

# Capítulo 1

## Geometria Computacional

---

Capítulo de:

C.M.H. de Figueiredo, M.J.M.S. Lemos, V.G.P. de Sá e G.D. da Fonseca. Introdução aos Algoritmos Randomizados. *26º Colóquio Brasileiro de Matemática, IMPA, Rio de Janeiro, 2007.*

---

Geometria computacional é a área da computação que estuda a complexidade de algoritmos, problemas e estruturas de dados de natureza geométrica. O problema mais famoso é talvez o fecho convexo, que consiste em determinar o menor polígono convexo que contém um dado conjunto de pontos.

Diversas razões justificam o fato de os algoritmos randomizados terem ganho importância central no estudo da geometria computacional. Primeiro, algoritmos randomizados são, como já observamos nos capítulos anteriores, freqüentemente mais simples e eficientes na prática do que seus equivalentes determinísticos. Segundo, algoritmos randomizados para problemas no plano geralmente se estendem para espaços  $d$ -dimensionais, com pequenas modificações. Finalmente, os algoritmos determinísticos mais eficientes para vários problemas foram obtidos a partir das de-randomizações de algoritmos randomizados. Por exemplo, o único algoritmo ótimo conhecido para computação do fecho convexo em dimensões ímpares maiores que 3 é a de-randomização de um algoritmo randomizado incremental.

Devido à natureza contínua dos problemas, o modelo computacional mais utilizado é o *real RAM*, onde operações algébricas com números reais podem ser realizadas em tempo constante. Os algoritmos são normalmente descritos de modo geométrico, sem entrar em detalhes de implementação de funções como cálculo de ângulos e distâncias. Assume-se que qualquer computação envolvendo um número constante de primitivas geométricas pode ser realizada em tempo  $O(1)$ .

Também assume-se que a dimensão  $d$  do espaço Euclidiano é uma constante. Portanto, um algoritmo cuja complexidade seja  $O(2^d n)$  tem sua complexidade escrita como  $O(n)$ . Dependências exponenciais na dimensão do espaço são comuns. Sendo assim, a maior parte dos algoritmos não é eficiente para dimensões muito altas.

Outra prática comum em geometria computacional é assumir que a entrada do problema

está em *posição geral*. Não tentamos definir formalmente posição geral, mas a idéia é desconsiderar propriedades que se perdem com uma perturbação infinitesimal da entrada. Por exemplo, caso a entrada seja um conjunto de pontos, podemos assumir que não há três pontos colineares, ou que não há quatro pontos cocirculares. Caso a entrada seja um conjunto de retas, podemos assumir que não há retas verticais, horizontais, ou duas retas paralelas. Na maioria dos casos, assumir posição geral não altera a complexidade do problema e simplifica a explicação e a análise dos algoritmos.

Na seção 1.1, apresentamos um algoritmo randomizado incremental para o problema de programação linear com um número constante de variáveis. Na seção 1.2, introduzimos o conceito de função hash randomizada, que utilizamos no algoritmo para obtenção do par de pontos mais próximos descrito na seção 1.3. O leitor encontrará, na seção 1.4, diversos exercícios algorítmicos usando as técnicas introduzidas neste capítulo. E, como de costume, incluímos notas bibliográficas e comentários mais avançados na seção 1.5.

## 1.1 Programação linear

Um *problema de programação linear* com  $d$  variáveis consiste em determinar o vetor  $d$ -dimensional  $X$  que maximiza a função linear  $f = CX$  e que satisfaz um sistema de desigualdades lineares  $AX \leq B$ . Reescrevendo o problema sem usar notação matricial temos:

$$\begin{aligned} \text{Maximizar: } & f = c_1x_1 + c_2x_2 + \cdots + c_dx_d \\ \text{Satisfazendo: } & a_{11}x_1 + a_{12}x_2 + \cdots + a_{1d}x_d \leq b_1 \\ & \vdots \\ & a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nd}x_d \leq b_n. \end{aligned}$$

Geometricamente, cada desigualdade linear representa um semi-espaco  $d$ -dimensional. A interseção dos  $n$  semi-espacos é chamada de *região viável*. Maximizar a função linear  $f = CX$  satisfazendo as desigualdades  $AX \leq B$  consiste em determinar um ponto extremo na direção  $C$ , que esteja contido na região viável. Um problema de programação linear com duas variáveis está ilustrado na figura 1.1(a).

Concentramos nossa explicação no caso de apenas duas variáveis ( $d = 2$ ), mencionando brevemente como estender o algoritmo para um número arbitrário de variáveis. Sem perda de generalidade, assumimos que  $C = (1, 0)$ , pois os demais casos podem ser reduzidos ao caso  $C = (1, 0)$  rodando o espaço (figura 1.1(b)). Assim, estamos interessados em obter o ponto mais à direita da interseção de um conjunto de semiplanos  $S$ .

Existem dois casos especiais em que o problema não possui solução. O primeiro caso ocorre quando a região viável é vazia (figura 1.2(a)) e é chamado de *problema inviável*. O segundo caso ocorre quando a região viável não é limitada do lado direito (figura 1.2(b)) e é chamado de *problema ilimitado*. Dizemos que dois semiplanos  $s_1, s_2 \in S$  *limitam o problema* se a região formada por  $s_1 \cap s_2$  é limitada do lado direito.

O caso de o problema ser ilimitado é o primeiro que tratamos. Desejamos obter um algoritmo que ou diga que o problema é ilimitado ou encontre dois semiplanos que limitem

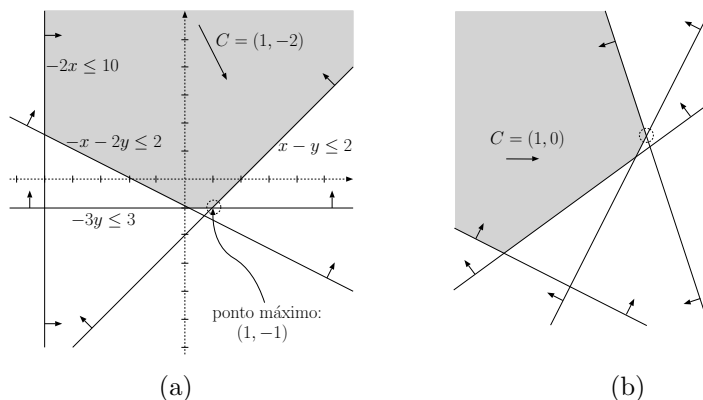


Figura 1.1: (a) Interpretação geométrica de um problema de programação linear com duas variáveis. (b) O mesmo problema após rotação.

o problema. Dado um semiplano  $s$ , definimos  $N(s)$  como o vetor unitário normal à borda de  $s$  e que aponta para o interior de  $s$ . Definimos  $N'(s)$  como o ângulo entre  $N(s)$  e  $(-1, 0)$ , com  $-\pi < N'(s) \leq \pi$ . Seja  $s_1$  o semiplano com  $N'(s) \geq 0$  que minimiza  $N'(s)$  e  $s_2$  o semiplano com  $N'(s) < 0$  que maximiza  $N'(s)$ . É possível provar que  $s_1$  e  $s_2$  limitam o problema se  $N'(s_1) - N'(s_2) < \pi$ . O problema é ilimitado se  $s_1$  ou  $s_2$  não existir, ou se  $N'(s_1) - N'(s_2) > \pi$ . O caso  $N'(s_1) - N'(s_2) = \pi$  é mais delicado, mas não ocorre quando os semiplanos estão em posição geral.

Usando o procedimento descrito no parágrafo anterior, não só podemos nos concentrar apenas em problemas limitados, como também sabemos como obter duas restrições que limitam o problema, caso elas existam. Esta é a condição inicial do nosso algoritmo incremental. A partir daí, acrescentamos as restrições uma a uma, atualizando o ponto extremo satisfazendo as restrições. De modo geral, um *algoritmo randomizado incremental* primeiro resolve o problema para um subconjunto pequeno dos elementos da entrada e, a cada passo, acrescenta um novo elemento da entrada escolhido aleatoriamente, atualizando a solução. Quando todos os elementos da entrada tiverem sido acrescentados, tem-se a solução do problema.

Sejam  $s_1, \dots, s_n$  os  $n$  semiplanos da entrada do problema, onde  $s_1, s_2$  são um par de semiplanos que limitam o problema. Definimos  $S_i = \{s_1, \dots, s_i\}$  e  $p_i$  como o ponto mais à direita da interseção de semiplanos de  $S_i$ . Iniciamos o algoritmo determinando o ponto  $p_2$ . O ponto  $p_2$  pode ser determinado resolvendo o sistema linear formado pelas retas que definem a borda de  $s_1$  e  $s_2$ . Para calcularmos  $p_n$ , que é a solução do nosso problema, o algoritmo procede calculando  $p_i$  incrementalmente, para  $i$  de 3 até  $n$ . Existem dois casos que podem ocorrer: ou bem  $p_{i-1} \in s_i$ , ou  $p_{i-1} \notin s_i$ . Analisamos estes dois casos separadamente.

Note que a região viável definida por  $S_i$  está contida na região viável definida por  $S_{i-1}$ , pois a região viável de  $S_i$  é a interseção de  $s_i$  com a região viável de  $S_{i-1}$ . Conseqüentemente, se  $p_{i-1} \in s_i$ , então  $p_i = p_{i-1}$ . Neste caso,  $p_i$  pode ser computado em tempo  $O(1)$ .

Caso  $p_{i-1} \notin s_i$ , precisamos examinar os semiplanos de  $S_i$  para determinar o valor de  $p_i$ . Fica como exercício provar que, se o problema é viável, então  $p_i$  está na borda de  $s_i$ . No caso

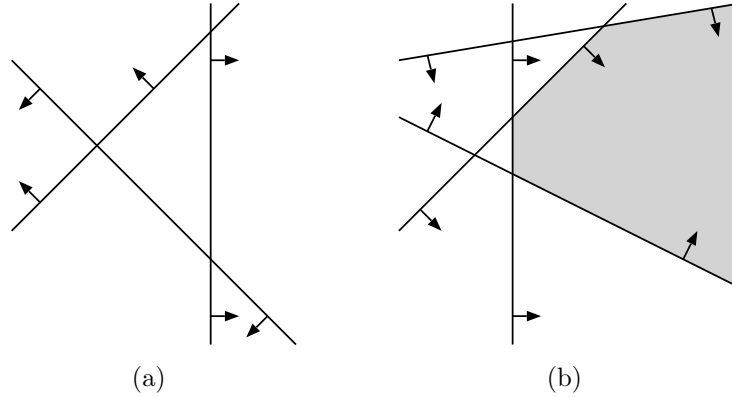


Figura 1.2: (a) Problema de programação linear inviável. (b) Problema de programação linear ilimitado.

de duas variáveis, é fácil determinar o valor de  $p_i$ , em tempo  $O(i)$ , examinando a interseção de cada semiplano de  $S_{i-1}$  com a reta formada pela borda de  $s_i$ . É possível que, neste procedimento, não encontremos nenhum ponto viável. Se isso ocorrer, podemos afirmar que o problema é inviável. Vale notar que, no caso de  $d$  variáveis, como a solução se encontra no subespaço  $s_i$ , é necessário resolver recursivamente um problema de programação linear com  $d - 1$  variáveis.

Assim, para acrescentarmos um semiplano a um problema com  $i$  semiplanos, a complexidade de tempo é  $O(i)$  no pior caso. Para acrescentarmos  $n$  semiplanos, um a um, começando com 2 semiplanos, a complexidade de tempo é

$$T(n) = \sum_{i=3}^n O(i) = O(n^2).$$

Desejamos reduzir o valor esperado da complexidade de tempo. Para isto, permutamos aleatoriamente os semiplanos de  $s_3$  a  $s_n$ . O algoritmo para permutar aleatoriamente um vetor de  $n$  elementos em tempo  $O(n)$  está descrito na figura 1.3. O pseudo-código do algoritmo de programação linear encontra-se na figura 1.4.

Podemos escrever o valor esperado da complexidade de tempo como

$$\mathbf{E}[T(n)] = \sum_{i=3}^n (q_i O(i) + (1 - q_i) O(1)),$$

onde  $q_i$  é a probabilidade de  $p_{i-1} \notin s_i$ . Para determinarmos  $\mathbf{E}[T(n)]$ , precisamos calcular um limite superior para o valor de  $q_i$ . Este limite superior deve depender apenas da permutação aleatória dos semiplanos, e não da entrada do problema.

A técnica *análise de trás para frente* é freqüentemente utilizada para analisar algoritmos incrementais randomizados. O algoritmo incremental descrito parte de uma solução inicial e segue acrescentando um semiplano por vez. Podemos imaginar esse processo de trás para

Entrada:

$v$ : Vetor com  $n$  elementos a serem permutados aleatoriamente.

Saída:

O vetor  $v$  permutado aleatoriamente.

Observações:

$\text{rand}(n)$ : Número aleatório distribuído uniformemente de 0 a  $n - 1$ .

permutação Aleatória( $v, n$ ):

para  $i$  decrescendo de  $n - 1$  até 1

troca  $v[i]$  com  $v[\text{rand}(i)]$

---

Figura 1.3: Algoritmo que permuta aleatoriamente um vetor.

frente, partindo da solução do problema e removendo um semiplano por vez, até chegar na solução inicial. A cada passo, removemos um semiplano aleatório dentre  $i - 2$  semiplanos, pois 2 semiplanos fazem parte da solução inicial. O valor de  $q_i$  é a probabilidade de removermos um dos 2 semiplanos que definem  $p_i$ . Então concluímos que  $q_i \leq 2/(i - 2) = O(1/i)$ . Assim, temos

$$\mathbf{E}[T(n)] = \sum_{i=3}^n (O(1/i)O(i) + O(1)O(1)) = \sum_{i=3}^n O(1) = O(n).$$

## 1.2 Funções hash

Funções *hash*, também chamadas de funções *de dispersão*, não são um tópico particularmente geométrico. Entretanto, elas encontram diversas aplicações dentro de geometria computacional, assim como nas áreas mais diversas da computação, de linguagens de programação a teoria da complexidade. Dados um parâmetro inteiro  $m$  e um conjunto de números inteiros  $C$ , uma *função hash* é uma função  $h$  que mapeia os elementos de  $C$  em números inteiros de 0 a  $m - 1$ . Chamamos os elementos de  $C$  de *chaves*. Desejamos escolher aleatoriamente uma função hash de modo que, se  $x$  e  $y$  são duas chaves distintas, então  $h(x) \neq h(y)$  com alta probabilidade (no caso, pelo menos  $1 - 1/m$ ).

Por exemplo, considere o conjunto de chaves  $C = \{7, 92, 1092\}$  e o parâmetro  $m = 3$ . Uma excelente função hash seria  $h(x) = x \bmod 3$ , pois  $h(7) = 1$ ,  $h(92) = 2$ ,  $h(1092) = 0$ , de modo que  $h(x) \neq h(y)$  sempre que  $x \neq y$ . Porém, a função  $h(x) = x \bmod 3$  funciona extremamente mal para certos conjuntos de chaves, como por exemplo,  $C = \{3, 33, 129\}$ . Neste caso, temos  $h(x) = 0$  para todo  $x \in C$ . A utilização de números aleatórios permite construir funções hash que funcionem bem, no caso esperado, para qualquer conjunto de chaves  $C$ , sem nem mesmo necessitarmos examinar o conjunto  $C$ .

Para construirmos uma função hash, primeiro determinamos um número primo  $p$  tal que  $p > x$  para todo  $x \in C$ . A obtenção de tal número primo não é tarefa trivial, porém  $p$  pode ser obtido eficientemente usando as técnicas descritas no capítulo ??, juntamente com

Entrada:

$S$ : Conjunto de  $n$  semiplanos.

Saída:

$p$ : Ponto extremo direito na interseção dos semiplanos de  $S$ .

progLin( $S$ ):

determinar 2 semiplanos  $s_1, s_2$  que limitem o problema

se  $s_1, s_2$  não existem:

retorne "problema ilimitado"

$p \leftarrow$  interseção das bordas de  $s_1$  e  $s_2$

fazer  $s_3, \dots, s_n$  uma permutação aleatória de  $S \setminus \{s_1, s_2\}$ .

para  $i$  de 3 até  $n$ :

se  $p \notin s_i$ :

$p \leftarrow$  ponto extremo direito na borda de  $s_i$  e

contido na interseção de  $s_1, \dots, s_{i-1}$

se  $p$  não existe:

retorne "problema inviável"

retorne  $p$

---

Figura 1.4: Algoritmo que resolve o problema de programação linear.

o postulado de Bertrand, que diz que, para todo  $x > 3$ , existe um número primo  $p$  tal que  $x \leq p \leq 2x - 2$ . Após calcularmos  $p$ , obtemos dois números inteiros aleatórios  $a \in [1, p - 1]$  e  $b \in [0, p - 1]$ . Definimos a função hash como

$$h(x) = \text{res}(\text{res}(ax + b, p), m),$$

onde  $\text{res}(x, y)$  representa, como na seção ??, o resto da divisão de  $x$  por  $y$ . Para essa função hash, temos o seguinte resultado:

**Teorema 1.2.1.** *Dadas duas chaves distintas  $x, y \in C$ , a probabilidade de  $h(x) = h(y)$  é no máximo  $1/m$ . A probabilidade é obtida em função dos valores aleatórios  $a$  e  $b$ , independente do valor de  $x$  e  $y$ .*

*Demonstração.* Nesta demonstração usamos notação e propriedades estabelecidas no capítulo ??. Definimos  $H(x) = \bar{a} \bar{x} + \bar{b}$  em  $\mathbb{Z}_p$ . Considere duas chaves  $\bar{c}_1, \bar{c}_2 \in \mathbb{Z}_p$ . Temos que

$$H(c_1) = \bar{a} \bar{c}_1 + \bar{b} \quad \text{e} \quad H(c_2) = \bar{a} \bar{c}_2 + \bar{b}.$$

Primeiro, mostramos que  $H(c_1) = H(c_2)$  se e só se  $\bar{c}_1 = \bar{c}_2$ . Partindo de  $H(c_1) = H(c_2)$ , por definição temos que  $\bar{a} \bar{c}_1 + \bar{b} = \bar{a} \bar{c}_2 + \bar{b}$ . Pela existência de inverso aditivo em  $\mathbb{Z}_p$ , temos que  $\bar{a} \bar{c}_1 = \bar{a} \bar{c}_2$ . Como  $a \neq 0$  e  $p$  é primo,  $\bar{a}$  possui inverso multiplicativo em  $\mathbb{Z}_p$ . Então concluímos que  $\bar{c}_1 = \bar{c}_2$ .

O próximo passo é mostrarmos que podemos determinar  $\bar{a}$  e  $\bar{b}$  em função de  $\bar{c}_1$ ,  $\bar{c}_2$ ,  $H(c_1)$  e  $H(c_2)$ . Subtraindo  $\bar{a} \bar{c}_1$  de  $H(c_1)$  temos que

$$\bar{b} = H(c_1) - \bar{a} \bar{c}_1.$$

Subtraindo  $H(c_1) - H(c_2)$  temos

$$H(c_1) - H(c_2) = \bar{a} (\overline{c_1 - c_2}).$$

Como  $\bar{c}_1 \neq \bar{c}_2$ , então  $\overline{c_1 - c_2}$  possui inverso multiplicativo em  $\mathbb{Z}_p$ . Denotamos o inverso multiplicativo de  $\overline{c_1 - c_2}$  por  $(\overline{c_1 - c_2})^{-1}$ . Segue que

$$\bar{a} = (\overline{c_1 - c_2})^{-1} (H(c_1) - H(c_2)).$$

Note que existem  $p(p-1)$  atribuições possíveis para o par  $(\bar{a}, \bar{b})$ , e existem também  $p(p-1)$  pares  $(H(c_1), H(c_2))$  com  $H(c_1) \neq H(c_2)$ . Sendo assim, existe uma correspondência 1 para 1 entre  $(\bar{a}, \bar{b})$  e  $(H(c_1), H(c_2))$ . Como  $(\bar{a}, \bar{b})$  é escolhido aleatória e uniformemente em  $(\mathbb{Z}_p^*, \mathbb{Z}_p)$ , o par  $(H(c_1), H(c_2))$  também é escolhido aleatória e uniformemente dentre pares de valores distintos em  $\mathbb{Z}_p$ .

A partir daqui, consideramos  $H(c_1)$  e  $H(c_2)$  como inteiros de 0 a  $p-1$ , e não mais como elementos de  $\mathbb{Z}_p$ . Temos que  $h(c_1) = \text{res}(H(c_1), m)$  e  $h(c_2) = \text{res}(H(c_2), m)$ . Queremos determinar a probabilidade  $\Pr[h(c_1) = h(c_2)]$  e sabemos que  $H(c_1) \neq H(c_2)$ .

Se fixarmos o valor de  $H(c_1)$ , existem no máximo  $\lceil p/m \rceil - 1$  valores possíveis de  $H(c_2)$  com  $H(c_2) \neq H(c_1)$  e  $h(c_1) = h(c_2)$ . Como há no total  $p-1$  valores possíveis para  $H(c_2)$ , então

$$\Pr[h(c_1) = h(c_2)] \leq \frac{\lceil p/m \rceil - 1}{p-1} \leq \frac{(p-1)/m}{p-1} \leq \frac{1}{m}.$$

□

**Corolário 1.2.2.** *Considere uma chave  $x \in C$ , e um conjunto  $C' \subseteq C$  com  $|C'| \leq m$ . O valor esperado do número de chaves  $y \in C'$  com  $h(x) = h(y)$  é menor que 2, ou seja,  $\mathbf{E}[|\{y \in C' : h(x) = h(y)\}|] < 2$ .*

*Demonstração.* O valor esperado do número de chaves  $y \in C'$  com  $h(x) = h(y)$  é a soma das probabilidades de  $h(x) = h(y)$  para as chaves  $y \in C'$ . Caso  $x \in C'$ , usando o teorema 1.2.1,

$$\mathbf{E}[|\{y \in C' : h(x) = h(y)\}|] \leq 1 + \sum_{i=1}^{|C'|-1} \frac{1}{m} = 1 + \frac{|C'| - 1}{m} < 2.$$

Caso  $x \notin C'$ , e usando ainda o teorema 1.2.1,

$$\mathbf{E}[|\{y \in C' : h(x) = h(y)\}|] \leq \sum_{i=1}^{|C'|} \frac{1}{m} = \frac{|C'|}{m} \leq 1.$$

□

### 1.3 Par de pontos mais próximos

Dado um conjunto  $P$  contendo  $n$  pontos de um espaço Euclidiano  $d$ -dimensional, é natural perguntar qual o *par de pontos mais próximos*, isto é, quais são os dois pontos distintos  $p, q \in P$  que minimizam a distância  $|p - q|$ . Um algoritmo trivial para este problema tem complexidade de tempo  $O(n^2)$ , simplesmente calculando as distâncias entre todos os  $n(n - 1)/2$  pares de pontos e escolhendo o par que determina a distância mínima. Entretanto, existem algoritmos mais eficientes. Por bastante tempo, o problema foi considerado completamente solucionado, pois há diversos algoritmos determinísticos com tempo  $O(n \log n)$  e existe um limite inferior de  $\Omega(n \log n)$  para o problema, mesmo no caso unidimensional. O limite inferior se refere apenas a algoritmos determinísticos e restritos a comparar resultados de operações algébricas.

O *piso* de um número real  $x$  é denotado por  $\lfloor x \rfloor$  e é definido como o maior inteiro  $i$  tal que  $i \leq x$ . O piso não é uma operação algébrica. Surpreendentemente, usando randomização e computação de pisos, é possível resolver o problema em tempo  $O(n)$ . Usando pisos, mas sem usar randomização, é possível resolver o problema em tempo  $O(n \log \log n)$ . Descrevemos um algoritmo randomizado que resolve o problema em tempo  $O(n)$ . O algoritmo pode ser facilmente generalizado para espaços  $d$ -dimensionais, mas nos restringimos ao caso bidimensional.

O algoritmo usa randomização de duas maneiras. Primeiro, usa randomização explicitamente na escolha do ponto a ser amostrado a cada iteração. Segundo, usa randomização implicitamente através da utilização de funções hash (vide seção 1.2). Iniciamos com algumas definições que facilitam a descrição do algoritmo. O *vizinho mais próximo* de um ponto  $p$  (com respeito ao conjunto  $P$ ), denotado por  $vmp(p)$ , é o ponto  $p' \in P \setminus \{p\}$  que minimiza  $|p - p'|$ . Definimos  $f(p) = |p - vmp(p)|$ , ou seja,  $f(p)$  é a distância do ponto  $p$  ao seu vizinho mais próximo.

A idéia do algoritmo é, a cada iteração, escolher um ponto  $p$  aleatoriamente e remover de  $P$  todo ponto  $q$  com  $f(q) \geq f(p)$ . Este procedimento é repetido até todos os pontos de  $P$  serem removidos. Note que  $f(p)$  limita superiormente a distância entre o par de pontos mais próximos. Conseqüentemente, os pontos removidos não podem ser um dos pontos do par de pontos mais próximos, a não ser que  $p$  seja um dos pontos do par de pontos mais próximos. Quando todos os pontos foram removidos, sabe-se que  $p$  e seu vizinho mais próximo são de fato o par de pontos mais próximos. O pseudo-código do algoritmo encontra-se na figura 1.5.

Entretanto, calcular  $f(q)$  leva tempo  $O(n)$ , pois determinar o vizinho mais próximo de um ponto  $q$  exige examinar todos os demais pontos de  $P$ . Assim, a primeira chamada da função `removePontos` leva tempo  $\Theta(n^2)$  ao calcular  $f(q)$  para todo  $q \in P$ . Como a nossa meta é determinar o par de pontos mais próximos em tempo  $O(n)$ , precisamos acelerar a função `removePontos`. Temos que remover do conjunto  $P$  todos os pontos  $q$  com  $f(q) \geq f(p)$ , sem calcularmos  $f(q)$  individualmente para cada ponto  $q \in P$ .

Podemos, antes de iniciar o algoritmo, transladar e escalar os pontos sem alterar o par de pontos mais próximos. Portanto, assumimos, sem perda de generalidade, que os pontos de  $P$  estão contidos no *quadrado unitário*, ou seja,  $P \subset [0, 1]^2$ . Imagine um quadriculado dividindo o quadrado unitário em *células* de diâmetro  $f(p)/2$  (e lado  $f(p)/2\sqrt{2}$ ). Chamamos

Entrada:

$P$ : Conjunto de  $n$  pontos.

Saída:

$p, p'$ : Par de pontos mais próximos de  $P$ .

pontosMaisPróximos( $P$ ):

enquanto  $P \neq \emptyset$ :

$p \leftarrow$  ponto de  $P$  escolhido aleatoriamente

$p' \leftarrow vmp(p)$

$P \leftarrow \text{removerPontos}(P, f(p))$

retorne  $p, p'$

removerPontos( $P, fp$ ):

    para cada  $q \in P$ :

        se  $f(q) \geq fp$ :

$P \leftarrow P \setminus \{q\}$

---

Figura 1.5: Primeira versão do algoritmo que determina o par de pontos mais próximos.

a célula que contém um ponto  $q$  de  $c(q)$ , e definimos a *adjacência* de uma célula  $c(q)$  como a região formada por  $c(q)$  e as no máximo 8 células no seu entorno (figura 1.6). Note que, se  $f(q) \geq f(p)$ , então a adjacência de  $c(q)$  contém somente o ponto  $q$ . Por outro lado, a adjacência de  $c(q)$  conter somente o ponto  $q$  não significa que  $f(q) \geq f(p)$ , mas sim que  $f(q) \geq f(p)/2\sqrt{2}$ . Para removermos eficientemente todo ponto  $q$  tal que a adjacência de  $c(q)$  contém somente  $q$ , usamos uma função hash e a função piso.

Definimos uma função hash  $h$  com parâmetro  $m = n$  e conjunto de chaves sendo o conjunto de todas as células do quadriculado, representadas por números inteiros como definido a seguir. A função piso é importante porque, para representar a célula de um ponto  $q = (q_x, q_y)$  como um número inteiro, fazemos  $c(q) = \lfloor q_x / (f(p)/2\sqrt{2}) \rfloor + k \lfloor q_y / (f(p)/2\sqrt{2}) \rfloor$ , onde  $k = 1 + \lfloor 1 / (f(p)/2\sqrt{2}) \rfloor$ .

Criamos então um vetor  $v$  com  $n$  posições inicializadas com um conjunto vazio. O vetor  $v$  é definido como um vetor de coleções de conjuntos de pontos. Associamos cada ponto  $q \in P$  a um dos conjuntos em  $v[h(c(q))]$ . Desejamos que dois pontos pertençam ao mesmo conjunto se e só se pertencerem à mesma célula, mas duas células distintas podem ter o mesmo valor da função hash. Esta é razão de associarmos  $v[h(c(q))]$  a uma *coleção* de conjuntos, sendo um conjunto (de pontos) para cada célula distinta. Deste modo, a posição  $v[h(c(q))]$  armazena as células cuja função hash tem valor  $h(c(q))$ , e um dos conjuntos de  $v[h(c(q))]$  contém os pontos da célula  $c(q)$ .

Usando o vetor  $v$ , podemos remover todo ponto  $q$  tal que a adjacência de  $c(q)$  contém somente  $q$ . Para isso, examinamos um elemento do conjunto de pontos — de modo a determinar a célula correspondente — e o número de pontos no conjunto — de modo a determinar se a célula contém algum ponto que não seja  $q$  —, conforme o pseudo-código da figura 1.7.

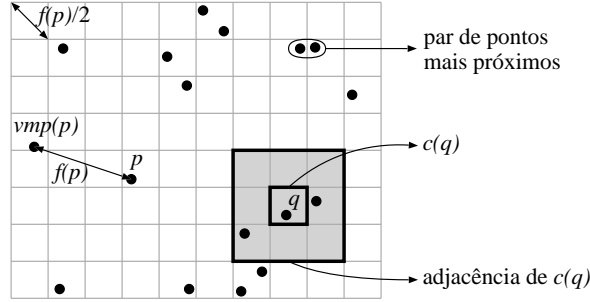


Figura 1.6: Divisão do plano por um quadriculado, com adjacência da célula do ponto  $q$  em cinza.

Infelizmente, a função `removePontos` remove pontos  $q$  com  $f(p)/2\sqrt{2} \leq f(q) \leq f(p)$ . Deste modo, nosso algoritmo não encontra necessariamente o par de pontos mais próximos, mas sim uma *aproximação* do par de pontos mais próximos. Mais especificamente, o algoritmo encontra um par de pontos  $p, p'$  cuja distância  $|p - p'|$  é no máximo  $2\sqrt{2}$  vezes a distância entre o par de pontos mais próximos. Antes de mostrarmos como obter uma solução exata para o problema, a partir da solução aproximada, analisamos a complexidade de tempo do algoritmo.

Primeiro, analisamos a complexidade da função `removePontos`. Para mostrar que a função leva tempo  $O(n)$ , precisamos mostrar que a condição `se` do loop mais interno é avaliada  $O(n)$  vezes. O número de conjuntos  $C \in v[h(x)]$  é igual ao número de células  $y$  com  $h(y) = h(x)$ . Pelo corolário 1.2.2, o valor esperado do número de células  $y$  com  $h(y) = h(x)$  é no máximo 2. O número de células na adjacência de uma célula qualquer é no máximo 9. Portanto, a condição `se` do loop mais interno é avaliada no máximo 18 vezes por ponto de  $P$  e, conseqüentemente, a função `removePontos` leva tempo esperado  $O(n)$ .

Desejamos analisar a complexidade do algoritmo da figura 1.5 usando a função `removePontos`. Note que, a cada iteração, o conjunto  $P$  pode ser ordenado segundo o valor de  $f(p)$  para cada  $p \in P$ . Escolhemos um ponto  $p$  aleatoriamente. Portanto, no caso esperado, pelo menos metade dos pontos  $q \in P$  tem  $f(q) \geq f(p)$ , sendo então removidos. O algoritmo termina quando não há mais pontos em  $P$ . Sendo assim, limitamos o valor esperado da complexidade de tempo usando a linearidade da esperança:

$$\mathbf{E}[T(n)] \leq \sum_{i=0}^{\infty} O(n/2^i) = O(n).$$

Voltamos agora ao problema do algoritmo analisado não determinar o par de pontos mais próximos, mas sim uma aproximação do par de pontos mais próximos. Para obtermos o par de pontos mais próximos a partir da aproximação, usamos um quadriculado e função hash novamente. Seja  $a$  a distância entre o par de pontos retornada pelo algoritmo aproximado. Criamos um quadriculado com células de lado  $a$  e usamos a função hash para organizar

```

removerPontos( $P, fp$ ):
   $P' \leftarrow \{\}$ 
   $v[0 \dots (n-1)] \leftarrow \{\}$ 
   $k \leftarrow 1 + \lfloor 1/(fp/2\sqrt{2}) \rfloor$ 
  seja  $c(q) = \lfloor q_x/(fp/2\sqrt{2}) \rfloor + k \lfloor q_y/(fp/2\sqrt{2}) \rfloor$ 
  para cada  $q \in P$ :
    para cada conjunto  $C \in v[h(c(q))]$ 
      se  $c(C[0]) = c(q)$ 
         $C = C \cup \{q\}$ 
      se  $q$  não foi adicionado a nenhum conjunto  $C$ 
         $v[h(c(q))] = v[h(c(q))] \cup \{\{q\}\}$ 
  para cada  $q \in P$ :
     $remover \leftarrow 1$ 
    para cada célula  $x$  na adjacência de  $c(q)$ :
      para cada conjunto  $C \in v[h(x)]$ 
        se  $c(C[0]) = x$  e  $(c(C[0]) \neq c(q) \text{ ou } |C| > 1)$ 
           $remover \leftarrow 0$ 
      se  $remover \neq 1$ :
         $P' \leftarrow P' \cup \{q\}$ 
  retorne  $P'$ 

```

---

Figura 1.7: Função que remove todos os pontos que não possuem nenhum outro ponto na adjacência de sua célula em tempo  $O(n)$ .

os pontos em um vetor, do mesmo modo que feito anteriormente. Para cada ponto  $q$ , determinamos o ponto mais próximo de  $q$  na adjacência da célula que contém  $q$ , se houver. Note que, como  $a$  é um limite superior para a distância entre o par de pontos mais próximos, e as células têm lado  $a$ , sabemos que o par de pontos mais próximos encontra-se em células adjacentes.

Para analisarmos a complexidade do algoritmo, precisamos saber o número máximo de pontos que podem pertencer a uma mesma célula do quadriculado. Sabemos que não há dois pontos mais próximos que  $a/2\sqrt{2}$ , já que  $a$  é uma aproximação de fator  $2\sqrt{2}$  da distância do par de pontos mais próximos. Dividimos uma célula do quadriculado em 16 sub-células de diâmetro  $a/2\sqrt{2}$  (figura 1.8). Como não há dois pontos mais próximos que  $a/2\sqrt{2}$ , cada sub-célula pode ter no máximo 1 ponto, e uma célula pode ter no máximo 16 pontos. Usando o corolário 1.2.2 de maneira análoga à anterior, mostramos que esta rotina leva tempo  $O(n)$  — e obtemos o par de pontos mais próximos em tempo  $O(n)$ .

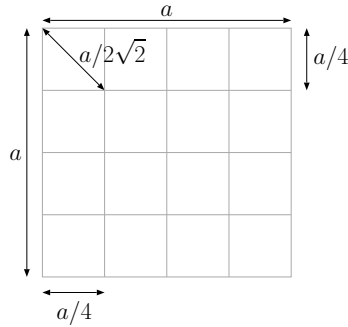


Figura 1.8: Divisão de uma célula em 16 sub-células com no máximo um ponto por sub-célula.

## 1.4 Exercícios

1. Escreva algoritmos randomizados incrementais, com complexidade de tempo  $O(n)$ , para os seguintes problemas:
  - (a) Dado um conjunto de  $n$  pontos  $P$ , e um ponto  $p \in P$ , determinar o menor círculo  $C$  que contém todos os pontos de  $P$ , tal que  $p$  pertence à borda de  $C$ .
  - (b) Dado um conjunto de  $n$  pontos  $P$ , determinar o menor círculo  $C$  que contém todos os pontos de  $P$ .
2. Dados dois conjuntos de pontos  $P_1, P_2$ , separados por uma reta vertical, chamamos de *ponte* a reta  $r$  que contém dois pontos  $p_1 \in P_1$  e  $p_2 \in P_2$ , de modo que todos os demais pontos de  $P_1 \cup P_2$  estão abaixo de  $r$ . Escreva um algoritmo randomizado incremental que determina a ponte em tempo  $O(|P_1| + |P_2|)$ .
3. Dado um conjunto de  $n$  chaves inteiras  $C$ , uma função hash  $h$  é dita *perfeita* quando  $h(x) \neq h(y)$  para todo par de chaves distintas  $x, y$ . Neste exercício, você deve escrever e analisar um algoritmo que, dado um conjunto  $C$ , obtenha uma função hash perfeita  $h : C \rightarrow \{0, \dots, m - 1\}$ . A função  $h$  deve poder ser avaliada em tempo  $O(1)$ . Forneça inicialmente um algoritmo de Monte Carlo e, em seguida, converta este algoritmo em um algoritmo de Las Vegas. Considere que o parâmetro  $m$  é:
  - (a)  $m = O(n^2)$ ;
  - (b)  $m = O(n)$ . (*Dica*: use dois níveis de funções hash.)
4. Escreva um algoritmo que, dados um conjunto de pontos  $P$  e um número real  $r$ , liste todos os pares de pontos  $p_1, p_2$  tal que  $|p_1 - p_2| \leq r$ . O seu algoritmo deve ter complexidade de tempo  $O(n + k)$ , onde  $k$  é o número de pontos listados.
5. Modifique o algoritmo que determina o par de pontos mais próximos para funcionar em espaços  $d$ -dimensionais, onde  $d$  é constante.

## 1.5 Notas bibliográficas

Diversos livros estudam algoritmos randomizados em geometria computacional. De Berg, van Kreveld, Overmars e Schwarzkopf [6] fazem uma excelente introdução ao estudo de geometria computacional, cobrindo diversos algoritmos randomizados, entre eles o algoritmo de programação linear apresentado aqui. Motwani e Raghavan [14] analisam um grande número de algoritmos randomizados de programação linear, geometria computacional e funções hash. Mulmuley [15] introduz problemas fundamentais de geometria computacional usando algoritmos randomizados. Em língua portuguesa, Figueiredo e Carvalho [7] e também Rezende e Stolfi [19] escreveram ótimos textos introdutórios de geometria computacional.

O algoritmo de programação linear apresentado aqui foi descoberto por Seidel [18]. Caso consideremos a dimensão  $d$  como uma variável assintótica, a complexidade de tempo é  $O(d!n)$ . Outros algoritmos randomizados são mais eficientes em função de  $d$ , tendo complexidades como  $O(d^2n + e^{\sqrt{d \ln d}})$  [13].

Funções hash vêm sendo estudadas na ciência da computação desde a década de 50 e costumam ser apresentadas em livros introdutórios de algoritmos [5, 10]. Knuth [11] cobre funções hash, porém considerando a distribuição probabilística das chaves. Funções hash que garantem resultados probabilísticos independentemente da distribuição das chaves foram introduzidas por Carter e Wegman [1].

O algoritmo clássico para o par de pontos mais próximos leva tempo  $O(n \log n)$  e é baseado no paradigma de divisão e conquista [16]. A primeira solução randomizada foi descoberta por Rabin [17] e leva tempo  $O(n)$ . Um algoritmo determinístico que usa a função piso e leva tempo  $O(n \log \log n)$  foi descoberto por Fortune e Hopcroft [8]. Khuller e Matias [9] descobriram o algoritmo descrito aqui.

Grande parte dos algoritmos randomizados para geometria computacional possuem versões de-randomizadas com complexidades similares. Chazelle e Friedman [3] introduziram diversas técnicas para de-randomizar algoritmos geométricos. Matoušek [12] compilou diversos resultados de de-randomização em geometria computacional. O algoritmo randomizado incremental para o fecho convexo em tempo ótimo  $O(n^{\lfloor (d-1)/2 \rfloor} + n \log n)$  foi descoberto por Clarkson e Shor [4] e de-randomizado por Chazelle [2].

# Bibliografia

- [1] J. L. Carter and M. N. Wegman. Universal classes of hash functions. *J. of Computer and System Sciences*, 18(2):143–154, 1979.
- [2] B. Chazelle. An optimal convex hull algorithm in any fixed dimension. *Discrete Comput. Geom.*, 10:377–409, 1993.
- [3] B. Chazelle and J. Friedman. A deterministic view of random sampling and its use in geometry. *Combinatorica*, 10(3):229–249, 1990.
- [4] K. L. Clarkson and P. W. Shor. Applications of random sampling in computational geometry II. *Discrete Comput. Geom.*, 4:387–421, 1989.
- [5] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2001.
- [6] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer, second edition, 2000.
- [7] L. H. Figueiredo and P. Cezar. *Introdução à Geometria Computacional*. 18° Colóquio Brasileiro de Matemática, IMPA, 1991.
- [8] S. Fortune and J. E. Hopcroft. A note on Rabin’s nearest neighbor algorithm. *Information Processing Letters*, 8(1):20–23, 1979.
- [9] S. Khuller and Y. Matias. A simple randomized sieve algorithm for the closest-pair problem. *Inf. Comput.*, 118(1):34–37, 1995.
- [10] J. Kleinberg and É. Tardos. *Algorithm Design*. Addison-Wesley, 2006.
- [11] D. Knuth. *The Art of Computer Programming vol. 3: Sorting and Searching*. Addison-Wesley, 1973.
- [12] J. Matoušek. Derandomization in computational geometry. *J. Algorithms*, 20(3):545–580, 1996.
- [13] J. Matoušek, M. Sharir, and E. Welzl. A subexponential bound for linear programming. *Algorithmica*, 16:498–516, 1996.

- [14] R. Motwani and Prabhakar Raghavan. *Randomized Algorithms*. Cambridge Univ. Press, 1995.
- [15] K. Mulmuley. *Computational Geometry: An Introduction through Randomized Algorithms*. Prentice-Hall, Englewood Cliffs, 1994.
- [16] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, 1985.
- [17] M. O. Rabin. Probabilist algorithms. In J. F. Traub, editor, *Algorithms and Complexity, Recent Results and New Directions*. Academic Press, New York, 1976.
- [18] R. Seidel. Small-dimensional linear programming and convex hulls made easy. *Discrete Comput. Geom.*, 6:423–434, 1991.
- [19] J. Stolfi and P. J. Rezende. *Fundamentos de Geometria Computacional*. IX Escola de Computação, Recife, PE, 1994.