então q ← i    max ← ÁREA(X,Y,p,r,q)
5  devolva q
```

O algoritmo PONTOEXTREMO está correto. O algoritmo devolve o índice de um ponto da coleção mais distante da reta que passa por $(X[p],Y[p])$ e $(X[r],Y[r])$. Este é claramente um ponto extremo, já que a coleção está em posição geral. Um ponto mais distante da reta que passa pelos pontos $(X[p],Y[p])$ e $(X[r],Y[r])$ forma um triângulo de maior altura tendo $(X[p],Y[p])$ e $(X[r],Y[r])$ como base.



	1	2	3	4	5	6	7	8	9
X	2	−2	3	0	1	1	−1	2	4
Y	−2	−1	4	−1	4	1	3	2	2

PONTOEXTREMO$(X,Y,1,9) = 7$

Como a área de um triângulo é metade do produto do comprimento de sua base por sua altura, e os triângulos considerados têm todos a mesma base, então um ponto que forma um triângulo de maior altura é um cujo triângulo tem a maior área. PONTOEXTREMO simplesmente encontra um tal ponto.

Consumo de tempo de PONTOEXTREMO. É fácil ver que PONTOEXTREMO consome tempo $\Theta(n)$, onde $n := r-p+1$.

O algoritmo PARTICIONE recebe uma coleção de pelo menos três pontos $X[p..r], Y[p..r]$ em posição geral tal que os pontos de índice p e r são extremos consecutivos na fronteira do fecho convexo da coleção no sentindo anti-horário. Ele rearranja $X[p..r], Y[p..r]$ e devolve índices p' e q tais que $p \le p' < q < r$ e

(i) o ponto de índice r permaneceu na mesma posição, enquanto que o ponto de índice p foi para a posição p',

(ii) o ponto de índice q é extremo,

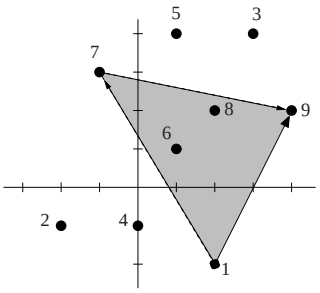
(iii) $X[p..p'-1], Y[p..p'-1]$ é uma coleção de pontos interiores ao fecho convexo da coleção $X[p..r], Y[p..r]$.

(iv) $X[p'+1..q-1], Y[p'+1..q-1]$ é a coleção dos pontos que estão à esquerda da reta orientada determinada por $(X[p'], Y[p'])$ e $(X[q], Y[q])$.

(v) $X[q+1..r-1], Y[q+1..r-1]$ é a coleção dos pontos que estão à esquerda da reta orientada determinada por $(X[q], Y[q])$ e $(X[r], Y[r])$.

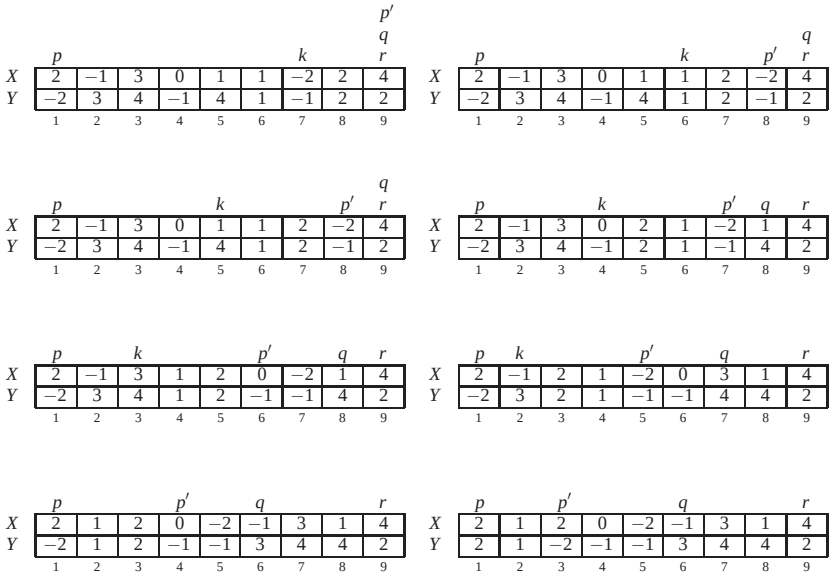
```
PARTICIONE (X,Y,p,r)
 1  q ← PONTOEXTREMO(X,Y,p,r)
 2  (X[q],Y[q]) ↔ (X[p+1],Y[p+1])
 3  p' ← r    q ← r
 4  para k ← r−1 decrescendo até p+2 faça
 5      se ESQ(X,Y,p,p+1,k)
 6          então  p' ← p'−1    (X[p'],Y[p']) ↔ (X[k],Y[k])
 7          senão se ESQ(X,Y,p+1,r,k)
 8                      então q ← q−1    (X[q],Y[q]) ↔ (X[k],Y[k])
 9                            p' ← p'−1  (X[k],Y[k]) ↔ (X[p'],Y[p'])
10  q ← q−1       (X[q],Y[q]) ↔ (X[p+1],Y[p+1])
11  p' ← p'−1     (X[p'],Y[p']) ↔ (X[p+1],Y[p+1])
12  p' ← p'−1     (X[p'],Y[p']) ↔ (X[p],Y[p])
13  devolva (p',q)
```

A seguir mostramos uma simulação de uma chamada do PARTICIONE.



PARTICIONE$(X,Y,1,9)$

	p	$p+1$						k	p' q r
X	2	−1	3	0	1	1	−2	2	4
Y	−2	3	4	−1	4	1	−1	2	2
	1	2	3	4	5	6	7	8	9



O algoritmo PARTICIONE está correto. O item (i) vale, pois o ponto de índice r não é trocado de lugar e o de índice p é trocado apenas na linha 12, pelo de índice p'. O item (ii) vale pela especificação do PONTOEXTREMO e pelas linhas 2 e 10. A cada passagem pela linha 4, imediatamente antes da comparação de k com $p+2$, pode-se demonstrar que valem os seguintes invariantes:

- $p+1 \leq k < p' \leq q \leq r$,
- os pontos de índice $p+1$ e r são extremos consecutivos na fronteira do fecho convexo de $X[q..r], Y[q..r]$ acrescido de $(X[p+1], Y[p+1])$ no sentindo anti-horário,
- os pontos de índice p e $p+1$ são extremos consecutivos na fronteira do fecho convexo de $X[p'..q-1], Y[p'..q-1]$ acrescido de $(X[p], Y[p])$ e $(X[p+1], Y[p+1])$ no sentindo anti-horário,
- os pontos da coleção $X[k+1..i-1], Y[k+1..i-1]$ são interiores ao fecho convexo de $X[p..r], Y[p..r]$.

No início da última iteração, $k = p+1$. Portanto as linhas 10-12 e esses invariantes garantem que, ao final, X, Y, p' e q satisfazem (ii)-(v). O primeiro invariante e o fato dos vetores X e Y serem alterados apenas por trocas simultâneas nos vetores X e Y implicam que a coleção $X[p..r], Y[p..r]$ ao final tem os mesmos pontos que no início.

Consumo de tempo de PARTICIONE. Seja $n := r - p + 1$. A linha 1 consome tempo $\Theta(n)$. As linhas 2,3 e 10-13 consomem tempo $\Theta(1)$. O bloco de linhas 5-9 é executado $n-3$ e cada execução consome tempo $\Theta(1)$. Logo o consumo de tempo de PARTICIONE é $\Theta(n)$.

Com isso, fica fácil descrever o QUICKHULL. Ele consiste em um pré-processamento e uma chamada ao algoritmo recursivo QUICKHULLREC, que é o semelhante ao QUICKSORT. O algoritmo QUICKHULLREC recebe uma coleção de pelo menos dois pontos $X[p..r], Y[p..r]$ em posição geral tal que os pontos de índices p e r são extremos e os pontos $X[p+1..r-1], Y[p+1..r-1]$ estão à esquerda da reta orientada determinada por $(X[p], Y[p])$ e $(X[r], Y[r])$ e devolve a descrição combinatória do fecho convexo de $X[p..r], Y[p..r]$ que começa de r. O algoritmo QUICKHULL recebe uma coleção de pontos $X[1..n], Y[1..n]$ em posição geral e devolve uma representação combinatória do seu fecho convexo.

```
QUICKHULL (X,Y,n)
 1  se n = 1
 2     então h ← 1    H[1] ← 1
 3     senão k ← min{i ∈ [1..n] : Y[i] ≤ Y[j], 1 ≤ j ≤ n}
 4           (X[1],Y[1]) ↔ (X[k],Y[k])
 5           i ← 2
 6           para j ← 3 até n faça
 7              se DIR(X,Y,1,i,j) então i ← j
 8           (X[n],Y[n]) ↔ (X[i],Y[i])
 9           (H,h) ← QUICKHULLREC(X,Y,1,n,H,h)
10  devolva (H,h)
```

O algoritmo QUICKHULL está correto. As linhas 3-4 rearranjam X e Y de modo que na posição 1 fique um ponto com coordenada Y mínima. Tal ponto é extremo pois a coleção dada está em posição geral.

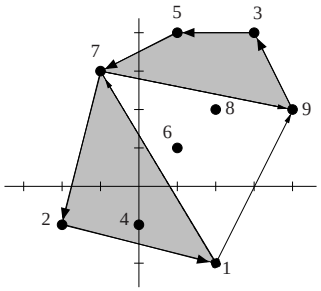
As linhas 5-7 são semelhantes a uma iteração de EMBRULHO, onde se busca o próximo ponto extremo do fecho, caso a coleção tenha pelo menos dois pontos. Ou seja, o índice i imediatamente antes da linha 8 é o índice do sucessor de $(X[1], Y[1])$ na descrição combinatória do fecho convexo da coleção. A linha 8 coloca na posição n este ponto extremo. Com isso, $X[1..n], Y[1..n]$ na linha 9 satisfaz a especificação de QUICKHULLREC.

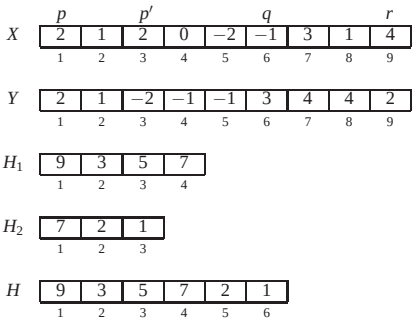
Desde que QUICKHULLREC esteja correto, o vetor H na linha 10 contém uma descrição combinatória do fecho convexo de $X[1..n], Y[1..n]$.

Consumo de tempo de QUICKHULL. As linhas 1-8 consomem tempo $\Theta(n)$. Já, como veremos a seguir, o algoritmo QUICKHULLREC consome tempo $O(n^2)$, onde $n := r - p + 1$. Portanto, QUICKHULL consome tempo $O(n^2)$.

O QUICKHULLREC é recursivo.

```
QUICKHULLREC (X,Y,p,r)
 1  se p = r−1     ▷ há exatamente dois pontos na coleção
 2     então h ← 2     H[1] ← r     H[2] ← p
 3     senão (p′,q) ← PARTICIONE(X,Y,p,r)
 4           (H1,h1) ← QUICKHULLREC(X,Y,q,r,H,h)
 5           (H2,h2) ← QUICKHULLREC(X,Y,p′,q,H,h)
             ▷ H ← H1 · H2 removendo uma cópia do q
 6           h ← 0
 7           para i ← 1 até h1 faça
 8              h ← h+1     H[h] ← H1[i]
 9           para i ← 2 até h2 faça
10              h ← h+1     H[h] ← H2[i]
11  devolva (H,h)
```





O algoritmo QUICKHULLREC está correto. A correção do QUICKHULL-REC é verificada por indução no número $n := r - p + 1$ de pontos. É evidente, pelas linhas 1-2, que o algoritmo dá a resposta correta quando há apenas dois pontos na coleção. Suponha, portanto, que a coleção tem pelo menos três pontos. Neste caso, o algoritmo executa o bloco de linhas 2-12.

O PARTICIONE rearranja $X[p..r], Y[p..r]$ e devolve dois índices p' e q entre p e r tais que (i)-(v) valem. Pelo (iii) da especificação do PARTICIONE, os pontos que não estão nas coleções das chamadas recursivas não são extremos e portanto são irrelevantes. Por (ii), o ponto de índice q é extremo, e os demais pontos extremos estão particionados entre as duas coleções de modo que a 'concatenação' das descrições combinatórias devolvidas, removida uma das cópias do q, formam uma descrição combinatória do fecho da coleção dada.

Consumo de tempo de QUICKHULLREC. O consumo de tempo de QUICKHULLREC é medido em relação ao número $n := r - p + 1$ de pontos na coleção $X[p..r], Y[p..r]$. Seja $T(n)$ o tempo consumido pelo algoritmo QUICKHULLREC quando aplicado a uma coleção de n pontos. A linha 3 consome tempo $\Theta(n)$. O tempo consumido pelo bloco de linhas 6-12 é proporcional ao número de pontos extremos da coleção. Como esse número é no máximo n, então o consumo de tempo desse bloco de linhas é $O(n)$. Portanto,

$$T(n) \;=\; T(n_1) + T(n_2) + \Theta(n), \tag{2}$$

onde $n_1 := q - p' + 1$ e $n_2 := r - q + 1$. O termo $T(n_1)$ corresponde ao consumo de tempo da linha 4 e o termo $T(n_2)$, ao consumo da linha 5. Vale que $n_1 + n_2 = r - p' + 2$ e também que $n_1 \geq 2$ e $n_2 \geq 2$, já que $p' < q < r$. Do invariante (i) e das linhas 11-12 de PARTICIONE, temos que $p' \geq p$, logo $n_1 + n_2 \leq n + 1$. A função $T(n)$ está em $O(n^2)$ (Exercício 4). Ou seja, o algoritmo QUICKHULLREC consome tempo $O(n^2)$.

Exercícios

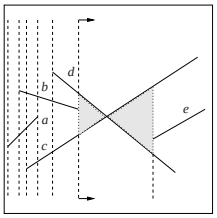
1. Quanto vale o determinante da página 14 quando (x_1, y_1), (x_2, y_2), e (x_3, y_3) são colineares?
2. Calcule os determinantes associados às chamadas de ESQUERDA do exemplo da página 14.
3. Escreva uma versão do algoritmo INSEREPONTO que funciona mesmo que a coleção dada de pontos não esteja em posição geral. Simule o INCREMENTAL com o seu algoritmo INSEREPONTO para a coleção abaixo:

X	0	1	2	4	2	6
	1	2	3	4	5	6

Y	0	1	2	0	0	0
	1	2	3	4	5	6

4. Mostre que $T(n)$ dada pela recorrência (2) está em $O(n^2)$.
5. Mostre uma coleção de n pontos, para n arbitrário, para a qual o algoritmo QUICKHULL consome tempo $\Theta(n^2)$ para dar sua resposta.

7.5. Interseção de segmentos



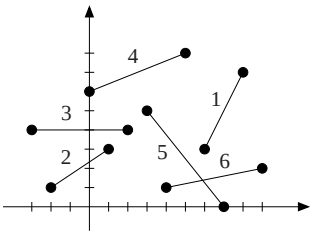
Um dos problemas mais básicos em geometria computacional é o de detectar interseção. O cálculo de interseções no plano e no espaço 3-dimensional é uma operação básica em diversas áreas. Em robótica e planejamento de movimento, é importante sabermos quando dois objetos se intersectam para evitarmos colisões. Em computação gráfica, *ray shooting* é um método importante para o processamento digital de cenas, e a parte deste que consome mais tempo é justamente determinar interseções entre o raio e os outros objetos.

7.5.1. O problema

Muitos problemas complexos de interseção são decompostos em problemas mais simples. Apresentamos nesta seção um algoritmo para uma versão bem simples de problema de interseção.

> **Problema de interseção de segmentos.** Dada uma coleção de segmentos no plano, decidir se existem dois segmentos na coleção que se intersectam.

Uma coleção de n segmentos no plano é representada por dois vetores $e[1..n]$ e $d[1..n]$ de pontos. A coordenada do ponto $e[i]$ é $(e_X[i], e_Y[i])$ e do ponto $d[i]$ é $(d_X[i], d_Y[i])$.



e_X	6	−2	−3	0	3	4
e_Y	3	1	4	6	5	1
	1	2	3	4	5	6

d_X	8	1	2	5	7	9
d_Y	7	3	4	8	0	2
	1	2	3	4	5	6

É fácil projetar um algoritmo que resolve esse problema e consome tempo $\Theta(n^2)$. Como já vimos, às vezes considerar o problema em um espaço de dimensão menor nos ajuda a encontrar um algoritmo mais rápido para espaços de dimensão maior. Vamos então fazer um desvio e pensar no problema quando os segmentos dados estão em uma reta.

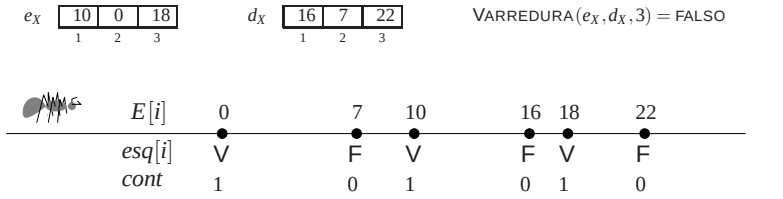
7.5.2. Interseção de intervalos

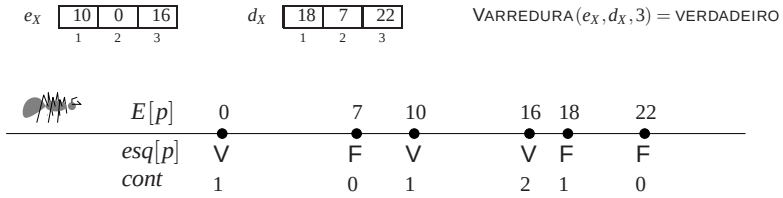
Quando os segmentos dados estão numa reta, podemos pensar neles simplesmente como intervalos. Cada intervalo pode ser dado por um par de números. Usamos dois vetores $e_X[1..n]$ e $d_X[1..n]$ para representar os n intervalos $[e_X[1]..d_X[1]], \ldots, [e_X[n]..d_X[n]]$. O seguinte algoritmo decide se há interseção entre dois dos intervalos dados, supondo que os pontos extremos são todos distintos.

```
VARREDURA(e,d,n)
 1  para i ← 1 até n faça    ▷ para cada intervalo
 2      E[i] ← e_X[i]        esq[i] ← VERDADEIRO    ▷ extremo esquerdo
 3      E[i+n] ← d_X[i]      esq[i+n] ← FALSO       ▷ extremo direito
 4  MERGESORT(E,esq,1,2n)
 5  cont ← 0    resp ← FALSO
 6  para p ← 1 até 2n faça    ▷ para cada ponto extremo
 7      se esq[p]
 8          então cont ← cont+1
 9                se cont = 2 então resp ← VERDADEIRO
10          senão cont ← cont−1
11  devolva resp
```

Após as linhas 1-3, o valor de $esq[p]$ indica se o número $E[p]$ é o extremo esquerdo de um intervalo da coleção. A linha 4 rearranja simultaneamente os vetores $E[1..2n]$ e $esq[1..2n]$ de modo que $E[1] < \cdots < E[2n]$.

Imagine uma formiga que começa a caminhar para a direita a partir do ponto extremo mais à esquerda. Para saber se há interseção entre dois dos intervalos, cada vez que a formiga encontra o extremo esquerdo de um intervalo, ela incrementa um contador que indica o número de intervalos que contêm o ponto em que ela está. Se o contador assume o valor 2, então há um ponto em dois dos intervalos dados. Assim há na coleção dois intervalos que se intersectam. O algoritmo acima implementa esta ideia. Perceba que o valor do contador é alterado apenas quando a formiga passa por um extremo de um dos intervalos dados.





O consumo de tempo de VARREDURA é $\Theta(n \lg n)$ por causa da linha 4. As demais linhas consomem tempo $\Theta(n)$.

Este algoritmo é um exemplo simples de aplicação de um método mais geral e poderoso que passamos a descrever.

7.5.3. Método da linha de varredura

No método da linha de varredura (*sweepline*), uma linha imaginária, digamos vertical, move-se da esquerda para a direita. À medida que a linha prossegue, o problema restrito aos objetos que ficaram à esquerda dela é resolvido. Toda a informação da parte do problema que está à esquerda da linha, e que é necessária para estender a solução parcial corrente, é mantida numa descrição combinatória da linha de varredura.

Apesar da metáfora da linha mover-se continuamente da esquerda para a direita, a sua descrição combinatória muda apenas em posições chaves, chamadas de *pontos eventos*. Podemos imaginar que a linha de varredura avança de ponto evento em ponto evento, da esquerda para a direita. Pontos eventos são então mantidos em uma fila, que é usualmente uma lista ordenada de acordo com o valor da coordenada X dos pontos. Esta fila dá a ordem em que estes pontos devem ser processados. Em algumas aplicações, a fila de eventos está completamente definida no instante inicial do algoritmo. Em outras, novos pontos eventos podem ser detectados e inseridos na fila a medida que a linha avança.

No algoritmo VARREDURA, a formiga faz o papel da linha de varredura e o contador é a descrição combinatória da informação necessária naquele caso. Cada extremo de um intervalo é um ponto evento.

O tratamento de cada ponto evento consiste tipicamente nas seguintes tarefas:

Atualizar a fila: remover o ponto evento corrente, junto com qualquer outro que tenha se tornado obsoleto e, eventualmente, inserir novos pontos eventos na fila;

Atualizar a linha: atualizar a descrição combinatória da linha de varredura para que esta represente a situação atual;

Resolver o problema: estender a solução corrente.

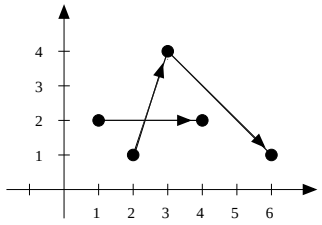
A fila de eventos de VARREDURA consiste em $E[i..2n]$, a atualização da linha é feita nas linhas 8 e 10, quando *cont* é ajustado, e a atualização da solução do problema é feita quando a variável *resp* é eventualmente alterada na linha 9.

O método da linha de varredura reduz um problema estático bidimensional a um problema dinâmico unidimensional: o problema de manter a descrição combinatória da linha. O problema unidimensional resultante é geralmente mais simples que o problema bidimensional original. Veremos duas aplicações do método: a primeira para o problema da interseção de segmentos no plano e a segunda para um problema de divisão de polígonos.

7.5.4. Predicado geométrico

O predicado geométrico usado no principal algoritmo desta seção é o INTERSECTA. Ele usa um dos predicados introduzidos na Subseção 7.4.2. O INTERSECTA recebe dois segmentos e devolve VERDADEIRO se os segmentos se intersectam, e FALSO caso contrário.

INTERSECTA$(x_1,y_1,x_2,y_2,x_3,y_3,x_4,y_4)$
1 se ESQUERDA$(x_1,y_1,x_2,y_2,x_3,y_3) \neq$ ESQUERDA$(x_1,y_1,x_2,y_2,x_4,y_4)$
2 e ESQUERDA$(x_3,y_3,x_4,y_4,x_1,y_1) \neq$ ESQUERDA$(x_3,y_3,x_4,y_4,x_2,y_2)$
3 então devolva VERDADEIRO
4 senão devolva FALSO



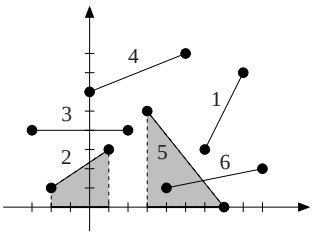
ESQUERDA$(1,2,4,2,2,1)$ = FALSO
ESQUERDA$(1,2,4,2,3,4)$ = VERDADEIRO
ESQUERDA$(2,1,3,4,1,2)$ = VERDADEIRO
ESQUERDA$(2,1,3,4,4,2)$ = FALSO
INTERSECTA$(1,2,4,2,2,1,3,4)$ = VERDADEIRO
ESQUERDA$(3,4,6,1,1,2)$ = FALSO
ESQUERDA$(3,4,6,1,4,2)$ = FALSO
INTERSECTA$(1,2,4,2,3,4,6,1)$ = FALSO

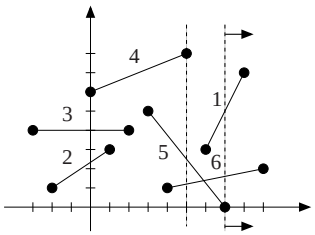
Para melhorar a legibilidade, temos a seguinte variante do INTERSECTA, que recebe uma coleção $e[1..n], d[1..n]$ de segmentos e dois índices i e j, e devolve VERDADEIRO se os segmentos de índices i e j da coleção se intersectam, e FALSO caso contrário.

INTER(e,d,i,j)
1 devolva INTERSECTA$(e_X[i],e_Y[i],d_X[i],d_Y[i],e_X[j],e_Y[j],d_X[j],d_Y[j])$

7.5.5. Algoritmo de Shamos e Hoey

Uma ideia natural para projetar um algoritmo eficiente é evitar o teste de interseção entre pares de segmentos que não tem chance de se intersectar. Como podemos fazer isto? Primeiro, pode-se eliminar os casos fáceis. Dois segmentos cuja projeção no eixo X sejam disjuntas não se intersectam. Abaixo, os segmentos de índices 2 e 5 não precisam ser submetidos ao teste de interseção.



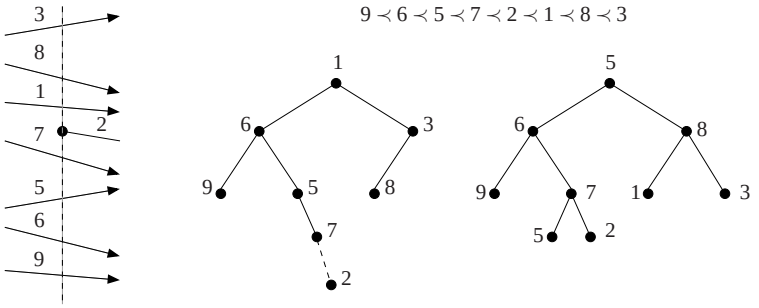


Se a projeção no eixo X de dois segmentos tem interseção, então há uma linha vertical que intersecta ambos. Para determinarmos estes pares de segmentos, usamos o método da linha de varredura. Imaginamos uma linha vertical varrendo o plano da esquerda para a direita. Enquanto a linha varre o plano, mantemos todos os segmentos intersectados por ela na descrição combinatória da linha. Estes são os candidatos a realizarmos o teste de interseção. Os segmentos que intersectam a linha pontilhada mais à esquerda na figura acima são $\{4,5,6\}$. Infelizmente, fazer o teste de interseção entre cada dois segmentos que são simultaneamente intersectados pela linha de varredura ainda resultaria num algoritmo que consome tempo $\Theta(n^2)$ no pior caso, pois a linha pode intersectar $\Omega(n)$ segmentos simultaneamente.

A segunda observação é que, se mantivermos os segmentos na ordem em que intersectam a linha de varredura, então basta, como veremos, realizarmos o teste de interseção entre pares de segmentos que são consecutivos nesta ordem em algum momento.

O algoritmo que vamos apresentar nesta subseção implementa essa ideia [Shamos and Hoey 1976]. Quando a linha de varredura é a reta $x = t$, a ordem $\prec_t$ em que os segmentos intersectados pela linha são mantidos é a seguinte. Para dois segmentos de índices i e j intersectados pela linha, $i \prec_t j$ se o ponto de interseção da linha com o segmento i fica abaixo do ponto de interseção com o segmento j. No exemplo acima, $6 \prec_5 5 \prec_5 4$ e $5 \prec_7 6 \prec_7 1$. A descrição combinatória da linha é esta ordem, que é alterada apenas quando a linha atinge um ponto extremo de um dos segmentos ou um ponto de interseção entre segmentos. No entanto, o algoritmo aqui apresentado para assim que detecta o primeiro ponto de interseção. Logo, a descrição combinatória da linha será alterada apenas em pontos extremos.

Usamos uma árvore de busca binária balanceada (ABBB) T para representar a descrição combinatória da linha. Mais precisamente, T representa a ordem $\prec_t$ quando a linha de varredura for a reta $x = t$, ou seja, os segmentos armazenados em T são exatamente aqueles que intersectam a reta $x = t$ e eles estão armazenados de acordo com a ordem $\prec_t$.



O algoritmo utiliza as seguintes rotinas de manipulação de uma ABBB:

CRIE(T): cria uma ABBB T vazia;

INSIRA(T,i): insere i na ABBB T, usando $\prec_t$ com $t = e_X[i]$;

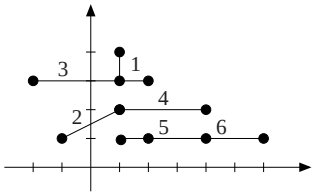
REMOVA(T,i): remove i da ABBB T, usando $\prec_t$ com $t = d_X[i]$;

PREDECESSOR(T,x,y): devolve o predecessor em T de um segmento que passa pelo ponto (x,y), usando $\prec_x$;

SUCESSOR(T,x,y): devolve o sucessor em T de um segmento que passa pelo ponto (x,y), usando $\prec_x$.

O consumo de tempo de cada uma destas rotinas é $O(\lg m)$, onde m é o número de elementos em T [Cormen et al. 2001].

O algoritmo INTERSEÇÃO-SH recebe uma coleção $e[1..n], d[1..n]$ de segmentos, e devolve VERDADEIRO se há dois segmentos na coleção que se intersectam, e FALSO caso contrário. Para simplificar, INTERSEÇÃO-SH supõe que não há dois pontos extremos com a mesma coordenada X e não há dois segmentos que se intersectam em mais do que um ponto. Ou seja, casos como os abaixo não são tratados.



	1	2	3	4	5	6
e_X	1	−1	−2	1	1	2
e_Y	3	1	3	2	1	1

	1	2	3	4	5	6
d_X	1	1	2	4	4	6
d_Y	4	2	3	2	1	1

O algoritmo INTERSEÇÃO-SH utiliza uma rotina FILADEEVENTOS que recebe os vetores $e[1..n]$ e $d[1..n]$ de pontos, representando uma coleção de n segmentos, e troca $e[i]$ por $d[i]$ para todo i tal que $e_X[i] > d_X[i]$, de modo que $e[i]$ indique o extremo esquerdo do segmento i e $d[i]$ indique o direito. Além disso, INTERSEÇÃO-SH devolve um vetor $E[1..2n]$, com os pontos de $e[1..n]$ e $d[1..n]$ ordenados pelas suas coordenadas X, um vetor $segm[1..2n]$ onde $segm[p]$ é o índice do segmento da coleção do qual $E[p]$ é extremo, e um vetor booleano $esq[1..2n]$ onde $esq[p]$ é VERDADEIRO se $E[p]$ é o extremo esquerdo do segmento de índice $segm[p]$ da coleção, e FALSO caso contrário. O consumo de tempo de FILADEEVENTOS é $\Theta(n \lg n)$.

```
FILADEEVENTOS(e, d, n)
 1  para i ← 1 até n faça    ▷ para cada segmento
 2      se e_X[i] > d_X[i] então e[i] ↔ d[i]
 3      E[i] ← e[i]        segm[i] ← i        esq[i] ← VERDADEIRO
 4      E[i+n] ← d[i]      segm[i+n] ← i      esq[i+n] ← FALSO
 5  MERGESORT(E, segm, esq, 1, 2n)
 6  devolva (E, segm, esq)
```

Para o exemplo do início desta subseção, os vetores E, *segm* e *esq* devolvidos pela rotina FILADEEVENTOS seriam

	1	2	3	4	5	6	7	8	9	10	11	12
E_X	−2	−1	1	1	2	2	2	3	3	4	5	6
E_Y	3	1	4	2	4	3	1	5	2	2	0	1

	1	2	3	4	5	6	7	8	9	10	11	12
segm	3	2	1	2	4	3	6	1	5	4	5	6

	1	2	3	4	5	6	7	8	9	10	11	12
esq	V	V	V	F	V	F	V	F	V	F	F	F

```
INTERSEÇÃO-SH(e,d,n)
 1  (E,segm,esq) ← FILADEEVENTOS(e,d,n)
 2  CRIE(T)
 3  para p ← 1 até 2n faça
 4      i ← segm[p]
 5      pred ← PREDECESSOR(T,E_X[p],E_Y[p])
 6      suc ← SUCESSOR(T,E_X[p],E_Y[p])
 7      se esq[p]
 8          então INSIRA(T,i)
 9                se (pred ≠ NIL e INTER(e,d,i,pred))
                  ou (suc ≠ NIL e INTER(e,d,i,suc))
10                    então devolva VERDADEIRO
11          senão REMOVA(T,i)
12                se pred ≠ NIL e suc ≠ NIL e INTER(e,d,pred,suc)
13                    então devolva VERDADEIRO
14  devolva FALSO
```

O algoritmo está correto. É evidente que se o algoritmo devolve VERDADEIRO então essa resposta é correta: há interseção entre dois dos segmentos. Assim, se não há interseção entre dois dos segmentos da coleção, o algoritmo devolve FALSO. Falta então mostrar que, sempre que há dois segmentos que se intersectam na coleção, ele devolve VERDADEIRO.

Depois da linha 1, vale que $E[1],\ldots,E[2n]$ são todos os pontos extremos dos segmentos da coleção e $E_X[1] \le \cdots \le E_X[2n]$. Seja $E[0]$ um ponto arbitrário à esquerda do ponto $E[1]$. Para um número t, denotamos por $\prec_{t^+}$ a ordem induzida pela linha de varredura assim que ela passou da reta $x = t$.

A cada passagem pela linha 4, valem os seguintes invariantes:

- T representa a ordem $\prec_{t^+}$ onde $t = E_X[p-1]$;
- já foram submetidos ao teste de interseção todos os segmentos que são consecutivos em $\prec_{t^+}$ para $t = E_X[1],\ldots,E_X[p]$.

O primeiro invariante decorre do teste da linha 7, que detecta se a linha de varredura atingiu o começo ou o fim do segmento i, definido na linha 4, e das linhas 8 e 11, que atualizam T adequadamente.

O segundo invariante decorre do seguinte. Se i é inserido em T, a linha 9 faz o teste de i com os segmentos consecutivos na ordem $\prec_t$ com $t = E_X[p]$, isto é, seu predecessor e seu sucessor em $\prec_t$, se existirem. Como não há dois pontos extremos com a mesma coordenada X, temos que neste caso $\prec_t = \prec_{t^+}$. Se i é removido de T, a linha 12 testa a interseção entre o predecessor e o sucessor de i em $\prec_t$, se ambos existirem. Estes dois segmentos são consecutivos em $\prec_{t^+}$. Estes são os únicos testes de interseção necessários para manter o segundo invariante.

Suponha que há dois segmentos na coleção que se intersectam. Por hipótese, sempre que dois segmentos se intersectam, isso ocorre em um único ponto. Escolha dois segmentos s e r da coleção que se intersectam num ponto (x,y) com x o menor possível. Observe que s e r são consecutivos em $\prec_{t^+}$ para $t = E_X[j]$ onde $j = \max\{q : E_X[q] < x\}$. Então, pelos invariantes, o algoritmo termina com $p \leq j$, devolvendo VERDADEIRO.

Consumo de tempo. O consumo da linha 1 é $\Theta(n \lg n)$ e da linha 2 é $\Theta(1)$. O bloco de linhas 4-13 é executado $2n$ vezes. Como T armazena no máximo n segmentos, cada execução deste bloco consome tempo $O(\lg n)$. Logo, o consumo total deste bloco de linhas é $O(n \lg n)$ e o consumo de tempo do INTERSEÇÃO-SH é $\Theta(n \lg n)$.

Exercícios

1. Ajuste INTERSEÇÃO-SH para que aceite pontos extremos com mesma coordenada X.
2. Escreva uma versão de INTERSEÇÃO-SH que, dada uma coleção de segmentos, devolva todas as interseções entre os segmentos. Para simplificar, suponha que não existam pontos extremos e interseções com a mesma coordenada X e que interseções entre mais do que dois segmentos não ocorram.
3. Descreva os campos de um nó da ABBB T que armazena a descrição combinatória da linha de varredura usada em INTERSEÇÃO-SH. Utilizando essa descrição do nó, escreva o algoritmo PREDECESSOR(T,x,y) usando o predicado ESQUERDA.

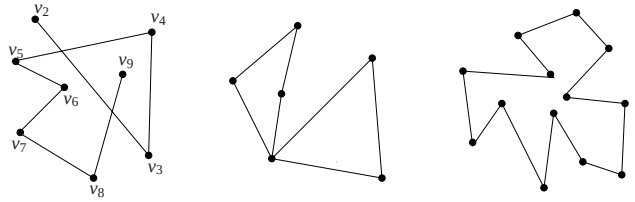
7.6. Divisão em polígonos monótonos

O interesse aqui é decompormos um certo domínio complexo em uma coleção de objetos simples. A região mais simples na qual podemos decompor um objeto plano é um triângulo. Dado um polígono, queremos adicionar diagonais que não se cruzem de modo a dividi-lo em triângulos. Esse processo é chamado de triangulação do polígono.

O algoritmo que veremos nesta seção utiliza o método da linha de varredura para dividir um polígono dado em polígonos ditos monótonos. Em conjunto com um algoritmo de Garey, Johnson, Preparata e Tarjan [Garey et al. 1978], que triangula em tempo linear polígonos monótonos, este algoritmo pode ser usado como pré-processamento, para triangular um polígono de n vértices em tempo $\Theta(n \lg n)$.

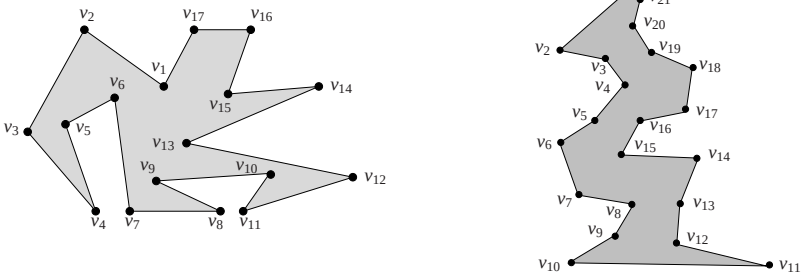
7.6.1. Polígonos e polígonos monótonos

Uma *curva poligonal* é uma sequência finita $(v_1, a_1, v_2, \ldots, a_{n-1}, v_n)$ onde $v_1, \ldots, v_n$ são pontos no plano e a_i é um segmento com extremos v_i e v_{i+1} para $i = 1, \ldots, n-1$. Os pontos $v_1, \ldots, v_n$ são chamados de *vértices* e os segmentos $a_1, \ldots, a_{n-1}$ de *arestas*. Uma curva poligonal é *fechada* se $v_1 = v_n$. Considere os índices dos vértices e arestas de um polígono ciclicamente, ou seja, $v_{n+1} = v_1$, $v_0 = v_n$, $a_n = a_1$ e $a_0 = a_{n-1}$. Uma curva poligonal fechada é *simples* se sempre que dois segmentos a_i e a_j com $i \neq j$ se intersectam, então $j = i-1$ e a interseção é exatamente o vértice v_i, ou $j = i+1$ e a interseção é exatamente o vértice v_{i+1}.



curva poligonal fechada não-simples fechada simples

Um *polígono* é a região fechada do plano limitada por uma curva poligonal fechada simples. A sua *descrição combinatória* é uma sequência de seus vértices listados na ordem em que aparecem ao percorrermos a fronteira do polígono no sentido anti-horário, sem repetições. A descrição combinatória do polígono mostrado abaixo é $(v_1, \ldots, v_{17})$. O fecho convexo de um conjunto de pontos nada mais é do que um polígono.

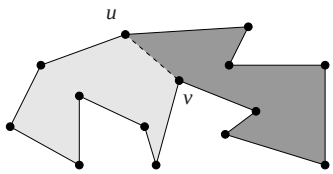


Um polígono é chamado de Y*-monótono* se a interseção dele com toda reta horizontal é um segmento de reta, um ponto ou é vazia. Acima, o polígono da esquerda não é Y-monótono, já o da direita é.

Para simplificar a exposição, vamos assumir que o polígono não possui dois vértices com a mesma coordenada Y.

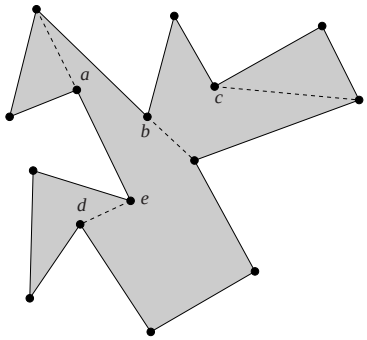
7.6.2. O problema

Sejam u e v dois vértices de um polígono P. O segmento uv é uma *diagonal* de P se ele está contido em P e intersecta a fronteira de P apenas em u e v. A adição de uma diagonal a um polígono P o divide naturalmente em dois polígonos.



Problema da divisão em polígonos monótonos. Dado um polígono P, encontrar um conjunto de diagonais de P que o dividam em polígonos Y-monótonos.

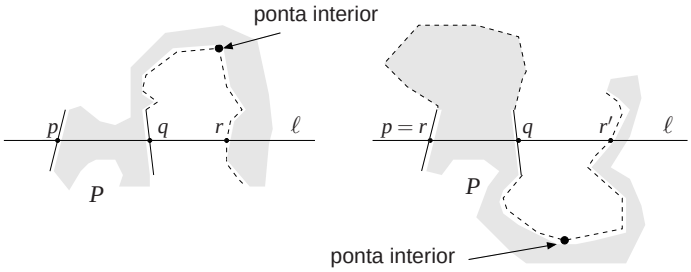
O polígono abaixo é dividido em cinco polígonos Y-monótonos pela adição das quatro diagonais indicadas.



Uma *ponta interior* de um polígono é um vértice v_i para o qual ambos v_{i-1} e v_{i+1} estão acima ou ambos estão abaixo de v_i e o ângulo interno determinado por v_{i-1}, v_i, v_{i+1} é maior que π. Se ambos estão acima, v_i é uma ponta *para baixo*, senão é uma ponta *para cima*. Acima, a e d são as pontas interiores para cima, e b, c e e são as pontas interiores para baixo.

Pontas interiores são fontes locais de não-monotonicidade: um polígono é Y-monótono se não possui pontas interiores. De fato, suponha que P não é Y-monótono. Temos que mostrar que P contém uma ponta interior. Como P não é Y-monótono, existe uma reta horizontal ℓ que intersecta P em mais de um componente conexo. Podemos escolher ℓ tal que o componente mais à esquerda seja um segmento e não um simples ponto. Seja p o extremo esquerdo deste segmento e q o seu extremo direito. Se seguirmos a fronteira de P em sentido anti-horário a partir de q, ela intersecta ℓ novamente em algum ponto r. Se $r \neq p$

então o vértice mais alto que encontramos neste percurso é necessariamente uma ponta interior para cima.



Se $r = p$ e seguirmos a fronteira de P a partir de q no sentido horário, ela intersecta ℓ num ponto r'. Observe que $r' \neq p$, já que a fronteira de P intersecta ℓ mais do que duas vezes. Assim, o vértice mais baixo que encontramos no percurso de q até r' é necessariamente uma ponta interior para baixo.

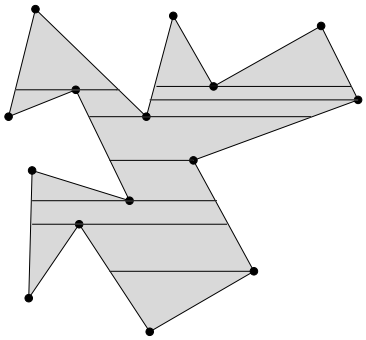
Um polígono terá sido dividido em polígonos Y-monótonos assim que nos livramos das suas pontas interiores. Fazemos isto adicionando diagonais que vão para cima a partir de pontas interiores para cima e que vão para baixo a partir de pontas interiores para baixo.

Vamos descrever um algoritmo que resolve o problema da divisão em polígonos monótonos através dessa ideia, usando a técnica da linha de varredura [Lee and Preparata 1977].

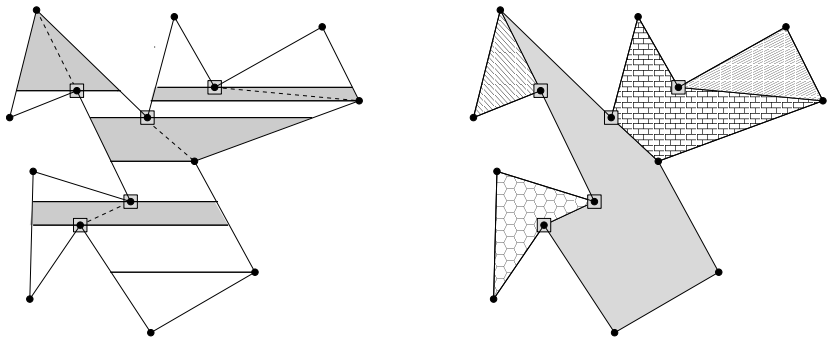
Um polígono com n vértices é representado por vetores $X[1..n]$ e $Y[1..n]$, com os vértices na ordem em que aparecem em sentido anti-horário na fronteira do polígono.

7.6.3. Trapézios e pontas interiores

Um *trapézio* é um quadrilátero que possui duas arestas paralelas. Para cada vértice v de um polígono P, trace um segmento horizontal maximal passando por v e contido em P. Esses segmentos dividem P naturalmente em trapézios cujas arestas paralelas são horizontais.



Observe que, como não há vértices com a mesma coordenada Y, o vértice v é o único vértice no seu segmento maximal. Assim, todo trapézio dessa divisão tem exatamente um vértice de P na sua aresta horizontal superior, chamado de *vértice de suporte superior*, e outro na sua aresta horizontal inferior, chamado de *vértice de suporte inferior*. Se um tal vértice estiver no interior de uma aresta do trapézio, ele será chamado de *vértice interior de suporte*. Abaixo, à esquerda, destacamos os trapézios que têm vértices interiores de suporte. Todo vértice interior de suporte é uma ponta interior e vice-versa.



Se ligarmos cada vértice interior de suporte ao outro vértice de suporte do trapézio, estes segmentos serão diagonais que dividem P em polígonos Y-monótonos. De fato, cada ponta interior deixa de ser uma ponta interior pois passa a ter, nos dois subpolígonos gerados por esta diagonal, um vizinho acima e outro abaixo dela.

Os algoritmos mostrados a seguir utilizam a rotina PONTAPARABAIXO, que recebe o índice x de um vértice de suporte superior de um trapézio do polígono, o vetor $Y[1..n]$ com as coordenadas Y dos vértices do polígono e devolve VERDADEIRO se o vértice de índice x é uma ponta interior para baixo, e FALSO caso contrário. A rotina PONTAPARABAIXO consome tempo $\Theta(1)$.

```
PONTAPARABAIXO(x, Y, n)
 1  x⁻ ← x − 1    x⁺ ← x + 1
 2  se x⁻ = 0 então x⁻ ← n
 3  se x⁺ = n + 1 então x⁺ ← 1
      ▷ x⁻ e x⁺ são o predecessor e o sucessor de x na fronteira do polígono
 4  se Y[x⁻] > Y[x] e Y[x⁺] > Y[x]     ▷ x é ponta interior para baixo?
 5      então devolva VERDADEIRO
 6      senão devolva FALSO
```

7.6.4. Algoritmo de Lee e Preparata

O algoritmo que descrevemos aqui é uma implementação da ideia sugerida na subseção anterior [Lee and Preparata 1977]. Ele utiliza a técnica da linha de varredura. Abaixo descrevemos quais são os pontos eventos, o funcionamento

da fila de pontos eventos, da estrutura da linha de varredura e como cada tipo de ponto evento é processado pelo algoritmo, que recebe $X[1..n]$ e $Y[1..n]$, representando um polígono.

Pontos eventos

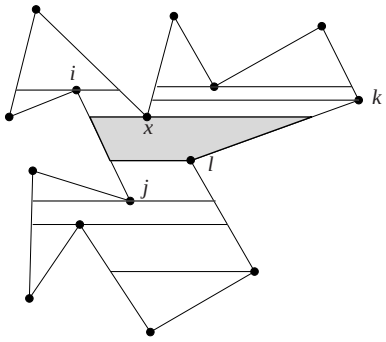
Os pontos eventos são os vértices do polígono. A fila de pontos eventos é determinada estaticamente e consiste em um vetor $E[1..n]$ com uma permutação dos índices de 1 a n dos vértices do polígono tal que

$$Y[E[1]] > Y[E[2]] > \cdots > Y[E[n]].$$

Linha de varredura

Descrevemos um trapézio do polígono pela tripla $((i,j),x,(k,l))$, onde

- (i,j) é o par de índices dos vértices da aresta do polígono que contém o lado esquerdo do trapézio, sendo que i está acima de j,
- x é o vértice de suporte superior do trapézio, e
- (k,l) é o par de índices dos vértices da aresta do polígono que contém o lado direito do trapézio, sendo que k está acima de l.



Imaginamos a linha varrendo o polígono de cima para baixo. Dizemos que um trapézio é *ativo* se ele intersecta a linha de varredura. A descrição combinatória da linha guarda os trapézios ativos na ordem em que são intersectados por ela, da esquerda para a direita. Assim, quando a linha de varredura é a reta $y = t$, temos que $((i,j),x,(k,l)) \prec_t ((i',j'),x',(k',l'))$ se a interseção da linha com o trapézio $((i,j),x,(k,l))$ está à esquerda da interseção com o trapézio $((i',j'),x',(k',l'))$.

Usamos novamente uma ABBB T para representar a descrição combinatória da linha. Mais precisamente, T representa a ordem $\prec_t$ quando a linha de varredura for a reta $y = t$, ou seja, os trapézios armazenados em T são os ativos e eles estão armazenados de acordo com a ordem $\prec_t$.

O algoritmo utiliza as seguintes rotinas para manipular T:

CRIE(T): cria uma ABBB T vazia;

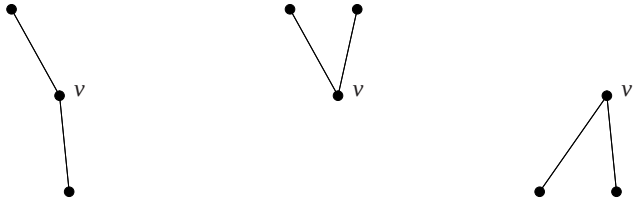
INSIRA(T,i,j,v,k,l): insere o trapézio $((i,j),v,(k,l))$ na ABBB T, usando $\prec_t$ com $t = Y[v]$;

REMOVA(T,v): devolve NIL se não há trapézio em T que contenha o vértice v, do contrário remove de T um trapézio que contenha o vértice v e devolve a sua descrição combinatória, usando $\prec_t$ com $t = Y[v]$.

O consumo de tempo de cada uma destas rotinas é $O(\lg m)$, onde m é o número de trapézios em T [Cormen et al. 2001].

Processamento dos pontos eventos

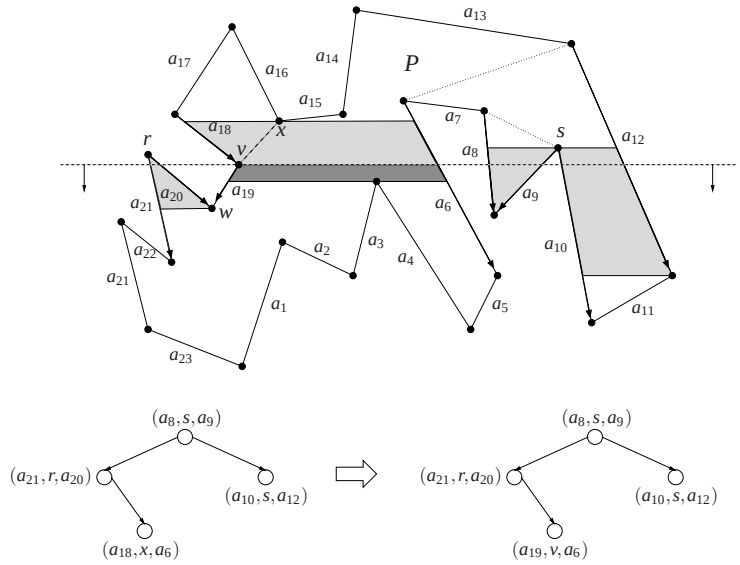
A atualização da descrição combinatória da linha de varredura depende do tipo do ponto evento. Seja v o ponto evento corrente, v^- o seu predecessor e v^+ o seu sucessor na fronteira do polígono. Dividimos o processo em três casos. O primeiro caso ocorre quando um dos vértices v^- e v^+ está acima de v e o outro está abaixo. O segundo caso ocorre quando os dois estão abaixo de v, e o terceiro caso, quando os dois estão acima de v.



Caso 1 Caso 2 Caso 3

No primeiro caso, temos que substituir em T o trapézio ativo que tem v como vértice de suporte inferior pelo que tem v como vértice de suporte superior. Além disso, se o vértice de suporte superior do trapézio removido for uma ponta para baixo, traçamos uma diagonal de v até ele.

A rotina TRATACASO1 implementa esse caso. Ela recebe a ABBB T com a descrição combinatória da linha de varredura imediatamente acima do vértice v, três índices u, v, e w de vértices consecutivos na fronteira do polígono tais que um dentre u e w está abaixo de v e o outro acima, as coordenadas $Y[1..n]$ dos vértices do polígono, e um vetor $D[1..t]$ com diagonais do polígono. Ela ajusta T de maneira a que passe a representar a linha de varredura imediatamente abaixo de v, e eventualmente acrescenta ao vetor D uma diagonal com uma das pontas em v, a fim de eliminar uma ponta para baixo.



TRATACASO1(T,u,v,w,Y,n,D,t)
1 se $Y[u] < Y[w]$ então $u \leftrightarrow w$
2 $((i,j),x,(k,l)) \leftarrow$ REMOVA(T,v)
3 se $v = j$ ▷ o trapézio está à direita de v?
4 então INSIRA(T,v,w,v,k,l)
5 senão INSIRA(T,i,j,v,v,w)
6 se PONTAPARABAIXO(x,Y,n)
7 então $t \leftarrow t+1$ $D[t] \leftarrow (x,v)$

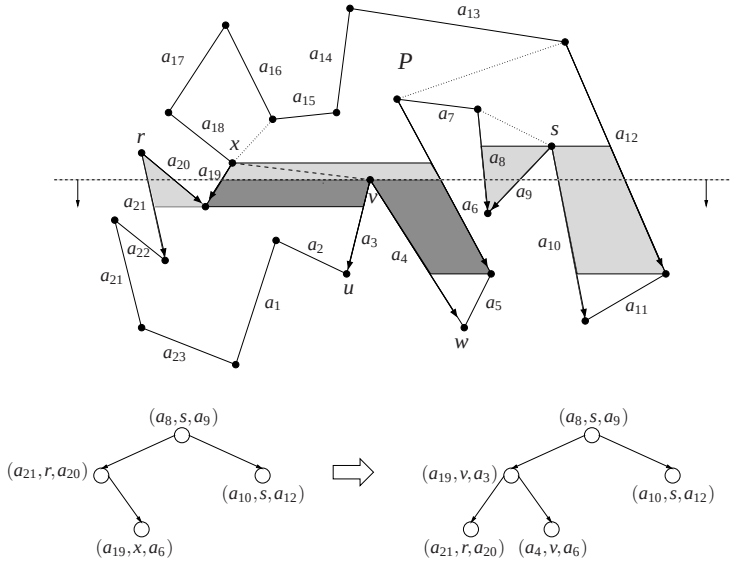
O segundo caso ocorre com os vértices v e r na situação ilustrada à frente. Estes são os dois casos possíveis: num o ponto evento está em um trapézio ativo que deixará de ser ativo, e no outro ele não está em nenhum trapézio ativo.

Se v está em um trapézio ativo, como na situação ilustrada, então ele é uma ponta interior para cima e será processado da seguinte maneira. Removemos de T o trapézio que contém v e inserimos os dois trapézios que passam a ser ativo, tendo v como vértice de suporte superior. Ademais, traçamos uma diagonal de v até x, onde x é o vértice de suporte do trapézio que foi removido de T.

Se v não está em um trapézio ativo, então um único trapézio passa a ser ativo, tendo as arestas incidentes a v como arestas laterias e v como vértice de suporte superior.

A rotina TRATACASO2 implementa esse caso. Ela recebe a ABBB T com a descrição combinatória da linha de varredura imediatamente acima do vér-

tice v, três índices u, v, e w de vértices consecutivos na fronteira do polígono tais que u e w estão abaixo de v, as coordenadas X dos vértices do polígono, e um vetor $D[1..t]$ com diagonais do polígono. Ela ajusta T de maneira a que passe a representar a linha de varredura imediatamente abaixo de v, e eventualmente acrescenta uma diagonal ao vetor D.



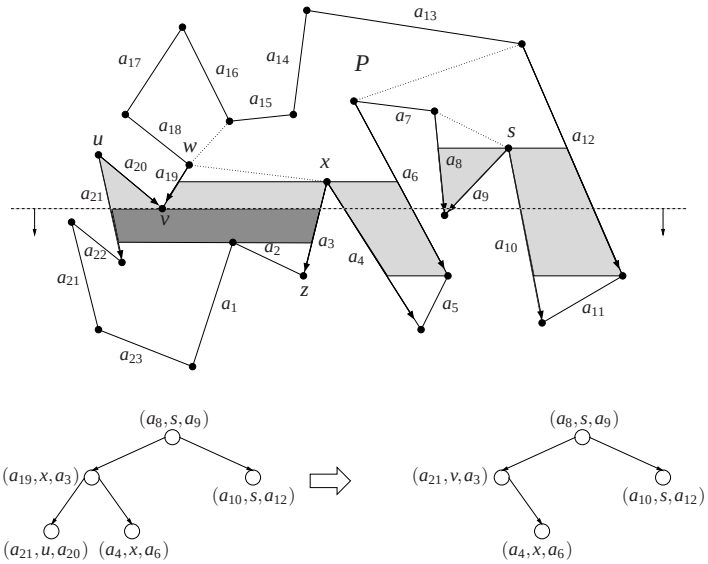
TRATACASO2(T, u, v, w, X, D, t)
1 se $X[u] > X[w]$ então $u \leftrightarrow w$
2 $trap \leftarrow$ REMOVA(T, v)
3 se $trap =$ NIL
4 então INSIRA(T, v, u, v, v, w)
5 senão $((i, j), x, (k, l)) \leftarrow trap$
6 INSIRA(T, i, j, v, v, u)
7 INSIRA(T, v, w, v, k, l)
8 $t \leftarrow t + 1 \quad D[t] \leftarrow (x, v)$ ▷ v é ponta interior para cima

O terceiro caso ocorre com os vértices v e z na situação ilustrada à frente. Estes são os dois casos possíveis: num o ponto evento está em um trapézio ativo que deixará de ser ativo, e no outro ele está em dois trapézios ativos que deixarão de ser ativos.

Se v está em um trapézio ativo apenas, então basta remover este trapézio de T e se o vértice de suporte superior deste trapézio for ponta interior para baixo, traçamos uma diagonal de v para este vértice.

Se v está em dois trapézios ativos, então removemos estes dois trapézios, que têm como vértices de suporte superiores, digamos, x e y, e caso x ou y sejam pontas interiores para baixo, traçamos diagonais de v para eles.

A rotina TRATACASO3 implementa esse caso. Ela recebe a ABBB T com a descrição combinatória da linha de varredura imediatamente acima do vértice v, três índices u, v, e w de vértices consecutivos na fronteira do polígono tais que u e w estão acima de v, as coordenadas Y dos vértices do polígono, e um vetor $D[1..t]$ com diagonais do polígono. Ela ajusta T de maneira a que passe a representar a linha de varredura imediatamente abaixo de v, e eventualmente acrescenta uma ou duas diagonais ao vetor D.




```
TRATACASO3(T,v,Y,n,D,t)
 1  ((i, j),x,(k,l)) ← REMOVA(T,v)
 2  se PONTAPARABAIXO(x,Y,n)
 3      então t ← t+1    D[t] ← (x,v)
 4  se j ≠ v ou k ≠ v    ▷ há um outro trapézio em T?
 5      então ((i′, j′),y,(k′,l′)) ← REMOVA(T,v)
 6            se PONTAPARABAIXO(y,Y,n)
 7                então t ← t+1    D[t] ← (y,v)
 8            se l = v
 9                então INSIRA(i, j,v,k′,l′)
10                senão INSIRA(i′, j′,v,k,l)
```

O consumo de tempo das rotinas TRATACASO1, TRATACASO2 e TRATACASO3 é $O(\lg m)$, onde m é o número de trapézios em T. Temos que $m < n$, assim esse consumo é $O(\lg n)$.

Algoritmo

Finalmente estamos prontos para apresentar o pseudocódigo do algoritmo de Lee e Preparata. As linhas 1-3 constroem a fila $E[1..n]$ de pontos eventos, formada pelos índices de 1 a n de modo que $Y[E[1]] > \cdots > Y[E[n]]$. O bloco 6-14 de linhas trata cada ponto evento conforme discutimos.

```
DIVIDEEMMONÓTONO-LP(X,Y,n)
 1  para k ← 1 até n faça
 2      E[k] ← k
 3  MERGESORT(Y,X,1,n,E)    ▷ ordenação indireta decrescente de Y
 4  CRIE(T)    t ← 0
 5  para k ← 1 até n faça
 6      v ← E[k]
 7      v⁻ ← v−1    v⁺ ← v+1
 8      se v⁻ = 0 então v⁻ ← n
 9      se v⁺ = n+1 então v⁺ ← 1
10      se Y[v⁻] < Y[v] < Y[v⁺] ou Y[v⁺] < Y[v] < Y[v⁻]
11          então TRATACASO1(T,v⁻,v,v⁺,Y,n,D,t)
12          senão se Y[v⁻] < Y[v]
13                    então TRATACASO2(T,v⁻,v,v⁺,X,D,t)
14                    senão TRATACASO3(T,v,Y,n,D,t)
15  devolva (D,t)
```

O algoritmo está correto. Primeiramente, cada par de pontos (x,v) inserido no vetor D nas rotinas TRATACASO1, TRATACASO2 e TRATACASO3 é uma diagonal do polígono. De fato, x e v são vértices de suporte de um mesmo trapézio: aquele que deixou de ser ativo e foi removido de T imediatamente antes do par ser incluído em D. Assim, o segmento com extremos x e v está contido no polígono. Ademais, como pelo menos um entre x e v é ponta interior, e portanto vértice de suporte interior deste trapézio, este segmento intersecta a fronteira do polígono apenas em seus extremos e logo é uma diagonal.

Cada trapézio do polígono estará ativo em alguma iteração. Como já observado, uma ponta interior é vértice de suporte de algum trapézio. Uma ponta interior para baixo é eliminada na iteração em que deixou de ser ativo o trapézio do qual ela é vértice de suporte superior. Isso pode ocorrer no TRATACASO1 ou no TRATACASO3. Uma ponta interior para cima é eliminada quando deixou de ser ativo o trapézio do qual ela é vértice de suporte inferior. Isso ocorre no TRATACASO2.

Consumo de tempo. As linhas 1-2 consomem tempo $\Theta(n)$ enquanto que a linha 3 consome tempo $\Theta(n \lg n)$. Em cada execução do bloco de linhas 6-14, exatamente uma entre TRATACASO1, TRATACASO2 ou TRATACASO3 é acio-

nada. Uma execução do bloco de linhas 6-9 consome tempo $\Theta(1)$. Assim, o consumo total de tempo do bloco de linhas 6-14 é $O(n \lg n)$, já que este bloco é executado n vezes. Com isso, o algoritmo de Lee e Preparata consome tempo $\Theta(n \lg n)$.

Exercícios

1. Ajuste DIVIDEEMMONÓTONO-LP para que aceite vértices com mesma coordenada Y.
2. Prove que a soma do número de vértices dos subpolígonos obtidos pela adição das diagonais encontradas pelo algoritmo DIVIDEEMMONÓTONO-LP é $O(n)$, onde n é o número de vértices do polígono dado.
3. Descreva os campos de um nó da ABBB T que armazena a descrição combinatória da linha de varredura usada em DIVIDEEMMONÓTONO-LP. Utilizando essa descrição, escreva um algoritmo BUSQUE(T, v), que recebe a árvore T e o índice de um vértice do polígono representado por $X[1..n], Y[1..n]$ e devolve NIL caso não haja um trapézio em T que contém v, e caso contrário devolve um trapézio de T que contenha v. Seu algoritmo deve usar o predicado ESQUERDA.

7.7. Conclusões

Neste texto, pudemos apresentar apenas alguns problemas e técnicas em geometria computacional. Deixamos de lado um grande número de tópicos centrais, tais como diagramas de Voronoi, triangulação de Delaunay, localização de pontos, grafos de visibilidade, problema de galeria de arte, etc. Tais tópicos, dentre outros, podem ser encontrados em vários dos livros aqui citados. Esperamos assim mesmo que o que foi apresentado desperte o interesse do leitor pela esta fascinante área.

Agradecimentos

Somos gratos ao Paulo Feofiloff (Departamento de Ciência da Computação do Instituto de Matemática e Estatística da USP), de quem copiamos o formato do texto e com quem trocamos várias ideias. Nosso trabalho foi simplificado pois pudemos nos apoiar em seu capítulo de Análise de Algoritmos. Agradecemos também aos revisores por suas valiosas sugestões e pela correção de vários erros. Finalmente, agradecemos a Alexis Sakurai Landgraf Carvalho, Natan Costa Lima e Lucas Piva pelas animações dos algoritmos exibidas durante o curso.

Referências bibliográficas

[Aho et al. 1974] Aho, A. V., Hopcroft, J. E., and Ullman, J. D. (1974). *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, Massachusetts.

[Bykat 1978] Bykat, A. (1978). Convex hull of a finite set of points in two dimensions. *Information Processing Letters*, 7:296–298.

[Chand and Kapur 1970] Chand, D. and Kapur, S. (1970). An algorithm for convex polytopes. *JACM*, 17(1):78–86.

[Cormen et al. 2001] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2001). *Introduction to Algorithms*. MIT Press, 2. edition.

[de Berg et al. 1997] de Berg, M., van Kreveld, M., Overmars, M., and Schwarzkopf, O. (1997). *Computational Geometry: Algorithms and Applications*. Springer. Second Edition, 2000.

[de Figueiredo and Stolfi 1997] de Figueiredo, L. and Stolfi, J. (1997). *Self-Validated Numerical Methods and Applications*. IMPA, Rio de Janeiro, Brasil.

[de Rezende and Stolfi 1994] de Rezende, P. and Stolfi, J. (1994). *Fundamentos de Geometria Computacional*. IX Escola de Computação.

[Eddy 1977] Eddy, W. (1977). A new convex hull algorithm for planar sets. *ACM Trans. Math. Software*, 3(4):398–403.

[Figueiredo and Carvalho 1991] Figueiredo, L. and Carvalho, P. (1991). *Introdução à Geometria Computacional*. 18º Colóquio Brasileiro de Matemática. IMPA. QA758 F475i.

[Garey et al. 1978] Garey, M., Johnson, D., Preparata, F., and Tarjan, R. (1978). Triangulating a simple polygon. *Information and Processing Letters*, 7:175–179.

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

[Green and Silverman 1979] Green, P. and Silverman, B. (1979). Constructing the convex hull of a set of points in the plane. *Computer Journal*, 22:262–266.

[Guibas et al. 1989] Guibas, L., Salesin, D., and Stolfi, J. (1989). Epsilon geometry: building robust algorithms from imprecise computations. In *Proceedings of the 5th Annual ACM Symposium on Computational Geometry*, pages 208–217.

[Jarvis 1973] Jarvis, R. (1973). On the identification of the convex hull of a finite set in the plane. *Information Processing Letters*, 2:18–21.

[Kleinberg and Tardos 2006] Kleinberg, J. and Tardos, E. (2006). *Algorithm Design*. Addison-Wesley.

[Laszlo 1996] Laszlo, M. (1996). *Computational Geometry and Computer Graphics in C++*. Prentice Hall, Upper Saddle River, NJ.

[Lee and Preparata 1977] Lee, D. and Preparata, F. (1977). Location of a point in a planar subdivision and its applications. *SIAM Journal on Computing*, 6:594–606.

[Mulmuley 1994] Mulmuley, K. (1994). *Computational Geometry: An Introduction Through Randomized Algorithms*. Prentice Hall, Englewood Cliffs, NJ.

[O'Rourke 1993] O'Rourke, J. (1993). *Computational Geometry in C*. Cambridge University Press, Cambridge. Second Edition, 1998.

[Preparata and Shamos 1985] Preparata, F. and Shamos, M. (1985). *Computational Geometry: An Introduction*. Texts and Monographs in Computer Science. Springer-Verlag, New York. QA758 P927c.

[Shamos and Hoey 1975] Shamos, M. and Hoey, D. (1975). Closest point problems. In *Proc. 16th Annual IEEE Symposium in Foundations of Computer Science*, pages 151–162.

[Shamos and Hoey 1976] Shamos, M. and Hoey, D. (1976). Geometric intersection problems. In *Proceedings of the 17th Annual IEEE Symposium on Foundations of Computer Science*, pages 208–215.