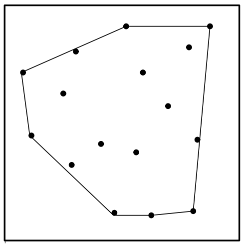
MAC 5747 GEOMETRIA COMPUTACIONAL

FECHO CONVEXO NO PLANO

1. Introdução



Dados n pontos no plano queremos encontrar o fecho convexo desses pontos. Uma aplicações práticas deste problema se encontra em robótica. Se o fecho convexo de um robô não colide com obstáculos então o robô também não colide. Segundo O'Rourke (cf. [19], pg. 80) talvez o primeiro artigo na área de geometria computacional tenha sido sobre fechos convexos.

Nos anos 60 uma aplicação da Bell Labs necessitava computar o fecho convexo de aproximadamente 10.000 pontos no plano e os algoritmos de complexidade de tempo $O(n^2)$ eram muito lentos. Tendo esse problema como motivação, no começo do anos 70, Graham [10] desenvolveu o primeiro algoritmo de complexidade de tempo $O(n \log n)$. O fecho convexo também pode ser construído em tempo $O(n \log n)$ por um algoritmo de divisão-e-conquista (cf. Capítulo 3 de Preparata e Shamos[21]).

Nestas parte do curso estudaremos diversos algoritmos para encontrar o fecho convexo de um conjunto dado de pontos no plano. Começaremos vendo algoritmos mais ingênuos (complexidade de tempo $O(n^4)$ e $O(n^3)$) e depois esturaremos algoritmos mais e mais rápidos até chegarmos em algoritmos ótimos para o problema (complexidade de tempo $O(n \log n)$).

Na próxima seção veremos várias definições equivalente de convexidade e fecho convexo. Na Seção 3, antes de continuarmos, descrevemos de uma maneira clara o problema tratado. Depois disto, na Seção 4, veremos dois algoritmos ingênuos que constroem o fecho convexo de um conjunto de pontos. Na Seção 5 estudaremos o algoritmo de Jarvis [13], que contrói o fecho convexo de um conjunto de n pontos no plano em tempo $O(nh)$, onde h é o número de vértices do fecho convexo de S — ou seja, o algoritmo é *ouput-sensitive*. O algoritmo que veremos na Seção 6, com algumas variações, foi proposto independentemente por várias pessoas quase ao mesmo tempo (cf. Eddy [8], Bykat [2], Green e Silverman [11]). Devido a semelhança com o algoritmo `Quicksort` o algoritmo que veremos foi batizado de `Quickhull` por Preparata e Shamos (cf. Seção 3 do Capítulo 3 de [21]). Na Seção 7 estudaremos o algoritmo de Graham [10], que foi o primeiro algoritmo de complexidade $O(n \log n)$ para o problema do fecho convexo planar. Um *algoritmo incremental* examina os dados do problema, um item por vez, enquanto uma solução corrente dos itens que já foram vistos até o momento é mantida, na Seção 8 descreveremos um algoritmo incremental para o

Date: 7 de julho de 2000.

problema do fecho convexo. Descreveremos, na Seção 9, uma versão probabilística de um algoritmo incremental para construir o fecho convexo de um conjunto de pontos no plano, a complexidade de tempo esperada deste algoritmo é $O(n \log n)$. Um algoritmo por divisão-e-conquista de complexidade de tempo $O(n \log n)$ será apresentado na Seção 10. Em seguida, na Seção 11, mostraremos que o problema do fecho convexo tem cota inferior $\Omega(n \log n)$. Na Seção 12 consideraremos os problema do par mais-distante, que pode ser resolvido com o auxílio de um algoritmo para o problema do fecho convexo. Finalmente, na Seção 13, temos uma lista de exercícios relacionados com o tópico estudado.

Observação. Os tópicos que cobriremos foram extraídos de: Capítulo 1 de De Berg, Van Kreveld, Overmars e Schwarzkoft [5] Capítulo 35 de Cormen, Leiserson e Rivest [4]; Capítulo 3 de Figueiredo e Carvalho [9]; Capítulo 6 de Laszlo [15], Capítulo 3 de O'Rourke [19]; Capítulos 3 e 4 de Preparata e Shamos [21]; e Capítulo 4 de Resende e Stolfi [6].

2. Definições de convexidade e fecho convexo

Existem várias definições equivalentes de fecho convexo de um conjunto de pontos. Daremos aqui algumas dessas definições. Notemos que algumas destas definições têm um natureza mais construtiva (algorítmica) e algumas definições serão dadas para o espaço Euclidiano d-dimensional E^d (e portanto serão mais gerais).

(1) Um conjunto X em E^d é *convexo* se $x \in X$ e $y \in X$ implica que o segmento (fechado) $xy \subseteq X$. Notemos que um polígono com um vértice reflexo não é convexo, veja Figura 1.

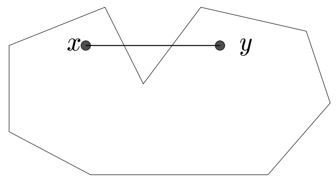


Figura 1. Um polígono com vértice reflexo não é convexo.

(2) (Segmento do ponto de vista algébrico) Dados dois pontos distintos x e y em E^d o segmento xy é o conjunto dos pontos $\lambda x + (1-\lambda)y$, com $0 \leq \lambda \leq 1$.

(3) Uma *combinação convexa* de um conjunto de pontos $S = \{p_1, \dots, p_k\}$ em E^d uma soma da forma $\alpha_1 p_1 + \cdots + \alpha_k p_k$, com $\alpha_i \geq 0$ (i=1,...,k) e $\alpha_1 + \cdots + \alpha_k = 1$. Logo, um segmento de reta consiste do conjunto dos pontos que podem ser escritos como combinação convexa dos extremos do segmento e um triângulo consiste do conjunto de pontos que podem ser escritos como combinação convexa dos seus vértices.

(4) O *fecho convexo* de um conjunto $S = \{p_1, p_2, \dots, p_k\}$ de pontos em E^d é conjunto de todos os pontos que são combinação convexa dos pontos em S, ou seja,

$$\operatorname{conv}(S) := \{\alpha_1 p_1 + \cdots + \alpha_k p_k \mid \alpha_1 + \cdots + \alpha_k = 1, \text{ e } \alpha_i \geq 0 \ (i = 1, \dots, k)\}.$$

(5) (Teorema de Carathéodory) O fecho convexo de um conjunto S de pontos em E^d é o conjunto de todos os pontos que são combinação convexa de $d+1$ pontos de S. Logo, o fecho convexo de um conjunto de pontos no plano é o conjunto de combinações convexas dos seus subconjuntos de três pontos (se os pontos forem não colineares o fecho convexo desses três pontos é um triângulo).

(6) O *fecho convexo* de um conjunto S de pontos é a intersecção de todos os conjuntos convexos contendo S. (Esta definição não depende de combinação convexa—logo é menos algébrica e mais *set-theoretical.*)

(7) O *fecho convexo* de um conjunto S de pontos em E^d é a intersecção de todos os semi-espaços contendo S. Um semi-espaço no plano é um *semiplano.*

(8) O *fecho convexo* de um conjunto S de pontos no plano é o menor polígono convexo contendo P que contém S; menor no sentido de que não existe outro polígono P' talque $P \supset P' \supseteq S$.

(9) O *fecho convexo* de um conjunto S de pontos no plano é o polígono convexo P com menor área tal que $P \supseteq S$.

(10) O fecho convexo de um conjunto S de pontos no plano é a união de todos os triângulos determinados por pontos em S.

3. O problema

Nós consideraremos o problema: *dado:* um conjunto finito S de pontos no plano; *encontrar:* o fecho convexo $\mathrm{conv}(S)$ dos pontos em S. Antes de continuarmos devemos ter claro o quê desejamos como resposta dos algoritmos. Podemos estar interessados em pelo menos duas respostas: (i) os pontos 'extremos' do fecho convexo; ou (ii) o polígono que forma a fronteira do fecho convexo. A diferença é que o polígono é representado pelas arestas ligando os pontos extremos na ordem que elas aparecem ao percorrermos a fronteira de $\mathrm{conv}(S)$ no sentido anti-horário. É razoável pensar que só encontrar os pontos extremos seja um problema mais fácil, entretanto isso não é verdade (pelo menos do ponto de vista de complexidade assintótica).

Um ponto v é *ponto extremo* de um conjunto S de pontos se v não pode ser escrito como $\lambda x+(1-\lambda)y$, para $x, y \in S$ e $\lambda > 0$. Logo os *pontos extremos* de um conjunto S de pontos no plano são os pontos do fecho convexo de S que não estão no interior de um segmento contido no fecho convexo. Também podemos pensar os pontos extremos de S como os vértices do fecho convexo nos quais o ângulo interior é estritamente convexo, ou seja, menor que π. Observemos ainda que um ponto v é ponto extremo de um conjunto S se e somente se existe uma reta cuja intersecção com $\mathrm{conv}(S)$ é v, veja Figura 2.

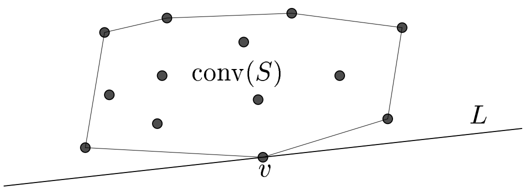


Figura 2. O ponto v é um extremo do conjunto de pontos S.

4. Dois algoritmos ingênuos

Nesta seção veremos dois algoritmos para construir o fecho convexo de um conjunto S de pontos no plano. O primeiro identifica quais dos pontos em S são pontos extremos de S (vértices de $\mathrm{conv}(S)$); mais precisamente, determina quais os pontos de S são pontos não-extremos. O segundo algoritmo encontra as arestas de $\mathrm{conv}(S)$.

4.1. **Pontos não-extremos.** O algoritmo que veremos é sugerido pela Definição 10 acima.

Lema 1. *Um ponto v de um conjunto S é ponto não-extremo de S se e somente se v pertence a algum triângulo cujos vértices são pontos de S distintos de v.* ∎

Baseado no lema acima podemos desenvolver o seguinte algoritmo.

Algoritmo `Pontos-Não-Extremos`(S)
`Entrada:` Um conjunto finito $S = \{p_1, \ldots, p_n\}$ de pontos no plano.
`Saída:` Os pontos extremos de S (vértices de $\mathrm{conv}(S)$).
for cada i **do**
for cada $j, j \neq i$ **do**
for cada $k, k \neq i, k \neq j$ **do**
  for cada $l, l \neq i, l \neq j, l \neq k$ **do**
    if p_l está à esquerda ou sobre (p_i, p_j) e /* `Left`$(p_i, p_j, p_l) \wedge$ */
      p_l está à esquerda ou sobre (p_j, p_k) e /* `Left`$(p_j, p_k, p_l) \wedge$ */
      p_l está à esquerda ou sobre (p_k, pi) /* `Left`(p_k, p_i, p_l) */
      then p_l é não-extremo.

A complexidade do algoritmo acima é $O(n^4)$: existem quatro laços cada um com $O(n)$ iterações: para cada um dos $O(n^3)$ triângulos verificamos quais dos $n-3$ pontos restantes pertencem ao triângulo.

4.2. **Arestas extremas.** É mais fácil identificarmos arestas do fecho convexo do que vértices extremos. Uma arestas xy é uma aresta *extrema* de um conjunto de pontos S se x e y forem vértices extremos e todos os pontos de S estão sobre ou de um mesmo lado da aresta.

Algoritmo `Arestas-Extremas`(S)
`Entrada:` Um conjunto finito $S = \{p_1, \ldots, p_n\}$ de pontos no plano.
`Saída:` As arestas extremas de S, i.e. as arestas de $\mathrm{conv}(S)$.
for cada i **do**
for cada $j, j \neq i$ **do**
  for cada $k, k \neq i, k \neq j$ **do**
    if p_k não está à esquerda ou sobre (p_i, p_j) /* $\neg$ `Left`(p_i, p_j, p_k) */
      then (p_i, p_j) não é aresta de $\mathrm{conv}(S)$.

O algoritmo acima tem complexidade de tempo $O(n^3)$ pois existem três laços encaixados e cada um desses laços fará $O(n)$ iterações: para cada um dos $O(n^2)$ segmentos testamos se o segmento e aresta de $\mathrm{conv}(S)$.

5. O método do embrulho para presente (gift wrapping)

O algoritmo que veremos agora também é conhecido como algoritmo de Jarvis (cf. Jarvis [13], veja também a Seção 3 do Capítulo 35 de Cormen, Leiserson e Rivest [4] e a Seção 3 do Capítulo 3 de Figueiredo e Carvalho [9]). O algoritmo pode ser visto como uma versão 2-dimensional do método sugerido por Chand e Kapur [3] para construir o fecho convexo de um conjunto de pontos em dimensões arbitrárias. A complexidade de tempo do algoritmo é $O(nh)$, onde h é o número de arestas (ou vértices) de $\operatorname{conv}(S)$, i.e. o algoritmo é *output-sensitive*.

Intuitivamente o algoritmo simula o enrolar do conjunto S por um barbante e pode também ser visto como uma melhoria do Algoritmo `Arestas-Extremas`: a idéia é usar a aresta extrema que acabou de ser encontrada para encontrarmos a próxima aresta. O método funciona, pois sabemos que no fecho convexo uma aresta (extrema) esta ligada a outra, que por sua vez esta ligada a uma outra aresta e assim por diante. Para cada aresta extrema encontrada testaremos $O(n)$ candidatas a próxima aresta extrema, logo a complexidade total do algoritmo será $O(nh) = O(n^2)$, que reduz por um fator de n a complexidade do Algoritmo `Arestas-Extremas`.

Para facilitar a exposição vamos supor que os pontos de S estão em posição geral: não existem três pontos em S que sejam colineares. Seja $e = xy$ uma aresta extrema de $\operatorname{conv}(S)$ (suponha y tem o valor de sua abscissa maior do que o valor da abscissa de x; y está a direita de x em relação à reta vertical). Sabemos que $\operatorname{conv}(S)$ possui uma outra aresta com uma extremidade em y. Para todo ponto $z \in S$ nos testamos se todos os pontos de S estão de um mesmo lado em relação à reta que passa pelo segmento yz (candidata a aresta de $\operatorname{conv}(S)$): este teste pode ser feito em tempo $O(n)$—o que continuaria a nos deixar com um algoritmo de complexidade de tempo $O(n^3)$. A observação chave em relação a complexidade de tempo é escolher um ponto z cujo ângulo polar θ_z com relação ao ponto y e a reta passando pelo segmento xy seja mínimo. Isto garante que todos os pontos de S estão de um mesmo lado da reta passando por yz (veja Figura 3) e portanto não precisamos fazer uma verificação para cada ponto de S.

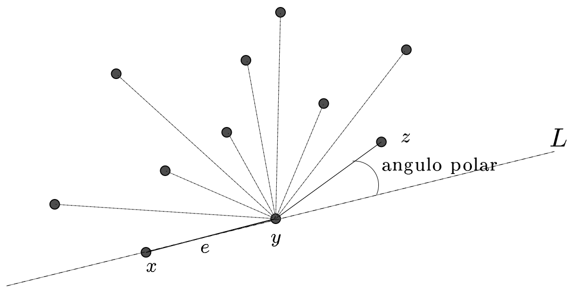


Figura 3. A próxima aresta yz tem o menor ângulo polar com relação à aresta anterior e.

Como começar o algoritmo já que no início não temos nenhuma aresta de $\operatorname{conv}(S)$? Basta começarmos com (digamos) um vértice p_0 mais baixo em S (menor y-coordenada) e encontrarmos o ponto p_1 em S com o menor ângulo polar (sentido anti-horário) com relação ao ponto p_0 e a reta horizontal, veja Figura 4.

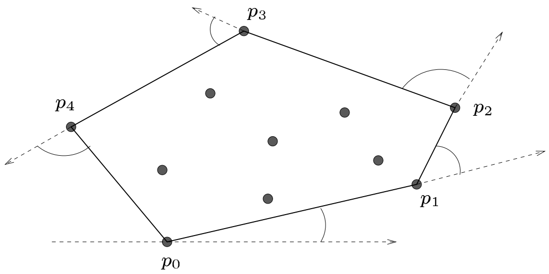


FIGURA 4. Operações feitas pelo algoritmo de Jarvis. O primeiro ponto é um ponto mais baixo. O próximo ponto tem o menor ângulo polar em relação ao ponto p_0 e a reta horizontal.

Algoritmo `Embrulho-Para-Presente`(S)
`Entrada:` Um conjunto finito $S = \{p_1, \ldots, p_n\}$ de pontos no plano.
`Saída:` As arestas extremas de S a medida que elas aparecem
na fronteira de $\text{conv}(S)$ no sentido anti-horário.
Encontre o ponto mais baixo de S. [custa tempo $O(n)$]
Seja i_0 o índice desse ponto e $i \leftarrow i_0$.
Seja L a reta horizontal passando por p_{i_0}.
repeat
  for cada $j, j \neq i$ **do**
    Calcule o ângulo polar θ_j do ponto p_j em relação ao
    ponto p_i e a reta L.
  Seja k o índice do ponto p_k tal que $\theta_k = \min_j \theta_j$.
  Output p_ip_k como uma aresta de $\text{conv}(S)$.
  $i \leftarrow k$.
  $L \leftarrow$ reta passado por p_ip_k.
until $i = i_0$. [Fechou o polígono.]

Se h é o número de arestas extremas de $\text{conv}(S)$ então a complexidade de tempo do algoritmo de Jarvis no pior caso é $O(hn) = O(n^2)$: um custo de $O(n)$ por aresta encontrada. Note que quando o número de arestas extremas de $\text{conv}(S)$ é pequeno, digamos $h = O(\log n)$, então o algoritmo de Jarvis é tão rápido quanto o algoritmo ótimo de Graham que veremos logo adiante (ótimo em termos de complexidade de tempo).
Observação. Notemos que em termos de implementação não precisamos calcular ângulo polar algum. Basta usarmos a primitiva `Left` para encontrar, em cada iteração, o ponto p_k tal que o predicado `Left`(p_i, p_k, p_j) é verdadeiro para todo j distintos de i e k.

6. ALGORITMO QUICKHULL

No livro de Mulmuley [18] geometria computacional é tratada como o estudo de problemas de busca e ordenação em dimensões maiores. Em um certo sentido, esses problemas em dimensões maiores, são generalizações de problemas de ordenação e busca em uma lista de elementos de um universo ordenado (versão unidimensional de ordenação e busca). Deste ponto de vista não é nenhuma surpresa o fato de que algoritmos de ordenação e busca têm sido uma constante fonte de inspiração para desenvolvimento de algoritmos em geometria computacional. O algoritmo que veremos nesta seção, com algumas variações, foi proposto independentemente por várias pessoas quase ao mesmo tempo (cf. Eddy [8], Bykat [2], Green e Silverman [11]). Devido a semelhança com o algoritmo `Quicksort` o algoritmo que veremos foi batizado de `Quickhull` por Preparata e Shamos (cf. Seção 3 do Capítulo 3 de [21]).

Observação. O material desta seção foi extraído do Capítulo 3 de Figueiredo e Carvalho [9], Capítulo 3 de O'Rourke [19] e Capítulo 3 de Preparata e Shamos [21].

6.1. **Descrição do algoritmo: pré-processamento.** A idéia básica por trás do algoritmo é: para o "maioria" dos conjuntos de pontos é mais fácil descartar muitos pontos que estão no interior do fecho convexo e concentrar o trabalho nos pontos que estão próximos da fronteira.

Antes de aplicarmos o algoritmo a um conjunto S de pontos dados o nosso primeiro passo será fazermos um certo pré-processamento (pelo menos nesta versão do algoritmo) e encontrar os pontos extremos nas direções: mais alta, mais baixa, mais à esquerda e mais à direita. (Em caso de empate escolha, por exemplo, o pontos com a menor x-coordenadas ou y-coordenadas, dependendo do caso.) Ligue esses quatro pontos formando um quadrilátero. Note que os pontos no interior do quadrilátero são dispensáveis para efeito de construirmos o fecho convexo, veja Figura 5. Agora nosso problema se reduz a quatro subproblemas independentes: encontrar o fecho convexo dos pontos contidos nas quatro regiões triangulares exteriores ao quadrilátero (concatenando as quatro cadeias correspondentes a essas regiões obteremos o fecho convexo do conjunto de pontos S).

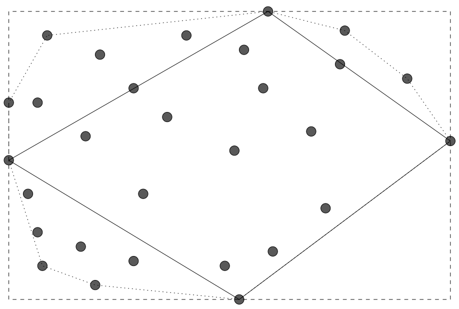


FIGURA 5. `Quickhull` descarta os pontos que estão no quadrilátero formado pelos 4 pontos extremos.

6.2. **Descrição do Algoritmo `Quickhull`.** O Algoritmo `Quickhull` resolve, recursivamente, cada um dos quatro subproblemas obtidos pelo pré-processamento. Chamemos de S o conjunto de pontos de um desses subproblemas e sejam a e b os pontos extremos de S que formam a base do triângulo. O algoritmo encontra um ponto extremo c de S, descarta pontos que estão no interior do triângulo formado pelos pontos a, b e c e particiona o problema em dois subproblemas idênticos menores.

Um pouco mais precisamente, em cada etapa do algoritmo nós temos um segmento ab e um conjunto de pontos S contidos em uma mesmo lado em relação à reta L que passa pelos pontos a e b e contidos na faixa formada pela reta L_a que passa pelo ponto a e é perpendicular ao segmento ab e pela reta L_b que passa pelo ponto b e é perpendicular ao segmento ab. Inicialmente, o algoritmo determina um ponto c que está o mais distante possível do segmento ab e cujo ângulo $\angle cab$ seja máximo. (Como podemos encontrar este ponto?) Note que este ponto é um ponto extremo de $\text{conv}(S)$. De fato, ele é um ponto extremo na direção da reta ortogonal ao segmento ab: os ponto de S estão de um mesmo lado da reta paralela a reta L que passa por c, e além disso c é o ponto mais à esquerda ou à direita nesta reta (dependendo do caso—direita e esquerda é em relação as x-coordenadas dos pontos), logo c não pode ser escrito como combinação convexa de pontos em S, veja Figura 6.

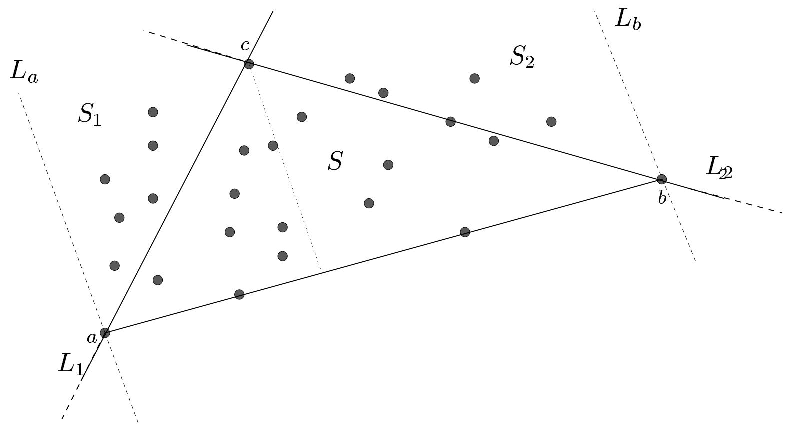


FIGURA 6. `Quickhull` descarta os pontos que estão no triângulo formado pelos pontos a, b e c.

O próximo passo do algoritmo é traçar retas L_1 passando por ac e L_2 passando por bc. Seja S_1 o conjunto de pontos de S que estão do lado oposto ao ponto b em relação à reta L_1 e seja S_2 o conjunto de pontos de S que estão do lado oposto ao ponto a em relação à reta L_2. Não existem pontos que estão simultaneamente em S_1 e S_2 ($S_1 \cap S_2 = \emptyset$) e os pontos de S que não estão nem em S_1 nem em S_2 são pontos que pertencem ao triângulo formado pelos pontos a, b e c. Agora o algoritmo, recursivamente, repete o mesmo procedimento com o segmento ac e o conjunto de pontos S_1 no lugar do segmento ab e o conjunto de pontos S. Analogamente, o mesmo procedimento é repetido para o segmento bc e conjunto de pontos S_2.

Algoritmo `Quickhull` (a, b, S)
`Entrada:` dois pontos distintos a e b e um conjunto de pontos S tal que todo

ponto de S está de um mesmo lado em relação à
reta que passa por a e b.

`Saída:` o fecho convexo de S.

if $S = \emptyset$ **then return** ab
else
$c \leftarrow$ ponto com distância máxima do segmento ab.
$S_1 \leftarrow$ pontos do lado oposto ao ponto b em relaçao à
reta L_2 passando pelo segmento ac.
$S_2 \leftarrow$ pontos do lado oposto ao ponto a em relaçao à
reta L_1 passando pelo segmento bc.
return `Quickhull`(a, c, S_1) concatenado com `Quickhull`(c, b, S_2).

A descrição completa do procedimento consiste do pré-processamento seguido de quatro chamadas ao Algoritmo `Quickhull`.

Observação. As operação primitivas usadas pelo algoritmo são `Left`, `Lefton` e `Area2`. Todas essas primitivas requerem um número pequeno de multiplicações, adições e subtrações (operações elementares).

6.3. **Análise do algoritmo.** Encontrar os quatro pontos extremos que formam o quadrilátero inicial durante o pré-processamento custa $O(n)$ operações elementares, onde n é o número de pontos.

A complexidade do Algoritmo `Quickhull` depende do número de pontos no conjunto S_1 e S_2. Seja $\alpha := |S_1|$ e $\beta := |S_2|$. Se denotaremos por $T(n)$ a função complexidade de tempo do Algoritmo `Quickhull` para encontrar o fecho convexo de um conjunto S com n pontos teremos que

$$T(n) \leq T(\alpha) + T(\beta) + cn,$$

onde c é uma constante universal. O termo cn na desigualdade acima corresponde ao número de operações gastas pelo algoritmo para encontrarmos o ponto c e construirmos os conjunto S_1 e S_2. Note que $\alpha + \beta \leq n$.

Quando $\alpha + \beta = n$ e cada subproblema tem aproximadamente o mesmo tamanho, por exemplo, $\alpha = \lfloor n/2 \rfloor$ e $\beta = \lceil n/2 \rceil$, teremos que

$$T(n) \leq T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + cn = O(n \log n).$$

Logo $T(n) = O(n \log n)$ é o melhor que pode ocorrer se tivermos $\alpha + \beta = n$.

O `Quickhull`, a exemplo do que ocorre com o `Quicksort`, é um algoritmo fácil de implementar que funciona bem quando os pontos de S são escolhidos aleatoriamente no plano. Entretanto, no pior caso, a sua complexidade de tempo é $O(n^2)$. Isso ocorre quando, $\alpha + \beta = n$ e (digamos) $\alpha = 1$ e $\beta = n - 1$. Nesse caso a nossa recorrência toma a forma:

$$\begin{aligned} T(n) &\leq T(n-1) + cn = T(n-2) + c(n-1) + cn \\ &= T(n-3) + c(n-2) + c(n-1) + cn = \cdots \end{aligned}$$

Repetindo a expansão nos chegaremos a

$$T(n) \leq c + c2 + \cdots + c(n-2) + c(n-1) + cn = O(n^2).$$

Logo, o `Quickhull` é um algoritmo que na prática é geralmente rápido, mas seu pior caso é quadrático (fica como exercício encontrar um conjunto de pontos genéricos onde o desempenho do `Quickhull` é realmente quadrático—por genérico queremos dizer uma construção de uma instância que 'funcione' para um valor de n arbitrariamente grande).

7. O Algoritmo de Graham

Segundo O'Rourke (cf. [19], pg. 80) talvez o primeiro artigo na área de geometria computacional tenha sido o de Graham [10] que apresenta um algoritmo para encontrar o fecho convexo de um conjunto de pontos no plano.[1]

O algoritmo de Graham constrói o fecho convexo de um conjunto de pontos S em duas fase. A primeira fase consiste de um pré-processamento onde primeiro um ponto extremo p_0 de S é selecionado e depois os pontos restantes são ordenados angularmente ao redor de p_0. Na segunda fase o algoritmo iterativamente processa os vértices produzindo uma seqüência de fechos convexos que converge para $\mathrm{conv}(S)$.

Observação. O material desta seção foi extraído do Capítulo 35 de Cormen, Leiserson e Rivest [4], Capítulo 3 de Figueiredo e Carvalho [9], Capítulo 6 de Laszlo [15], Capítulo 3 de Preparata e Shamos [21], Capítulo 4 de Resende e Stolfi [6], e Capítulo 3 de O'Rourke [19].

7.1. **Pré-processamento: 'Presorting phase'.**

Dado um conjunto de pontos S o pré-processamento do algoritmo de Graham consiste em encontrar um ponto extremo p_0 de S. Escolheremos p_0 como sendo o ponto em S com a menor y-coordenada (mais baixo) e em caso de empate escolheremos, entre os pontos com a menor y-coordenada, o ponto com a maior x-coordenada (mais a direita). Este passo do algoritmo pode ser feito em tempo $O(n)$.

Numa segunda etapa os pontos de $S \setminus \{p_0\}$ são ordenados em relação ao ângulo polar que estes formam ao redor do ponto p_0. Se dois pontos tem o mesmo ângulo polar (i.e. p_0 e esses dois pontos são colineares) então o ponto que está mais próximo de p_0 pode ser descartado, veja Figura 7.

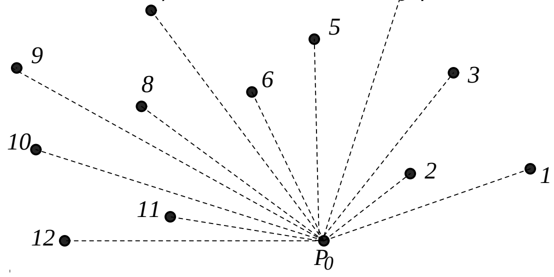


Figura 7. Pontos de S ordenados em relação ao ângulo polar que estes formam ao redor do ponto p_0.

[1] O 'talvez' nesta frase é pelo fato de Toussaint [23] ter encontrado um artigo mais antigo que contém muitas das idéias que apareceram no artigo de Graham [10].

O custo total desta fase de pré-processamento é dominado pelo custo da ordenação dos pontos que é $O(n \log n)$.

7.2. **Construção do fecho convexo: 'The hull finding fase'.** O pré-processamento tem o efeito de simplicar a fase de construção do fecho convexo. Nesta fase os pontos são examinado na ordem que estes foram colocados pelo pré-processamento. Sejam $p_1, p_2, \ldots, p_{n-1}$ os pontos de $S \setminus \{p_0\}$ ordenados em relação ao ângulo polar que esses pontos formam ao redor do ponto p_0. Seja ainda $S_i := \{p_0, p_1, \ldots, p_i\}$ $(i = 1, \ldots, n-1)$. Durante esta fase cada novo ponto p_i é um ponto extremo de $\mathrm{conv}(S_i)$; entretanto alguns pontos que eram pontos extremos de $\mathrm{conv}(S_{i-1})$ podem não ser pontos extremos de $\mathrm{conv}(S_i)$.

O algoritmo de Graham mantém em cada iteração i o fecho convexo dos ponto S_i. A Figura 8 mostra o algoritmo de Graham em ação.

Considere a iteração que processa o ponto p_7. Devido a ordenação angular em volta de p_0 é claro que o ponto p_7 é um ponto extremo de $\mathrm{conv}(S_7)$. Mas como saber quais os pontos que eram pontos extremos de $\mathrm{conv}(S_6)$ e não são pontos extremos de $\mathrm{conv}(S_7)$? Para responder a esta pergunta usaremos o fato de que ao percorrermos a fronteira do fecho convexo no sentido anti-horário devemos virar à esquerda em cada ponto extremo. Como indo de p_5 até p_7 passando por p_6 viramos à direita em p_6 sabemos que p_6 não pode ser um ponto extremo de $\mathrm{conv}(S_7)$. Analogamente, o ponto p_5 não é um ponto extremo de $\mathrm{conv}(S_7)$ pois indo de p_4 até p_5 e depois de p_5 até p_7 viramos à direita em p_5. Logo, na iteração correspondente ao ponto p_7 os pontos p_6, p_5 e p_4 serão removidos da lista de pontos extremos de $\mathrm{conv}(S_7)$ e o ponto p_7 será incluído nesta lista. Com isto completamos mais uma iteração do algoritmo.

Note que na descrição da iteração do algoritmo nos percorremos a lista de vértices pertencentes ao fecho convexo corrente do final para o começo (considerando que p_0 é o começo da lista). Por isto, ao descrevermos o algoritmo usaremos uma pilha T para guardar os pontos do fecho convexo corrente. Depois da última iteração do algoritmo a pilha T contém os vértices extremos de $\mathrm{conv}(S)$ em ordem anti-horária — a medida que eles foram encontrados ao percorrermos a fronteira de $\mathrm{conv}(S)$. Abaixo encontra-se uma decrição de todo o algoritmo.

Algoritmo `Graham-Scan`(S)
`Entrada:` Um conjunto $S = \{p_0, \ldots, p_{n-1}\}$ com $n > 3$ pontos do plano.
`Saída:` O fecho convexo dos pontos em S.

1. Seja p_0 o ponto em S com a menor y-coordenada, em caso de empate tome o ponto mais a direita. [custo $O(n)$]
2. Seja $p_1, \ldots, p_{n-1}$ os pontos de $S \setminus p_0$ ordenados em relação ao ângulo polar que estes pontos formam ao redor do ponto p_0. Se mais de um ponto tem o mesmo ângulo, descarte todos menos o que esta mais distânte de p_0. [custo $O(n \log n)$]
3. $T \leftarrow \emptyset$.
4. Empilhe p_0, p_1 e p_2, nesta ordem, na pilha T.
5. $i \leftarrow 3$.

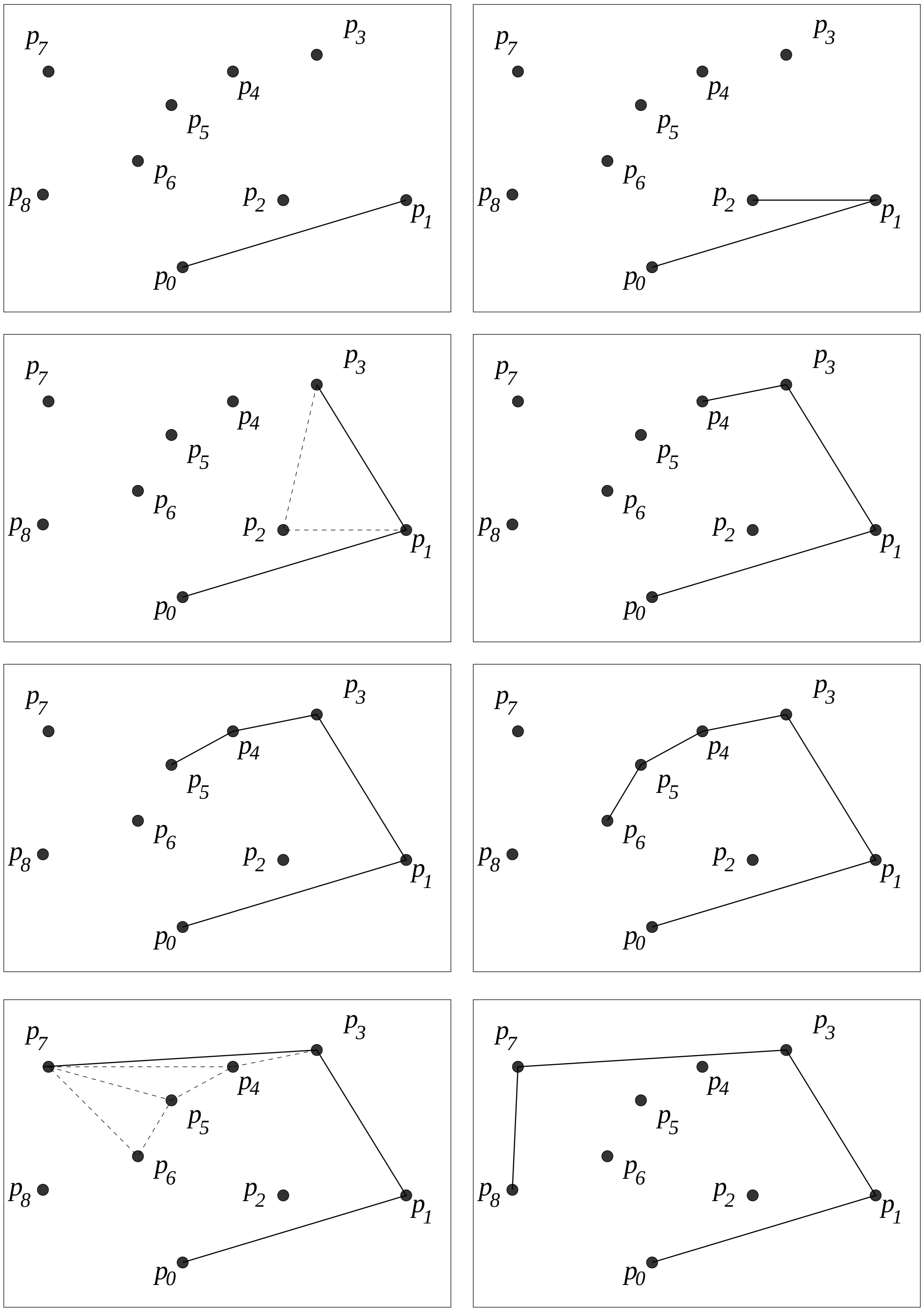


FIGURA 8. O Algoritmo de Graham em ação.

6. **while** $i < n$ **do**
   Seja p_t o ponto que está no topo de T e p_{t-1} o ponto que está embaixo de p_t em T.
7. **if** p_i está à esquerda de (p_{t-1}, p_t)

8. **then** Empilhe p_i em T e $i \leftarrow i+1$
9. **else** Desempilhe p_t de T.
10. **return** T.

7.3. **Análise do algoritmo.** A complexidade de tempo do Algoritmo de Graham é $O(n \log n)$, onde n é o número de pontos do conjunto S.

A Linha 1 do algoritmo tem um custo de $O(n)$. A ordenação da Linha 2 pode ser feita em $O(n \log n)$ se usarmos o Algoritmo `Heapsort` ou `Mergesort`. Logo a fase de pré-procesamento tem um custo total de $O(n \log n)$.

As Linhas 3, 4 e 5 são executadas em tempo constante ($O(1)$).

O número de operações Desempilha é menor do que o número de operações Empilha (já que p_0, p_1 e p_{n-1} nunca serão desempilhados de T). Como o número total de operações Empilha executadas durante todo o algoritmo será n então o número de operações Desempilha realizadas será menor que n. Logo, o tempo total gasto com as operações Empilha e Desempilha ao longo da execução do algoritmo é $O(n)$ (já que cada uma destas operações pode ser feita em tempo constante). Isto também implica que o número total de execuções do comando **while** da Linha 6 será $O(n)$ (mais precisamente, no máximo $2n-3$).

Assim, concluímos que o tempo total gasto pelo algoritmo para encontrar o fecho convexo de um conjunto com n pontos é $O(n \log n)$. Como veremos mais adiante este algoritmo é ótimo.

7.4. **Corretude do algoritmo.** Notemos primeiro que p_0, como foi escolhido, é um ponto extremo de $\text{conv}(S)$. Observemos ainda que se q é um ponto de S então qualquer ponto em $S \setminus \{p_0, q\}$ que pertença ao segmento p_0q não é um ponto extremo de $\text{conv}(S)$ (pois este ponto seria combinação convexa dos pontos q e p_0). Estas observações justificam a Linha 1 e 2 do algoritmo.

Seja $S_i := \{p_0, p_1, \ldots, p_i\}$ $(i = 2, \ldots, n-1)$. Ao final da execução da Linha 4 teremos que a pilha T contém os pontos extremos de $\text{conv}(S_2)$. A corretude do algoritmo é baseada no seguinte teorema.

Teorema 2 (Invariante 1)**.** *Imediatamente após qualquer atualização da variável i (Linha 5 e Linha 6) temos que os pontos que estão na pilha T são pontos extremos de $conv(S_{i-1})$ em ordem anti-horária.*

Note que o teorema acima implica imediatamente a corretude do algoritmo. De fato, quando a execução do algoritmo chega na Linha 10 temos que $i = n$ e teremos portanto que S contém, em ordem anti-horária, os vértices extremos de $\text{conv}(S_{n-1}) = \text{conv}(S)$. Para provarmos este teorema faremos uso do seguinte lema.

Lema 3 (Invariante 2)**.** *O algoritmo de Graham mantém o invariante de que os pontos na pilha T são os vértices, em ordem anti-horária, de um polígono convexo.*

Prova. O lema é verdadeiro logo após a execução da Linha 4 do algoritmo, já que p_0, p_1 e p_2 são vértices de um polígono convexo. Se os pontos em T são os vértices em ordem anti-horária de um polígono convexo então após desempinharmos um ponto do topo de T teremos que os pontos restantes satisfazem a mesma propriedade. Logo, desempilhar um vértice preserva o invariante. (Isto corresponde a cortar uma orelha do polígono.)

Antes de examinarmos o que ocorre quando o algoritmo empilha um novo ponto em T examinaremos uma propriedade geometrica. (Veja a Figura 9.) Seja P um polígono convexo e escolha uma aresta qualquer $p_r p_t$ de P. Considere a região limitada por $p_r p_t$ e pelas extensões das duas arestas adjacentes. Se nós adicionarmos um ponto p_s à esta região e trocarmos a aresta $p_r p_t$ de P pelas arestas $p_r p_s$ e $p_s p_t$ o polígono resultante é convexo.

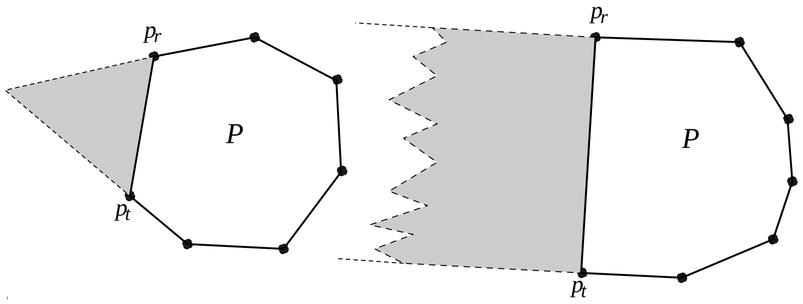


FIGURA 9. Colocar um vértice na região sombreada faz com que o novo polígono resultante continue a ser convexo.

Agora consideremos o ponto p_i que é empilhado em T. Seja p_t o vértice no topo de T antes de empilharmos p_i e seja p_{t-1} o predecessor de p_t em T. Afirmamos que p_i deve estar na região sombreada ilustrada na Figura 9. De fato, como estamos examinando os pontos em ordem crescente do ângulo polar que eles formam com p_0 temos que p_i está à esquerda de (p_0, p_t) e pela escolha de p_0 sabemos que p_i também está à esquerda de (p_0, p_1). Além disso, como p_i foi empilhado em S temos, pela comparação da Linha 6 do algoritmo, que p_i está à direita de (p_{t-1}, p_t). Logo, p_i está na região sombreada da figura. Com isto completamos a prova do lema. ∎

Prova do Teorema 2. Seja k o valor que acaba de ser recebido pela variável i. Quando $k = 3$ (atualização feita pela Linha 5) temos que T contém p_0, p_1 e p_2 que são os pontos extremos de $\text{conv}(S_2)$. (Observe que o fato destes pontos serem pontos extremos seguem da execução das Linhas 1 e 2 do algoritmo.)

Suponha agora que $k > 3$ e que para valores de i menores do que k o teorema vale. E considere a execução da Linhas 6, 7, 8 e 9 do algoritmo enquanto a variável i tem o valor $k-1$. Mostraremos primeiro que os pontos desempilhados de S na Linha 9 não são pontos extremos de $\text{conv}(S_{k-1})$.

Sejam p_t e p_{t-1} os pontos no topo da pilha T imediatamente antes do ponto p_t ser desempilhado durante a execução da Linha 9. Como p_t foi desempilhado temos então que p_i está à esquerda de (p_{t-1}, p_t); ou seja, viramos à direita no ponto p_t quando estavamos indo de p_{t-1} até p_i passando por p_t. Como estamos processando os pontos em ordem crescente do ângulo polar que eles formam com p_0 temos que o ponto p_t pertence ao triângulo $\triangle p_0 p_{t-1} p_i$ e portanto não pode ser um ponto extremo de $\text{conv}(S_{k-1})$. Veja Figura 10.

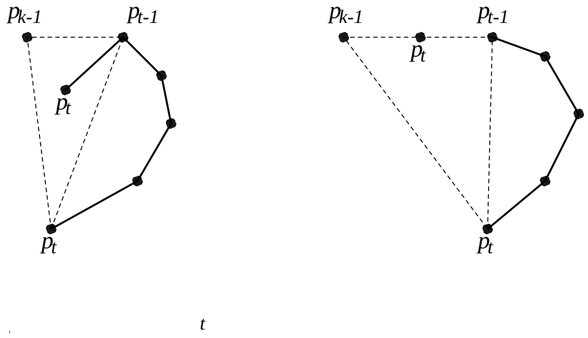


FIGURA 10. Em ambas as situações ilustradas acima o ponto p_t não é ponto extremo de $\text{conv}(S_{k-1})$.

Agora, pelo Lema 3, temos que quando p_i é empilhado em T (imediatamente antes de atualizarmos i) os pontos em T são vértices de um polígono convexo. Como T só contém pontos de S_{k-1} e esses pontos são vértices de um polígono convexo temos que quanto i recebe o valor k os pontos em T são todos os pontos extremos de S_{k-1}. ∎

8. Um algoritmo incremental

Depois de termos visto um algoritmo que constrói o fecho convexo de um conjunto de n pontos em tempo $O(n \log n)$ pode parecer sem sentido continuarmos explorando este problema. Entretanto, existem pelo menos duas motivações para continuarmos: extensão do problema para o espaço tridimensional (ou dimensão maior) e o estudo de outros métodos em geometria computacional. O Algoritmo de Graham não tem nenhuma extensão natural para o problema tridimensional: o algoritmo depende fortemente da ordenação angular, que não tem (ou não se conhece!?) nenhuma contraparte tridimemsional. O algoritmo que veremos nesta seção tem uma extensão natural para o problema tridimensional.

Um *algoritmo incremental* examina os dados do problema, um item por vez, enquanto uma solução corrente dos itens que já foram vistos até o momento é mantida. Em cada iteração, o próximo item é examinado e processado e a solução corrente para o problema é atualizada para satisfazer este novo item. (Lembre-se dos algoritmo de ordenação por inserção: inserção direta e inserção binária.)

Observação. O material desta seção foi extraído do Capítulo 35 de Cormen, Leiserson e Rivest [4], Capítulo 6 de Laszlo [15], Capítulo 3 de Mulmuley [17] Capítulo 3 de Preparata e Shamos [21], Capítulo 4 de Figueiredo e Carvalho [9], e Capítulo 3 de O'Rourke [19].

8.1. **O algoritmo.** A idéia básica do algoritmo é: examinar um ponto por vez, a cada passo construir o fecho convexo dos pontos já examinados, e usar esse fecho convexo no exame do próximo ponto. Dividir o problema desta maneira simplifica bastante o trabalho: a cada passo precisamos incluir um único ponto no fecho convexo do pontos já examinados.

Seja $S := \{p_0, p_1, \ldots, p_{n-1}\}$ um conjunto de pontos que não contém três pontos colineares, ou seja o conjunto de pontos está em posição geral (para facilitar a exposição).

Algoritmo `Incremental`
`Entrada:` Um conjunto de pontos $S = \{p_0, p_1, \ldots, p_{n-1}\}$.
`Saída:` O fecho convexo do conjunto S.
1. $P_2 \leftarrow \text{conv}(\{p_0, p_1, p_2\})$.
2. **for** $k = 3, \ldots, n-1$ **do**
3. $\quad P_k \leftarrow \text{conv}(P_{k-1} \cup \{p_k\})$.
4. **return** P_{n-1}.

Como mostra o algoritmo acima, o primeiro fecho convexo é o triângulo $\text{conv}(\{p_0, p_1, p_2\})$. Seja $Q := P_{k-1}$ e $p = p_k$. O problema de construir $\text{conv}(Q \cup \{p\})$ (Linha 3 do algoritmo) cai em um dos dois casos a seguir:

(1) $p \in Q$. Uma vez que foi determinado que $p \in Q$, o ponto p pode ser descartado e sabemos que $\text{conv}(Q \cup \{p\}) = Q$. Para decidir se $p \in Q$ podemos usar a função (primitiva) `Lefton`. Este teste pode ser feito em tempo $O(n)$ (como vocês fizeram no Exercício 4 da Lista 3): $p \in Q$ se e somente se está à esquerda ou sobre cada aresta (orientada) do polígono Q.

(2) $p \notin Q$. Se em algum momento p está à direita de uma aresta de Q então nós temos que construir $\text{conv}(Q \cup \{p\})$. O que faz esta tarefa mais fácil é que nós temos que encontrar somente as duas retas que passam pelo ponto p e que são tangentes ao polígono Q (como é ilustrado na Figura 11) e modificar o fecho convexo de maneira conveniente. Pela nossa hipótese de posição geral cada reta que passa por p e tangência Q tem um único ponto de intersecção com Q: um vértice p_i do polígono Q.

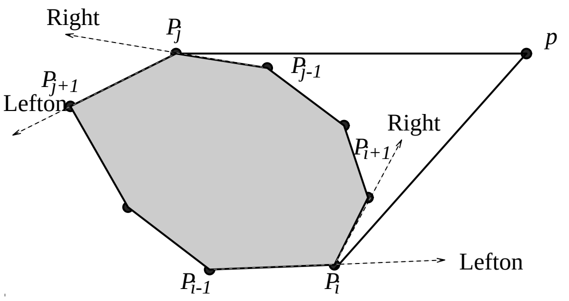


FIGURA 11. Retas tangentes a Q que passam por p; 'Lefton' significa que o ponto p está a esquerda ou é colinear ao vetor indicado, e 'Right' que p está à direita.

Examinando a Figura 11 nós vemos que é possível usarmos o resultado da primitiva `Lefton` para encontrar o ponto de tangência mais baixo p_i. Note que p esta à esquerda de (p_{i-1}, p_i) e à direita de (p_i, p_{i+1}). Para o ponto de tangência mais alto o raciocinio é análogo. Logo os dois vértices de tangência podem ser encontrados através da mesma série de testes **Lefton** usados para decidir se $p \in Q$. Logo o caso $p \notin Q$ também pode ser tratado em tempo $O(n)$.

Algoritmo `Vértices-Tangentes`(Q, p)
`Entrada:` Um polígono convexo Q com m vértices e um ponto p tal que $p \notin Q$.
`Saída:` Os dois vértices de tangência de Q em relação a p.
1. **for** $i = 0, 1, \ldots, m-1$ **do**
2. **if** `Lefton`$(p_{i-1}, p_i, p) \neq$ `Lefton`(p_i, p_{i+1}, p)
3. **then** p_i é vértice de tangência.

Precisamos agora só especificar como construir o novo polígono a partir do polígono Q, do ponto p e dos pontos de tangência (digamos) p_i e p_j. Através da Figura 11 nós vemos que o fecho (polígono) convexo corrente será

$$(p_0, p_1, \ldots, p_{i-1}, p_i, p, p_j, p_{j+1}, \ldots, p_{n-1}).$$

Se os vértices extremos do fecho convexo (em ordem anti-horária) estiverem representados por uma lista circular duplamente ligada então a atualização do fecho poderá ser feita através de uma simples seqüência de remoções e uma inserção.

8.2. **Análise do algoritmo.** Podemos decidir em tempo $O(n)$ se $p \in Q$. No caso de p não ser um ponto em Q os vértices de tangância podem ser encontrados em tempo $O(n)$. O número de remoções da lista ligada circular que armazena os vértices extremos de Q pode ser atualizada através de operações de remoções e inserções. Como o número total de remoções durante a execução de todo o algoritmo é menor que o número total de inserções que por sua vez é igual a n, temos que todas as inserções e remoções podem ser feitas em tempo $O(n)$. Portanto a complexidade de tempo do Algoritmo `Incremental` é $O(n^2)$.

Com um pouco de mais de esforço a complexidade de tempo do Algoritmo `Incremental` pode ser reduzida para $O(n \log n)$.

9. Um algoritmo probabilístico

Algoritmos incrementais para construir o fecho convexo de um conjunto de pontos é usualmente simples de ser implementado e pode ser generalizado para dimensões maiores. Embrulho para presente é outro algoritmo que freqüentemente é usado em dimensões maiores, mas o seu desempenho no pior caso não é tão bom mesmo no plano. O algoritmo incremental, como vimos, é baseado em inserirmos os pontos um após outro e incrementalmente atualizarmos o fecho convexo. Descreveremos a seguir uma versão probabilística de um algoritmo incremental para computar o fecho convexo de um conjunto de pontos no plano. (Sinto que não deveriamos chamar o algoritmo que veremos de incremental, já que, como veremos, no primeiro passo do algoritmo fazemos um pré-processamento com os pontos, mas na falta de nome melhor fica assim mesmo.)

Como em outros algoritmos probabilísticos (*randomized algorithms*), a complexidade de tempo esperada *depende* da ordem aleatória de inserção dos pontos, mas *não* depende do conjunto de pontos.

Observação. O material desta seção foi extraído de Khuller [14] e do Capítulo 9 de Motwani e Raghavan [16].

9.1. **Descrição algoritmo.** A descrição do algoritmo encontra-se logo abaixo.

Algoritmo `IncrementalProbabilístico`(S)
`Entrada:` Um conjunto de pontos $S = \{p_0, p_1, \ldots, p_{n-1}\}$.
`Saída:` O fecho convexo do conjunto S.
1. Permute os pontos aleatoriamente. [gasta tempo $O(n)$]
   Seja $(p_0, p_1, \ldots, p_{n-1})$ está seqüência.
2. $P_2 \leftarrow \text{conv}(\{p_0, p_1, p_2\})$.
3. **for** $k = 3, \ldots, n-1$ **do**
4. $\quad$ $P_k \leftarrow \text{conv}(P_{k-1} \cup \{p_k\})$.
5. **return** P_{n-1}.

A execução do algoritmo é feita como segue. Suporemos que cada ponto que ainda precisa ser inserido tem um apontador para uma aresta do fecho convexo corrente de tal forma que este ponto está à direita desta aresta (no semiplano que não contém o fecho convexo corrente — esta aresta é uma prova de que o ponto não está no interior do fecho convexo corrente). Se um ponto está no interior do fecho convexo então ele é rotulado como INTERIOR e não precisaremos mais nos preocupar com ele. Faremos a associação de pontos com arestas considerando um ponto O que pertence ao interior do fecho convexo corrente (tome, por exemplo, como O o centro de gravidade do triângulo inicial P_2). Imagine que um segmento de reta está ligando cada ponto p_k fora de P_{k-1} ao ponto O. Este segmento de reta intersecta alguma aresta do fecho convexo P_{k-1}, associe p_k a esta aresta. Ademais, para cada aresta do fecho convexo manteremos os pontos associados a ela. Para cada aresta e denotaremos por $L(e)$ o conjunto dos pontos associados a e da maneira descrita acima, e denotaremos por $l(e)$ a cardinalidade do conjunto $L(e)$.

Por exemplo, na Figura 12 (a) mostramos as arestas associadas a cada ponto através de um arco ligando o ponto à aresta. Por exemplo, $L(e) = \{p_{10}, p_{15}\}$ e $l(e) = 2$.

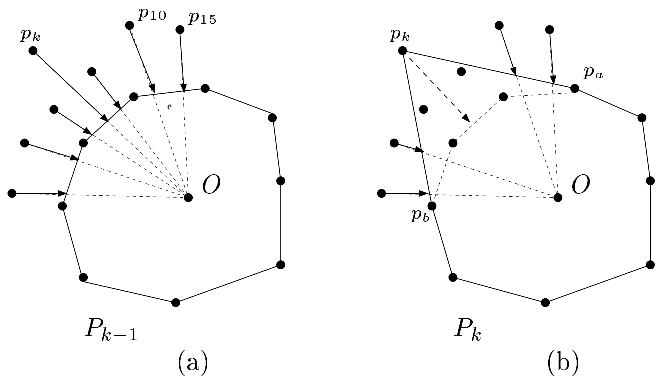


FIGURA 12. Fecho convexo incremental.

Quando um novo ponto p_k e adicionado ao fecho convexo corrente P_{k-1}, para formar o novo fecho convexo P_k, nós seguimos o apontador de p_k até a aresta associada a p_k (veja Figura 12). Notemos que esta aresta não pode mais fazer parte do novo fecho convexo. Nós caminharemos no sentido horário e anti-horário ao redor de P_{k-1}, a partir da aresta associada a p_k, para determinarmos os pontos de tangência p_a e p_b entre p_k e P_{k-1}. Ligaremos p_k a estes pontos p_a e p_b e removeremos do fecho as arestas que foram percorridas até encontrarmos os pontos de tangência para obtermos o fecho convexo novo (veja Figura 12(b)). O polígono resultante é P_k. Existe ainda uma coisa que precisa ser feita. Para cada aresta e do fecho convexo antigo que foram removidas, precisamos examinar cada ponto em $L(e)$ e determinamos se o ponto:

(1) deve ser associado à $p_a p_i$ em P_k;
(2) deve ser associado à $p_i p_b$ em P_k;
(3) pertence ao interior de P_k.

No primeiro caso o ponto será transferido para $L(p_a p_i)$, no segundo caso será transferido para $L(p_i p_b)$ e no terceiro caso o ponto deverá ser marcado como INTERIOR e poderemos nos esquecer dele.

9.2. **Análise do algoritmo.** Não é difícil verificarmos que o algoritmo está correto. Qual é a complexidade de tempo esperada para este algoritmo? Existem três tipos de operações básicas que o algoritmo faz a cada iteração:

Destruição: o algoritmo destrói arestas do fecho convexo antigo.
Criação: o algoritmo cria duas novas arestas novas.
Reclassificação: o algoritmo reclassifica alguns pontos de acordo com as arestas que passaram a ser associadas a eles.

Observemos que cada aresta destruída deve ter sido criada anteriormente pelo algoritmo. Assim, o tempo total gasto em destruir arestas (removê-la do fecho convexo) será assintoticamente limitado pelo tempo gasto para criar arestas. Como em cada iteração são criadas no máximo 2 arestas e temos $n-3$ iterações, o tempo total gasto em criar e destruir arestas é $O(n)$ (o polígono que descreve o fecho convexo pode ser representado por uma lista circular duplamente ligada, assim as operações de inserir e remover arestas ao fecho convexo corrente poderão ser realizadas em tempo constante).

A única operação que resta ser analisada é reclassificação. Afirmamos que o tempo total esperado com a operação de reclassificação é $O(n \log n)$. Em outras palavras, em média, cada ponto é reclassificado $O(\log n)$ vezes. Ao reclassificarmos um ponto ele pode passar a fazer parte do interior do fecho convexo corrente e ser rotulado como INTERIOR. Entretanto, isto só ocorre uma vez para cada ponto e portanto o tempo total gasto em reclassificações deste tipo é $O(n)$. Portanto, nós necessitamos apenas nos preocupar em contar o número esperado de reclassificações de pontos que permanecem no exterior do fecho convexo corrente.

Consideremos o *último* ponto inserido ao fecho convexo P_k. Com probabilidade $1/(k+1)$ cada ponto em $\{p_0, \dots, p_k\}$ foi o último inserido. Seja p este último ponto. Se p pertence ao interior de P_k então nenhum trabalho de reclassificação será necessário. Caso contrário, p será um vértices do fecho convexo corrente P_k. Sejam e e e' as duas arestas em P_k incidentes a p. Observemos que os pontos que terão que ser reclassificados são somente os pontos em $L(e)$ e $L(e')$. Assim, o custo de reclassificação será proporcional a $l(e) + l(e')$ se p foi o último ponto a ser inserido. Para determinar o custo esperado nós somamos esses valores sobre todos os $k+1$ possíveis último ponto e multiplicamos pela probabilidade $1/(k+1)$. Observemos que, na soma, cada aresta e colabora com $l(e)$ duas vezes (uma para o ponto em uma extremidade e outra para o ponto na outra extremidade). Logo, o custo esperado com reclassificação na construção de P_k é

$$\frac{1}{k+1}\sum_{e\in H_k} 2l(e) = \frac{2}{k+1}\sum_{e\in H_k} l(e).$$

Como a soma de $l(e)$ conta cada um dos $n-1-k$ pontos restantes no máximo duas vezes (exatamente duas vezes se o ponto não pertence a fecho convexo P_k), esta soma é limitada por $2(n-1-k)$ e portanto o tempo esperado da reclassificação é

$$\frac{2(n-1-k)}{k+1} \leq \frac{2n}{k+1}.$$

Agora, para obter o tempo total gasto pelo algoritmo com reclassificações, nós fazermos a soma sobre todas as iterações, e obtemos

$$T(n) = \sum_{k=3}^{n-1} \frac{2n}{k+1} = 2n\sum_{k=3}^{n-1}\frac{1}{k+1} = O(n\log n).$$

Este custo domina o tempo de execução do algoritmo. Com isto terminamos a análise do algoritmo. O mesmo algoritmo pode ser generalizado para o espaço tridimensional sem muitas dificuldades, apesar de que cada ponto não será ligado por duas tangentes, mas sim por um cone poligonal que tem como 'apex' o novo ponto que foi inserido. A análise é mais complicada pois o número de faces nesse cone pode não ser constante.

Teorema 4. *O Algoritmo* `IncrementalProbabilístico` *constrói o fecho convexo de um conjunto de n pontos no plano em tempo esperado $O(n\log n)$.* ∎

10. Um algoritmo por divisão-e-conquista

Muitos algoritmos têm a seguinte estrutura recursiva: para resolver um determinado problema eles fazem uma ou mais chamadas recursivas para resolver o mesmo problema em instâncias menores. Esses algoritmos tipicamente seguem o seguinte método de divisão-e-conquista: quebram o problema em diversos subproblemas que são similares ao problema original mas de tamanhos menores; resolvem recursivamente esses subproblemas menores; e finalmente combinam as soluções desses problemas menores para obter uma solução do problema para a instância original. (Veja Capítulo 1, Seção 3 de Cormen, Leiserson e Rivest [4].)

Já vimos esse método antes durante o andamento da disciplina—lembre-se do algoritmo de complexidade de tempo $O(n\log n)$ para o Problema do Par Mais Próximo, do algoritmo de complexidade de tempo $O(n^4)$ para triangularização de polígonos e do Algoritmo `QuickHull`. Um exemplo bem-conhecido de algoritmo que usa este método é o `MergeSort`.

Mais especificamente, algoritmos de divisão-e-conquista têm três passos em cada nível da recursão:

Dividir: O problema é dividido em subproblemas de tamanho menor.

Conquistar: Os subproblemas são resolvidos recursivamente. Entretanto, se algum subproblema é suficientemente pequeno ele é resolvido diretamente.

Combinar: As soluções dos subproblemas são combinadas para obtermos uma solução do problema original.

Nesta seção nos veremos uma algoritmo de divisão-e-conquista que encontra o fecho convexo de um conjunto de n pontos no plano em tempo $O(n\log n)$ (um algoritmo ótimo para o problema). A técnica de divisão-e-conquista é a única conhecida que ao ser estendida para o Problema do Fecho Convexo tri-dimensional atinge a mesma complexidade de tempo, i.e. $O(n\log n)$. Na realidade, a primeira vez que este método foi utilizado para resolver o Problema do Fecho Convexo foi em 1977 por Preparata e Hong [20] com o objetivo de encontrar o fecho convexo tri-dimensional. Por causa da sua similaridade com o `MergeSort`, o algoritmo que veremos é muitas vezes chamado de `MergeHull` (veja, por exemplo, Figueiredo e Carvalho [9], Laszlo [15] e Preparata e Shamos [21]).

Observação. O material desta seção foi extraído do Capítulo 1 de Cormen, Leiserson e Rivest [4], Capítulo 4 de Resende e Stolfi [6], Capítulo 3 de Figueiredo e Carvalho [9], Capítulo 8 de Laszlo [15], Capítulo 3 de O'Rourke [19], e Capítulo 3 de Preparata e Shamos [21].

10.1. **A descrição do algoritmo.** Para simplicar a exposição assumiremos que qualquer três pontos são não-colineraes (posição geral) e qualquer dois pontos não têm a mesma x-coordenada. O algoritmo está descrito logo abaixo. Note que para que o fecho convexo seja contruído em tempo $O(n \log n)$ necessitamos que a fase `Combinar` seja feita em tempo $O(n)$.

Algoritmo `MergeHull`(S)
`Entrada:` Um conjunto S de pontos no plano.
`Saída:` O fecho convexo de S.

(1) **if** $|S| \leq k_0$, onde k_0 é uma constante pequena, construa o fecho convexo diretamente por algum método e **return**.
(2) (`Dividir`) Particione o conjunto S em conjuntos S_1 com $\lceil n/2 \rceil$ pontos e S_2 com $\lfloor n/2 \rfloor$ pontos.
(3) (`Conquistar`) Recursivamente construa o fecho convexo de S_1 e S_2.
(4) (`Combinar` ou `Merge`) Tendo os fechos convexos de S_1 e S_2 construa o fecho convexo de $S_1 \cup S_2$.

10.2. **Pré-processamento: 'sorting step'.** Para facilitar a fase **`Conquistar`**, antes de aplicarmos o algoritmo **`MergeHull`**, ordenaremos o conjunto de pontos dado em relação as suas x-coordenadas.

10.3. **Dividir.** O conjunto de pontos S será divido em dois conjunto em relação a mediana das x-coordenas. Graças ao pré-processamento este passo pode ser feito rapidamente. Em outras palavras, o conjunto S será divido com relação a uma reta vertical: o conjunto de pontos S_1 será formado por os pontos de S que têm x-coordenada (digamos) menor ou igual a mediana das x-coordenadas dos pontos e o conjunto S_2 será formado pelos demais pontos de S (pontos à direita de reta vértical passando pela mediana das x-coordenadas). Note esta escolha de conjuntos S_1 e S_2 juntamente com a nossa hipótese de que S não tem dois pontos com a mesma x-coordenada garante que os fechos convexos de S_1 e S_2 são disjuntos, i.e. $\mathrm{conv}(S_1) \cap \mathrm{conv}(S_2) = \emptyset$.

10.4. **Conquistar.** O passo conquistar corresponde as chamadas recursivas onde cada subproblema será resolvido independentemente. A base da recursão é atinginda quando o conjunto de pontos em questão tem no máximo k_0 pontos, onde k_0 é uma constante. Na base da recursão o fecho convexo desses no máximo k_0 pontos pode ser construído por qualquer algoritmo, mesmo um algoritmo quadrático. Por exemplo, se $k_0 = 3$ o fecho convexo é um triângulo pela nossa hipótese de posição geral.

10.5. **Combinar: 'the merging step'.** A maior parte do trabalho deste algoritmo de divisão-e-conquista está neste passo (o mesmo fenômeno se virificou quando estudamos outros algoritmos de divisão-e-conquista). Precisamos ser cuidadoso para realizarmos este passo em tempo linear: um algoritmo ingênuo pode atingir a complexidade de tempo $O(n^2)$ para este passo, o que não nos permitiria obter a complexidade de tempo de $O(n \log n)$ para todo o algoritmo.

O trabalho deste passo é encontrar duas retas tangentes a $\text{conv}(S_1)$ e $\text{conv}(S_2)$: uma tangenciando os fechos convexos 'por baixo' e a outra tangenciando 'por cima'. A partir destas tangentes é fácil construirmos $\text{conv}(S_1 \cup S_2)$ em tempo linear. Discutiremos como encontrar a tangente inferior; a tangente superior pode ser encontrada de maneira análoga.

Denotaremos por u_i e v_j os vértices de $\text{conv}(S_1)$ e $\text{conv}(S_2)$, respectivamente. Seja $T = u_i v_j$ o segmento de reta que determina a tangente inferior. O problema é que não sabemos de antemão quais são os vértices u_i de $\text{conv}(S_1)$ e v_j de $\text{conv}(S_2)$ que são extremidades de T. (Observe que fizemos uma tarefa semelhante no algoritmo incremental visto na seção anterior, mas, daquela feita a tarefa era mais simples pois conheciamos umas das extremidades de T.) Uma busca ingênua pelas extremidades de T resultaria em complexidade quadrática para este passo.

A busca pelo segmento tangente T feita por Preparata e Hong [20] começa inicializando T como sendo o segmento formado pelo vértice v_i mais à direita de $\text{conv}(S_1)$ e pelo vértice v_j mais à esquerda de $\text{conv}(S_2)$, e 'deixar T cair', primeiro 'escorregando' de um lado e depois do outro, e assim sucessivamente até que as extremidades do segmento procurado sejam encontradas. (Veja Figura 13.)

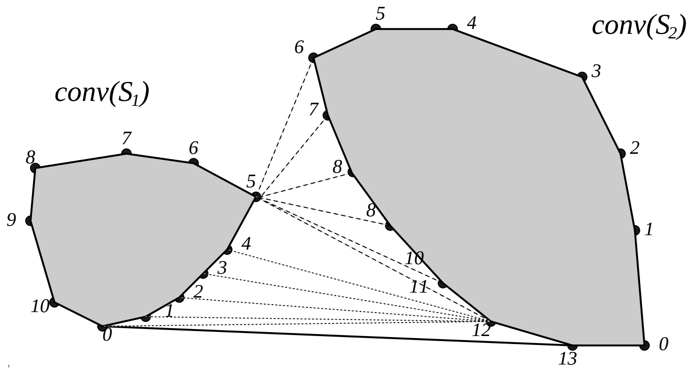


FIGURA 13. Procura pela tangente inferior: no início $T = (v_i, u_j) = (v_5, u_6)$ e a tangente procurada é (v_0, u_{13}).

Algoritmo `Tangente-Inferior`

`Entrada:` Dois fechos convexos $\text{conv}(S_1)$ e $\text{conv}(S_2)$, separados por uma reta vértical.

`Saída:` O segmento 'tangente' inferior a conv(S_1) e conv(S_2).
1. $v_i \leftarrow$ vértice mais à direita de conv(S_1).
2. $u_j \leftarrow$ vértice mais à esquerda de conv(S_2).
3. **while** $T = v_i v_j$ não é a tangente inferior de conv(S_1) e conv(S_2) **do**
4.   **while** T não é tangente inferior de conv(S_1) **do**
5. $j \leftarrow j + 1$
6.   **while** T não é tangente inferior de conv(S_2) **do**
7. $i \leftarrow i - 1$
8. **return** T.

No algoritmo acima, a frase '$T = v_i u_j$ não é tangente inferior de conv(S_1)' é equivalente a dizermos: v_{i-1} está à direita de (v_i, u_j). Analogamente, a frase '$T = v_i u_j$ não é tangente inferior de conv(S_2)' é equivalente a dizermos: u_{i+1} está à direita de (v_i, u_j). Da maneira semelhante ao que ocorreu com os algoritmos anteriores temos que a primitiva usada pelo Algoritmo `Tangente-Inferior` é `Lefton`.

Na Figura 13, inicialmente $T = v_5 u_6$ e o **while** da Linha 4 do algoritmo é executado até o momento em que u_{13} está à esquerda de $T = v_5 u_{12}$. Analogamente, o **while** da Linha 6 é executado até que v_{10} está à esquerda de $T = v_0 u_{12}$. Na próxima iteração do **while** da Linha 3 o segmento tangente $v_0 u_{13}$ é encontrado.

10.6. **Corretude do algoritmo.** Quando o **while** da Linha 3 do algoritmo pára temos que (a reta determinada por) T é tangente a ambos conv(S_1) e conv(S_2). Entretanto, precisamos verificar que:

(1) O **while** da Linha 3 termina: a princípio poderia ocorrer que ao atingirmos a condição de tangência da Linha 6 tivessemos destruído a condição de tangência alcançada pela Linha 4.
(2) Existem quatro tangentes a conv(S_1) e conv(S_2) (veja Figura 14), e o algoritmo deveria encontrar somente uma delas: a tangente inferior de ambos conv(S_1) e conv(S_2).

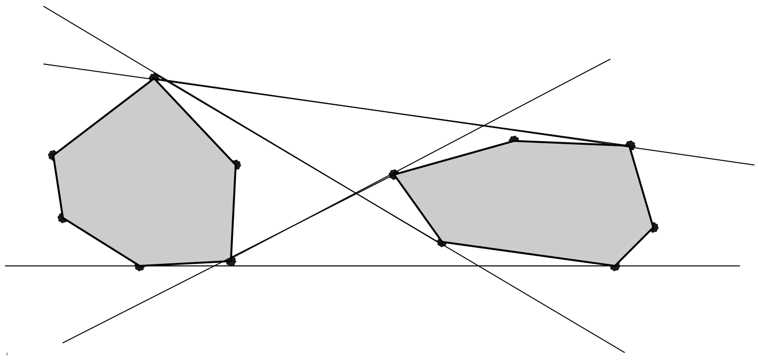


FIGURA 14. As quatro tangentes a conv(S_1) e conv(S_2).

O lemas abaixo serão usados para mostrar que os comandos das Linhas 5 e 7 do algoritmo não podem ser executados infinitamente, o que implica que o algoritmo sempre pára.

Lema 5. *Seja $T = vu$ a tangente inferior de conv(S_1) e conv(S_2). Então v (u, respectivamente) pertence à cadeia fechada que vai, no sentido anti-horário, do vértice mais à esquerda até o vértice mais à direita de conv(S_1) (conv(S_2), respectivamente).*

Prova. Seja L uma reta vertical separando $\text{conv}(S_1)$ e $\text{conv}(S_2)$. Se $\text{conv}(S_2)$ está muito 'acima' de $\text{conv}(S_1)$, então T se aproxima de ser paralelo com L e v_i se aproxima do vértice mais à direita de $\text{conv}(S_1)$. Analogamente, se $\text{conv}(S_2)$ está muito 'abaixo' de $\text{conv}(S_1)$, então v_i se aproxima do vértice mais à esquerda de $\text{conv}(S_1)$. O vértice v está entre estes extremos na cadeia inferior de $\text{conv}(S_1)$. ∎

Lema 6. *(Invariante) Durante toda a execução do Algoritmo `Tangente-Inferior` temos que T tem intersecção vazia com o interior de conv(S_1) e conv(S_2) (i.e. $T \cap \text{conv}(S_i) \subseteq \partial(\text{conv}(S_i))$ para $i = 1, 2$).*

Prova. A prova é por indução no número de alterações de T feitas pelo Algoritmo `Tangente-Inferior`. A afirmação do Lema é verdadeira no começo do algoritmo.

Suponha que o Lema é verdade após alguns passos e que o algoritmo está prestes a decrementar a variável i; isto só é feito se T não é tangente inferior de $\text{conv}(S_1)$. A nova tangente $T' = v_{i-1}u_j$ intersecta o interior de $\text{conv}(S_1)$ se u_j está à esquerda de (v_{i-1}, v_i). (Veja Figura 15.) Mas u_j não poderia também estar à esquerda de (v_i, v_{i+1}) pois senão T estaria intersectando o interior de $\text{conv}(S_1)$. Logo, se o próxima atualização causasse a intersecção de T com o interior de $\text{conv}(S_1)$ teriamos então que (a reta definida por) T seria tangente a $\text{conv}(S_1)$ em v_i e portanto o algoritmo nunca teria incrementado a variável i. ∎

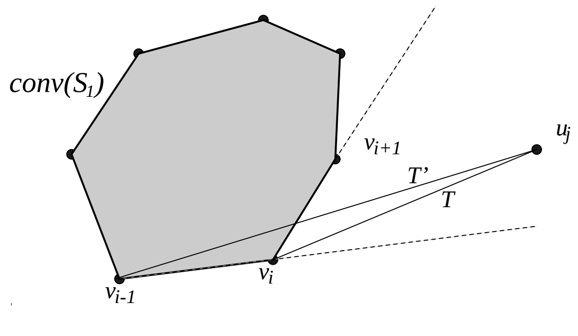


FIGURA 15. O vértice u_j de $\text{conv}(S_2)$ está à esquerda de (v_{i-1}, v_i).

Como T não intersecta o interior de $\text{conv}(S_1)$, e u_j está à direita da reta vertical L que separa $\text{conv}(S_1)$ e $\text{conv}(S_2)$, temos que a variavél i (Linha 7) não pode ser decrementada indefinidamente pois v_i não pode passar (andando no sentido horário) do vértices mais à esquerda de $\text{conv}(S_1)$. De maneira análoga, a variavél j (Linha 5) não pode ser incrementada indefinidamente. Portanto, ambos os comandos **while** das Linha 4 e 6 devem parar e também o comando **while** da Linha 3 (pois em cada iteração do laço das Linhas 3–7 ou a variável i e decrementada ou a variável j e incrementada). Como o algoritmo pára então, pela condição de tangência (Linha 3) o algoritmo deve devolver a tangente inferior.

10.7. **Análise do algoritmo.** A complexidade de tempo do pré-processamento (ordenação dos pontos) que é $O(n \log n)$. No passo `Combinar` os laços são executados $O(n)$ vezes. Se usarmos uma lista circular duplamente ligada para representar os fechos convexos então, dadas as tangentes inferior e superior o novo fecho convexo pode ser construído em tempo $O(n)$. Portanto, a complexidade total do passo `Combinar` é $O(n)$. Se $T(n)$ é a função complexidade de tempo do Algoritmo `MergeHull`, sem contar o custo do pré-processamento, temos que

$$T(n) \leq T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + cn = O(n \log n),$$

onde c é uma constante. Logo, a complexidade de tempo do Algoritmo `MergeHull` é $O(n \log n)$.

11. Cota inferior

Vimos dois algoritmos que encontram o fecho convexo de um conjunto de pontos em tempo $O(n \log n)$. Uma pergunta natural agora seria: Será que se pode fazer melhor do que isto? Poderia ser o caso (como ocorreu com o Problema de Triangularização de Polígonos) que exista um algoritmo, digamos, de complexidade de tempo $O(n)$ para o problema, mas este algoritmo está ainda aguardando para ser descoberto. Nesta seção nos mostraremos (ou argumentaremos) que um tal algoritmo não existe e que $O(n \log n)$ é o melhor que alguém pode esperar conseguir, i.e. $\Omega(n \log n)$ é cota inferior para o Problema do Fecho Convexo.

Observação. O material desta seção foi extraído do Capítulo 2 de Corme, Leiserson e Rivest [4], Capítulo 1, 3 e 5 de Preparata e Shamos [21], Capítulo 1 e 4 de Resende e Stolfi [6], e Capítulo 3 de O'Rourke [19].

11.1. **O que pretendemos provar.** Recordemos que $O(f(n))$ é o conjunto de funções

$$\begin{aligned} O(f(n)) = \{g(n) \quad | \quad & \text{existem contantes positivas } c \text{ e } n_0 \text{ tal que} \\ & 0 \leq g(n) \leq cf(n) \text{ para } n \geq n_0\}, \end{aligned}$$

ou seja, $g(n)$ é limitado por $cf(n)$ para valores de n suficientemente grandes.

O conceito de límite inferior ou cota inferior (*lower bound*) é semelhante,

$$\begin{aligned} \Omega(f(n)) = \{g(n) \quad | \quad & \text{existem contantes positivas } c \text{ e } n_0 \text{ tal que} \\ & 0 \leq cf(n) \leq g(n) \text{ para } n \geq n_0.\} \end{aligned}$$

O que queremos mostrar nesta seção é que a função complexidade de tempo de qualquer algoritmo que resolva o Problema do Fecho Convexo é $\Omega(n \log n)$, ou seja, qualquer algoritmo que contrói o fecho convexo de um dado conjunto de n pontos faz $\Omega(n \log n)$.[2]

[2]Aqui algum comentário extra é devido. Como estamos falando em operações elementares então o *modelo de computação* deve neste momento estar bem definido. O que nos faremos nesta seção é 'passar por cima' de alguns detalhes técnicos que são muito importantes.

11.2. **Redução entre problemas.** Em geral é um problema difícil encontrar cotas inferiores para problemas. A dificuldade é que cotas inferiores devem valer para os algoritmos que já foram escritos e para qualquer outro que possa ou não algum dia ser descoberto. Ou seja, a cota inferior é para *todos* os algoritmos que resolvem o problema, e portanto é uma complexidade intrínsica do problema. No entanto, foram encontradas cotas inferiores para alguns problemas importantes, como por exemplo ordenação. Isto faz com que o trabalho de muitas pessoas fique mais fácil, pois agora cotas inferiores podem ser obtidas via "redução entre problemas".

Suponha que seja sabido que um Problema A tem uma cota inferior $\Omega(n^4)$. Agora suponha que o Problema A pode ser *reduzido* para o Problema B no sentido em que um algoritmo que resolve o Problema B pode ser usado para resolver o Problema A (com um 'pequeno' esforço adicional). Logo, se existe um algoritmo que resolva B rapidamente então este algoritmo pode ser usado para resolver o Problema A 'rapidamente'. Isto frequentemente implica uma cota inferior para o problema B. De fato, suponha que o esforço adicional de resolver o Problema A fazendo uma única chamada a um algoritmo que resolva o Problema B é $O(n)$ (ou seja o algoritmo para resolver A usa um algoritmo que resolve B como subrotina). Podemos portanto concluir que, como $\Omega(n^4)$ é cota inferior de A, então $\Omega(n^3)$ é cota inferior para o Problema B. Esta será a maneira que usaremos para estabelecer uma cota inferior para o Problema do Fecho Convexo.

11.3. **A redução.** Sabe-se que o Problema de Ordenação tem cota inferior $\Omega(n \log n)$.[3] Reduziremos o Problema de Ordenação (este é o Problema A) ao Problema do Fecho Convexo (este é o Problema B). Esta redução é devida a Shamos [22] e é bem simples. Suponha que nos é dada um lista não-ordenada de números $(x_1, x_2, \ldots, x_n)$, $x_i \geq 0$ $(i = 1, \ldots, n)$; este são os dados do problema A. Suponha agora que nos é dado um Algoritmo `Fecho` que contrói o fecho convexo (o algoritmo dá como resposta um polígono convexo tendo como vértices pontos do conjunto e contendo todos os demais pontos) de n pontos em tempo $T(n)$ (função complexidade de tempo). Usaremos o Algoritmo `Fecho` para ordenarmos a lista dada de números em tempo $T(n) + O(n)$, onde o termo $O(n)$ corresponde ao tempo adicional de transformar uma instância do Problema de Ordenação para uma instância do Problema do Fecho Convexo e transformar uma solução do Problema do Fecho Convexo em uma solução do Problema de Ordenação.

Forme um conjunto de pontos no plano constituído pelos pontos (x_i, x_i^2) $(i = 1, \ldots, n)$ (veja Figura 16). Aplique o Algoritmo `Fecho` para encontrar o fecho convexo destes pontos. Encontre o ponto mais baixo $p := (x_k, x_k^2)$ do fecho convexo contruído, este passo pode se feito em $O(n)$. A ordem em que os vértices do fecho convexo ocorrem ao percorrermos a sua fronteira no sentido anti-horário a partir de p nos fornece uma lista dos números dados em ordem crescente. Concluímos que o Algoritmo `Fecho` pode ser usado em um algoritmo que ordena n números em tempo $O(T(n))$. Logo, $T(n) = \Omega(n \log n)$.

[3] Está afirmação é imprecisa, e nos custaria muito tempo fazê-la precisa.

Teorema 7. *O problema de construir o fecho convexo de um conjunto de n pontos no plano tem, no modelo de árvore de decisão algébrica, cota inferior $\Omega(n \log n)$.* ∎

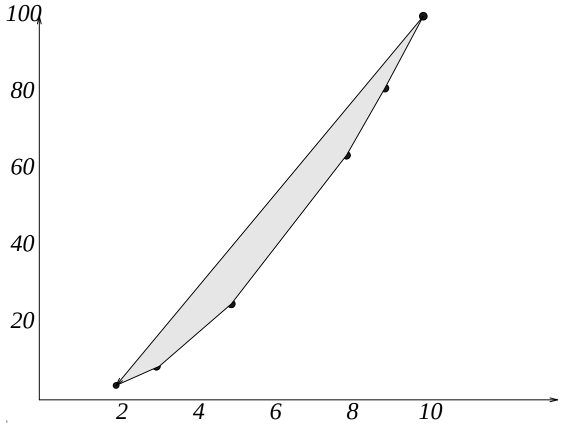


FIGURA 16. Parábola obtida na ordenação dos números $2, 3, 5, 8,$ e 10.

Observe que na redução acima foi fundamental o fato do Algoritmo `Fecho` devolver um polígono convexo como resposta. Não é claro se conseguiriamos fazer a mesma redução se os pontos extremos do fecho convexo fossem dados como resposta. Foi um problema aberto durante muitos anos descobrir se o problema de achar os pontos extremos de um conjunto era de fato mais fácil do que encontrar o fecho convexo desses pontos representado por um polígono. Pesquisadores estabeleceram que $\Omega(n \log n)$ é também uma cota inferior para este outro problema (veja Preparata e Shamos [21], páginas 101–103).

11.4. **Um protótipo computacional.** Como acabamos de ver, ao investigar a complexidade computacional de um problema, i.e. tentar descobrir uma cota inferior não trivial, podemos fazer uso de uma cota não trivial para algum outro problema. Assim, frequentemente reduzimos (ou transformamos) um problema em outro. (Estas reduções entre problemas é algo clássico que é freqüentemente feito em um curso de complexidade de computacional.) Podemos dizer que o Problema de Ordenação é um protótipo computacional, já que este problema é muito usado para derivarmos cotas inferiores de problemas. (Neste mesmo sentido o Problema de Satisfatibilidade é um protótipo computacional na Teoria de NP-completude.)

Um protótipo computacional bastante usado em geometria computacional é o *Problema da Unicidade de Elementos* (*Element Uniqueness*): *dado:* um conjunto com n números reais; *decidir:* se dois desses números são iguais. O Problema da Unicidade de Elementos tem cota inferior $\Omega(n \log n)$ (cf. Dobkin e Lipton [7]). É facil reduzirmos o Problema da Unicidade de Elementos ao Problema do Par Mais Próximo. Isto implica que o algoritmo de divisão-e-conquista que vimos para encontrar um par mais próximo é ótimo (tem complexidade de tempo $O(n \log n)$).

Aos poucos o número de problemas para os quais se conhece uma cota não trivial vai aumentado. Estes novos problemas podem ser usados para encontrarmos cotas inferiores para outros problemas e assim por diante.

12. Uma aplicação: O Problema do Par Mais Distante

Trataremos nesta seção do seguinte *Problema do Par Mais Distante*: *dado:* um conjunto S com n pontos no plano; *encontrar:* uma par de pontos de S cuja distância entre eles seja máxima (máxima em relação as distâncias entre cada par de pontos em S). A distância máxima entre dois pontos em S é chamada de o *diâmetro* de S. Por isto, o problema acima também é chamado de o *Problema do Diâmetro de um Conjunto* (usaremos os dois nomes indiscriminadamente). Este problema pode ser resolvido em tempo $O(n^2)$: basta calcularmos as distâncias entre todos os $n(n-1)/2$ pares de pontos de S. A pergunta natural agora é se este é o melhor que pode ser feito. A resposta é não. Nesta seção veremos um algoritmo que, usando como subrotina um algoritmo para o Problema do Fecho Convexo, resolve o Problema do Par Mais Distante em tempo $O(n \log n)$.

Observação. O material desta seção foi extraído do Capítulo 4 de Preparata e Shamos [21].

12.1. **Cota inferior.** Primeiro vamos obter uma cota inferior não trivial para o problema, para isto procederemos da mesma maneira que fizemos na seção anterior. Um dos problemas para o qual se conhece uma cota inferior não trivial é o *Problema dos Conjuntos Disjuntos*: *dados:* um conjunto $A = \{a_1, a_2, \ldots, a_n\}$ e $B = \{b_1, b_2, \ldots, b_n\}$ de números reais não negativos; *decidir:* se A e B não tem nenhum elemento em comum, i.e. se $A \cap B = \emptyset$. A cota inferior deste problema é $\Omega(n \log n)$ (cf. [1]). Agora reduziremos o Problema dos Conjuntos Disjuntos ao Problema do Diâmetro de um Conjunto. Seja C a circunferência de raio 1 centrada na origem. A cada ponto de a_j de A associaremos o ponto no primeiro quadrante que corresponde à intersecção de C com a reta $y = a_i x$, enquanto cada ponto b_i de B será associado ao ponto no terceiro quadrante que corresponde a intersecção de C com a reta $y = b_i x$. (Veja Figura 17.) Seja S o conjunto desses $2n$ pontos de interseção. Agora temos que o diâmetro de S é igual a 2 se e somente se existem dois pontos diametralmente oposto em S, ou seja, se e somente se $A \cap B \neq \emptyset$. Logo o Problema do Diâmetro de um Conjunto tem também cota inferior $\Omega(n \log n)$.

12.2. **A idéia.** O que pode evitar com que um algoritmo examine cada par de vértices é notar que se (u, v) é um par de pontos mais distantes de um dado conjunto S então u e v são vértices do fecho convexo de S. De fato, seja $v \in S$ tal que (u, v) um par de pontos mais distantes de S e suponha que (digamos) v não é um vértice de $\operatorname{conv}(S)$. Seja w a intersecção da extensão do segmento uv na direção do vetor (u, v) com a fronteira do fecho convexo (é possível que $w = v$). (Veja Figura 18.) Seja ab a aresta extrema do fecho convexo que contém w então ou a distância entre u e a é maior que a distância entre u e v ou a distância entre u e b é maior.

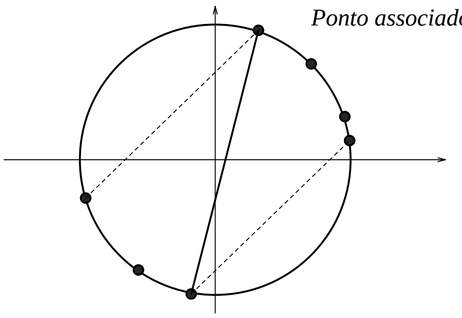


FIGURA 17. Redução do Problema dos Conjuntos Disjuntos ao Problema do Diâmetro de um Conjunto.

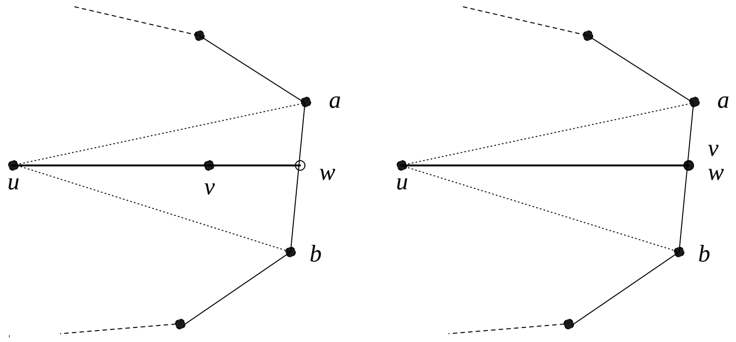


FIGURA 18. Se v não é vértices do fecho convexo então v não pertence a nenhum par mais distante.

Agora, se todos os pontos do conjunto dado forem vértices extremos do fecho convexo não estaremos eliminando nenhum ponto ao construir o fecho convexo. Por outro lado, um algoritmo que constrói o fecho convexo de um conjunto de pontos não devolve somente os vértices extremos. Podemos assumir que o algoritmo devolve o polígono convexo que representa o fecho convexo do conjunto (recordemos que isto significa que além dos pontos extremos uma representação do polígono também nos fornece uma ordem entre os seus vértices). Como veremos, a ordem entre os vértices do polígono nos será de muita valia para resolvermos o Problema Diâmetro de um Conjunto em tempo $O(n \log n)$. Portanto, passaremos agora a tratar do problema de encontrar o diâmetro (um par de vértices mais distantes) de um polígono convexo com n vértices.

12.3. **Diâmetro de um polígono convexo.** Notemos que o diâmetro de um polígono convexo é a maior distância entre duas retas paralelas que são tangentes ao polígono em vértices distintos. (Veja Figura 19.) Com efeito, seja (u, v) um par mais distante de pontos de um dado conjunto S. Seja ainda L_u a reta ortogonal ao segmento uv passando por u e L_v a reta ortogonal ao segmento uv passando por v. A distância entre as retas L_u e L_v é exatamente a distância entre u e v. Afirmamos que L_u e L_v são retas tangentes ao fecho convexo de S. De fato, se (digamos) L_v não é tangente a

$\operatorname{conv}(S)$ então existe um vértice w de $\operatorname{conv}(S)$ no semiplano determinado por L_v que não contém u e que é mais distante de u do que v. (Veja Figura 20.)

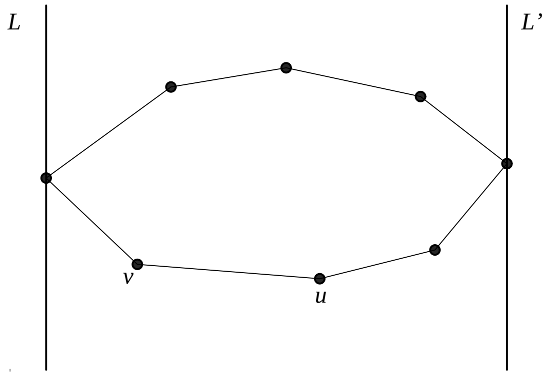


Figura 19. Pontos mais distantes.

Note que na Figura 19 alguns pares pontos não admitem retas de suporte (i.e. retas tangentes) paralelas, veja por exemplo os vértices v e u, isto significa que o comprimento de uv não pode ser o diâmetro do polígono. Um par de pontos que admitem retas de suporte paralelas são ditos *antípodas*. Logo, tendo em vista a nossa primeira observação, precisamos agora examinar pares de vértices do polígono que sejam antípodas. O problema de encontrar o diâmetro de um polígono convexo se reduz portanto ao problema encontrar todos esses pares antípodas do polígono (e torcer para que eles não sejam muitos, e.g. $\Omega(n^2)$ pares) e entre estes escolher um cuja distância entres seus pontos seja máxima.

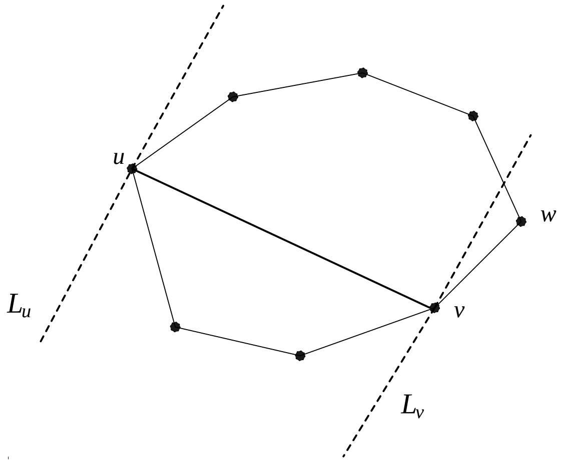


Figura 20. Retas perpendiculares ao segmento uv passando por u e v.

12.4. **Pares antípodas.** Considere um vértice v_i do polígono convexo P. Seja r_i o vértice mais distante do segmento $v_{i-1}v_i$: em caso de empate tome

r_i como sendo o vértice mais distante que foi encontrado primeiro ao percorrermos o polígono no sentido anti-horário a partir de v_i. Analogamente, seja l_i o vértice mais distante de v_iv_{i+1} que foi encontrado primeiro ao percorrermos o polígono no sentido anti-horário a partir de v_i. Seja ainda $A(v_i)$ o conjunto formado por todos os vértices de P que estão na cadeia entre r_i e l_i (inclusive) que não contém v_i. Afirmamos que $A(v_i)$ é precisamente o conjunto dos vértices de P que são antípodas com v_i. (Veja Figura 21.)

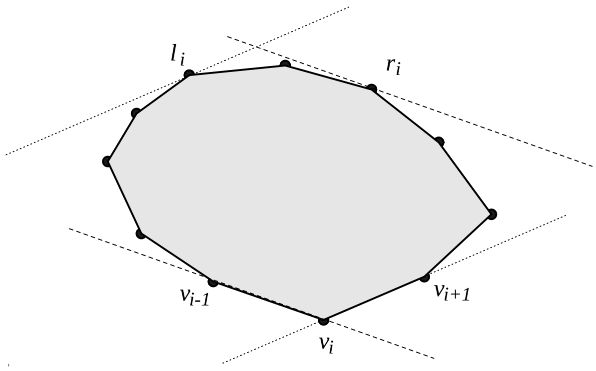


FIGURA 21. Vértices que são antípodas com v_i.

Segue da definição de par antípodal que o par de vértices (v_i, v_j) é antipodal se e somente se existe uma reta na intersecção do ângulo externo θ_i em v_i e o ângulo externo θ_j em v_j. (Veja Figura 22.) Como os vértices de $A(v_i)$ formam uma cadeia temos que todos eles e somente eles satisfazem a propriedade acima.

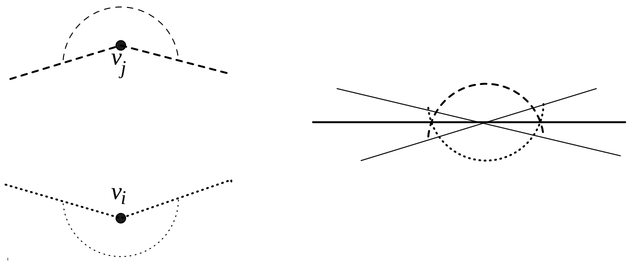


FIGURA 22. Caracterização de pares antípodas.

Observemos que as cadeias formadas por $A(v_i)$ e $A(v_{i+1})$ têm sempre o vértice l_i em comum. Além disso estas duas cadeias têm no máximo 2 vértices em comum (um além do vértice l_i). Isto ocorre quando $r_{i+1} \neq l_i$ (e portanto a aresta v_iv_{i+1} é paralela à aresta $r_{i+1}l_i$). (Veja Figura 23.)

Esta propriedade nos fornece uma algoritmo para encontrarmos os vértices de P que são antípodas com um determinado vértice. A primitiva usada pelo algoritmo é `Area2` (se v_j é um vértice v_j de P mais distante de v_iv_{i+1} então a área do triângulo $\triangle(v_i, v_{i+1}, v_j)$ é máxima).

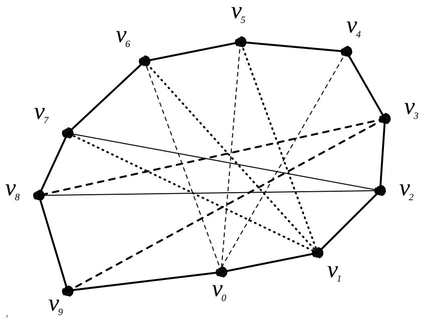


Figura 23. Um polígono convexo e seus pares antípodas representados na figura por diagonais tracejadas.

12.5. **Algoritmo Pares-Antípodas.** Abaixo encontra-se uma descrição do algoritmo que devolve todos os pares antípodas de um polígono.

Algoritmo `Pares-Antípodas`(P)
`Entrada:` Um polígono $P = (v_0, v_1, \ldots, v_{n-1})$.
`Saída:` Todos os pares de vértices antípodas de P.
1. $i \leftarrow n-1$;
2. $j \leftarrow 0$;
3. **while** v_j não é um vértice mais distante de $v_i v_{i+1}$ **do**
4. $\quad j \leftarrow j+1$;
5. $j_0 \leftarrow j$;
6. **while** $j \neq 0$ **do**
7. $\quad i \leftarrow i+1$;
8. $\quad$ **print** (v_i, v_j);
9. $\quad$ **while** v_j não é um vértice mais distante de $v_i v_{i+1}$ **do**
10. $\quad\quad j \leftarrow j+1$;
11. $\quad\quad$ **if** $(v_i, v_j) \neq (v_0, v_{j_0})$
12. $\quad\quad\quad$ **then print** (v_i, v_j);
13. $\quad$ **if** a distância de v_{j+1} até $v_i v_{i+1}$ é igual a distância de v_j até $v_i v_{i+1}$ [$v_j v_{j+1}$ e $v_i v_{i+1}$ são paralelos.]
14. $\quad\quad$ **then if** $(v_i, v_j) \neq (v_0, v_{j_0})$
15. $\quad\quad\quad\quad$ **then print** (v_i, v_j);

No algoritmo os testes do tipo 'v_j não é o vértices mais distante de $v_i v_{i+1}$' são equivalentes a '`Area2`$(v_i, v_{i+1}, v_{j+1}) >$ `Area2`(v_i, v_{i+1}, v_j).' A Linhas 3 e 4 do algoritmo encontram o vértice l_0. O algoritmo tem dois apontadores (indices) i e j que se movem ao redor de P no sentido anti-horário. Um par antipodal e gerado cada vez que um desses dois apontadores avança (Linhas 7 e 8 e Linhas 10, 11 e 12) ou cada vez que o algoritmo encontrar um arestas paralelas de P (Linhas 13, 14 e 15). O primeiro par a ser gerado é o par (v_0, v_{j_0}). O último par a ser gerado é o par (v_{n-1}, v_{j_0}) pois $n-1$ é o último valor assumido por j. Logo, cada par antipodal é gerado uma vez, e como i avança de $n-1$ até j_0 e j avança de j_0 até v_{n-1} nos temos um total de n movimentos, i.e. n pares antípodas no caso de não haver arestas paralelas. O número máximo de arestas paralelas é $\lfloor n/2 \rfloor$, logo o número total de pares antípodas gerados pelo algoritmo é no máximo $3n/2$. Com isso temos o seguinte teorema.

Teorema 8. *O diâmetro de um polígono convexo pode ser determinado em tempo $O(n)$, onde n é o número de vértices do polígono.* ∎

12.6. **Diâmetro de um conjunto de pontos.** Podemos encontrar o diâmetro de um conjunto de n pontos no plano em tempo $O(n \log n)$ se combinarmos o Algoritmo `Pares-Antípodas` com um algoritmo que constrói o fecho convexo de um conjunto de pontos em tempo $O(n \log n)$.

Algoritmo `Diâmetro`(S)
`Entrada:` Um conjunto S de n pontos no plano.
`Saída:` Um par de pontos de S cuja distância entre eles e máxima.
1. Construa o polígono P que represeta o fecho convexo do conjunto S.
2. Aplique o Algoritmo `Pares-Antipodas` para encontrar todos os $O(n)$ pares antípodas do polígono P.
3. Encontre entre esses $O(n)$ pares antípodas um para (u, v) cuja distância entre u e v seja máxima.
4. Retorne o par (u, v).

A Linha 1 do algoritmo acima pode ser executada em tempo $O(n \log n)$. Cada uma das Linhas 2 e 3 tem complexidade de tempo linear. Logo, a complexidade de tempo do algoritmo é $O(n \log n)$ e portanto ele é um algoritmo ótimo (como vimos, $\Omega(n \log n)$ é cota inferior para o problema). Note que o algoritmo poderia ser facilmente modificado para devolver todos os pares mais distantes mantendo a mesma complexidade de tempo, já que o número de pares antipodas (candidatos a pares mais distantes) é $O(n)$.

[A Seção 4.3 do Capítulo 4 de Preparata e Shamos [21] contém mais alguns comentários sobre resultados relacionados com o problema de encontrar o diâmetro de um conjunto de pontos. Por exemplo, Grünbaum [12] mostrou que num conjunto de n pontos no espaço tri-dimensional podem haver no máximo $2n - 2$ pares de pontos mais distantes. (Note que este limite é *tight*; considere o tetrahedro.)]

13. Exercícios

1. (Exercício 3.4.1.1 de [19] — `Embrulho-para-Presente`: melhor caso) Encontre o melhor caso para o Algoritm `Embrulho-para-Presente`: conjuntos de n pontos no plano tal que a complexidade de tempo assintótica do algoritmo seja o menor possível em função de n. Qual é esta complexidade de tempo?
2. (Exercício 3.4.1.2 de [19] — `Embrulho-para-Presente`: melhorias) Durante a execução do algoritmo `Embrulho-para-Presente` algumas vezes é possível determinar alguns pontos que não podem ser vértices do fecho convexo e portanto podem ser eliminado *"on the fly"*. Determine algumas regras para identificar tais pontos. Qual é o conjunto de pontos que exige mais trabalho do seu novo algoritmo?
3. (Exercício 3.4.1.4 de [19] — `QuickHull`: pior caso) Construa um conjunto de pontos genérico que força o Algoritmo `QuickHull` a ter

um comportamente quadrático. Por 'genérico' (como sempre) queremos dizer uma construção que seja o pior caso para o `QuickHull` para valores arbitrários de n.

4. (Exercício 3.4.1.5 de [19] — `QuickHull`: análise do pior caso) Argumente que o `QuichHull`, como o Algoritmo `Embrulho-para-Presente`, tem complexidade de tempo *output-sensitive* de $O(nh)$, onde n é o número de pontos e h é o número de pontos na fronteira do fecho convexo.
5. (Exercício 3.4.1.7 de [19] — `QuickHull`: complexidade no caso médio) Argumente que o `QuichHull` tem complexidade de tempo $O(n)$ para n pontos escolhidos uniformemente ao acaso no quadrado de dimensão 1.
6. (Exercício 3.8.5.4 de [19] — `MergeHull`: *merge* sem ordenação). Se o pré-porcessamento do Algoritmo `MergeHull` não é feito, os fechos convexos podem se intersectar. Descreva um algoritmo que faz o 'merge' de dois fechos convexos arbitrários com k e m vértices em tempo $O(k + m)$. Este seu algoritmo fornece um outro algoritmos por divisão-e-conquista de complexidade de tempo $O(n \log n)$ para construir o fecho convexo de um conjunto de n pontos.
7. (Exercício 3.5.6.1 de [19] — `Graham-Scan`: pior caso). Construa um conjunto de pontos para o qual o laço **while** da Linha 6 do algoritmo de Graham faz o maior número de iterações.
8. (Polígono passando por conjunto de pontos) Descreva um algoritmo de complexidade de tempo $O(n \log n)$ que dado um conjunto S de n pontos no plano contrói um polígono que tem os pontos em S como vértices.
9. (Algoritmo `Incremental` ótimo) Modifique o Algoritmo `Incremental` visto em sala de aula de tal forma que a sua complexidade de tempo seja reduzida para $O(n \log n)$.
   [**Sugestão.** Em um pré-processamento ordene os pontos de S em relação a x-coordenada de tal forma que $p \notin Q$ em cada passo. Examine os pontos nesta ordem.]
10. (Exercício 3.9.1.1 de [19] — fecho convexo de polígonos monótonos) Descreva um algoritmo que controi o fecho convexo de um polígono monótono em tempo linear.
11. (Exercício 3.9.1.2 de [19] — fecho convexo de um polígono arbitrário) Descreva um algoritmo que controi o fecho convexo de um polígono arbitrário em tempo linear. este execício é *tricky* e vários algoritmos publicados para este problema estavam errados.
    [**Observação.** A cota inferio de $\Omega(n \log n)$ é para um conjunto de pontos sem estrutura alguma, não para os pontos que são os vértices de um polígono. Os vértices de um polígono são dados na ordem em que eles ocorem ao percirrermos a fronteira do polígono.]
12. (Vértice do fecho convexo) Dado um conjunto S com n pontos de $\mathbb{R}^2$ descreva um algoritmo linear que determina se um ponto $p \in S$ é um vértice do fecho convexo de S. Mostre a corretude e analise complexidade do seu algoritmo.

13. (Camadas convexas) Dado um conjunto S de n pontos no plano nós definimos a *camadas convexas* de S indutivamente. A primeira camada convexa de S consiste da fronteira de $\mathrm{conv}(S)$ (ou seja, a linha poligonal (ou cadeia) fechada que tem como vértices os vertices de $\mathrm{conv}(S)$). Para $i > 1$, defina S_i como sendo os pontos de S que restaram após removermos as camadas $i = 1, 2, \ldots, i-1$. Então a i-ésima camada convexa de S é a fronteira de $\mathrm{conv}(S_i)$ se $S_i \neq \emptyset$. (Veja Figura 24.)

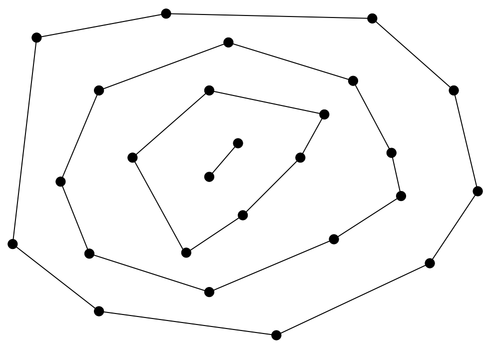


FIGURA 24. Camadas convexas de um conjunto de pontos.

Descreva um algoritmo de complexidade de tempo $O(n^2)$ que constrói as camadas convexas de um dado conjunto S de n pontos. Mostre a corretude e analise a complexidade do seu algoritmo.

(Conjunto de pontos $\Rightarrow$ polígono) Dado um conjunto S de n pontos no plano descreva um algoritmo de complexidade de tempo $O(n \log n)$ que constrói um polígono (simples) que tem como vértices os pontos de S. Mostre a corretude e analise a complexidade do seu algoritmo.

(Fecho convexo de intersecção de retas) Dado um conjunto H de n retas no plano tal que duas-a-duas as retas são não-paralelas descreva um algoritmo de complexidade de tempo $O(n \log n)$ que constrói o fecho convexo dos pontos de intersecção entre as retas. Mostre a corretude e analise a complexidade do seu algoritmo.

Referências

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

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

[3] D.R. Chand e S.S. Kapur, An algorithm for convex polytopes, *JACM* **17** (1970), no. 1, 78–86.

[4] T.H. Cormen, C.E. Leiserson e R.L. Rivest, *Introduction to algorithms*, The MIT Electrical Engineering and Computer Scienece Series, The MIT Press, MacGraw-Hill Book Company, 1990, QA758 C811i.

[5] M. de Berg, M. van Kreveld, M. Overmars e O. Schwarzkopf, *Computational geometry: Algorithms and applications*, Springer, 1997, Second Edition, 2000.

[6] P.J. de Resende e J. Stolfi, *Fundamentos de geometria computacional*, IX Escola de Computação, 1994.

[7] D. Dobkin e R. Lipton, On the complexity os computations under varying set of primitives, *Journal of Computer and System Sciences* **18** (1979), 86–91.

[8] W. Eddy, A new convex hull algorithm for planar sets, *ACM Trans. Math. Software* **3** (1977), no. 4, 398–403.

[9] L.H. Figueiredo e P.C.P. Carvalho, *Introdução à geometria computacional*, 18º Colóquio Brasileiro de Matemática, IMPA, 1991, QA758 F475i.

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

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

[12] B. Grübaum, A proof of Vazsonyi's conjecture, *Bull. Res. Council Israel* **6** (1956), 77–78.

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

[14] S. Khuller, *Cmsc 754 computational geometry*, Course Syllabu, Spring 1996.

[15] M.J. Laszlo, *Computational geometry and computer graphics in C++*, Prentice Hall, Upper Saddle River, NJ, 1996.

[16] Rajeev Motwani e Prabhakar Raghavan, *Randomized algorithms*, Cambridge University Press, New York, 1995.

[17] K. Mulmuley, A fast planar partition algorithm, I, *Proceedings of the 29th Annual IEEE Symposium on Fooundations of Computer Science*, 1988, pp. 580–589.

[18] K. Mulmuley, *Computational geometry: An introduction through randomized algorithms*, Prentice Hall, Englewood Cliffs, NJ, 1994.

[19] J. O'Rourke, *Computational geometry in C*, Cambridge University Press, Cambridge, 1993, Second Edition, 1998.

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

[21] F.P. Preparata e M.I. Shamos, *Computational geometry: An introduction*, Texts and Monographs in Computer Science, Springer-Verlag, New York, 1985, QA758 P927c.

[22] M.I. Shamos, *Computacional geometry*, Ph.D. thesis, Yale University, New Haven, 1978.

[23] G.T. Toussaint, A historical note on convex hull finding algorithms, *Patt. Recogn. Lett.* **3** (1985), 21–28.