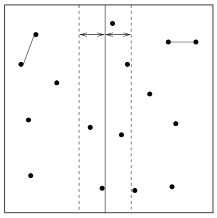
MAC 5747 GEOMETRIA COMPUTACIONAL

PROBLEMA DO PAR MAIS-PRÓXIMO

1. Introdução



No problema do par mais-próximo (*The Closest Pair Problem*) são dados n pontos e deseja-se encontrar dois cuja distância entre eles é mínima. Uma aplicação prática deste problema (na sua versão 'cinética') é em controle de tráfego aéreo: os dois aviões que estão em maior perigo de colisão são aqueles que estão mais próximos. Este problema pode ser resolvido facilmente, por um algoritmo do tipo força-bruta, em tempo $O(dn^2)$ onde d é a dimensão do espaço: nós simplesmente computamos a distância entre cada par de pontos enquanto mantemos um candidato a par mais-próximo até o momento (veja Seção 2).

Nas Seções 3 e 4 veremos que o problema do par mais-próximo pode ser resolvido por um algoritmo do tipo divisão-e-conquista em tempo $O(dn \log n)$ (cf. Shamos e Hoey [6]).

Apesar da formulação do problema fazer sentido em qualquer dimensão, nos ateremos aqui ao espaço bidimensional. Mais especificamente, consideraremos que os pontos estão $\mathbb{E}^2$ (= plano + distância euclidiana).

Problema 1 (Problema do Par Mais-Próximo)**.** *Dados pontos $p_1, \ldots, p_n$ no plano, encontrar pontos p_i e p_j, $i \neq j$, tal que a distância entre p_i e p_j é a menor entre todas as distância entre os pares de pontos dados.*

Se mais de um par de pontos tem distância mínima entre eles, então aceitamos como resposta qualquer um desses pares e o chamamos de *par mais-próximo*. Por questões estéticas escrevemos *o* par mais-próximo em vez de *um* par mais-próximo, mesmo podendo existir mais que um par mais-próximo.

Por $\mathbb{E}^d$ denotamos o espaço euclidiano d-dimensional, ou seja, o espaço das d-uplas $(x_1, \ldots, x_d)$ de números reais x_i $(i = 1, \ldots, d)$ onde a distância entre pontos $p = (p_1, \ldots, p_d)$ e $q = (q_1, \ldots, q_d)$, denotada por distância(p, q), é dada por

$$\text{distância}(p, q) := \sqrt{\sum_{i=1}^{d} (p_i - q_i)^2}.$$

Se $n = 1$, convencionaremos que a distância entre o par mais-próximo é infinita.

Date: 21 de agosto de 2002.

Observação. Os tópicos que cobrireros a seguir foram extraídos do Capítulo 35 de Cormen, Leiserson e Rivest [2], Capítulo 8 de Laszlo [4], Capítulo 5 de Preparata e Shamos [5], e Capítulo 3 de Resende e Stolfi [3].

2. Algoritmo ingênuo

Com foi mencionado na introdução, a primeira idéia para a resolução do problema do par mais-próximo nos leva a um algoritmo do tipo força-bruta: nós simplesmente computamos a distância entre cada par de pontos, enquanto mantemos o par mais-próximo encontrado até o momento.

Algoritmo INGÊNUO(P)
`Recebe:` uma coleção $P = \{p_1, \ldots, p_n\}$ de pontos em $\mathbb{E}^d$.
`Devolve:` um par par mais-próximo $\{p, q\}$ em P.
1. $\delta \leftarrow +\infty$;
2. **para** $i = 1, \ldots, n-1$ **faça**
3.   **para** $j = i+1, \ldots, n$ **faça**
4.   **se** distância$(p_i, p_j) < \delta$ **então**
5. $\delta \leftarrow$ distância(p_i, p_j);
6. $p \leftarrow p_i$;
7. $q \leftarrow p_j$;
8. **devolva** $\{p, q\}$.

2.1. Correção do algoritmo.

Não há muito o que dizer sobre a correção do algoritmo. Como exercício, encontre invariantes apropriado para o algoritmo, demonstre que são de fato invariantes e usando estes invariantes demonstre a correção do algoritmo algoritmo INGÊNUO.

2.2. Análise do algoritmo.

Também não há muito o que dizer aqui. Como existem, $\binom{n}{2} = n(n-1)/2$ pares de pontos e o algoritmo calcula a distância entre cada um desse pares, então o consumo de tempo do algoritmo é $\Theta(n^2)$. A quantidade de espaço utilizada pelo algorimo é proporcional ao número de pontos dados.

Um pergunta natural agora sera "É possível encontrar um algoritmo que gasta tempo $o(n^2)$? Ou seja, é possível encontrar um algoritmo mais rápido que o algoritmo INGÊNUO? Por algoritmo 'mais rápido' entenda-se um algoritmo que consome menos tempo segundo o critério de consumo de tempo assintótica no pior caso. Pelo que foi dito na introdução sabemos que a resposta para a pergunta é *sim.* Se estivessemos trabalhando em $\mathbb{E}^1$ em de $\mathbb{E}^d$ a resposta seria obviamente *sim.*

Algoritmo INGÊNUO2(P)
`Recebe:` uma coleção $P = \{p_1, \ldots, p_n\}$ pontos em $\mathbb{E}^1$.
`Devolve:` um par mais-próximo $\{p, q\}$ em P.
1. Ordene $p_1, \ldots, p_n$ de tal forma que $p_1 \leq \ldots \leq p_n$.
2. $\delta \leftarrow +\infty$;
3. **para** $i = 1, \ldots, n-1$ **faça**
4.   **se** distância$(p_i, p_i + 1) < \delta$ **então**
5. $\delta \leftarrow$ distância(p_i, p_j);

6. $\quad\quad p \leftarrow p_i$;
7. $\quad\quad q \leftarrow p_j$;
8. **devolva** $\{p, q\}$.

O consumo de tempo da linha 1 é $\Theta(n \log n)$ e o tempo gasto com o laço envolvendo as linhas 3–7 é $\Theta(n)$. Logo o cosnumo de tempo do algoritmo INGÊNUO2 é $\Theta(n \log n)$, que é substacialmente mais rápido que $\Theta(n^2)$. Infelizmente, não parece ser possível estender a idéia do algoritmo acima para $\mathbb{E}^2$ (pelo menos eu não sei como fazer isto). A propósito, a idéia de olhar o problema em um espaço de dimensão menor é justamente encontrar um algoritmo mais rápido em que a idéia central possa ser estendida para o espaço de dimensão maior.

Na próximo seção veremos um algoritmo que usa a técnica de divisão-e-consquista para resolver o problema do par mais-próximo em $\mathbb{E}^1$ em tempo $O(n \log n)$. Este algoritmo, que é um pouco mais complicado que os desta seção, será estendido para resolver o problema do par mais-próximo em $\mathbb{E}^2$ (na realidade, em dimensão arbitrária, porém fixa) em tempo $O(n \log n)$.

3. Par mais-próximo na reta

Nesta seção veremos uma algoritmo do tipo divisão-e-conquista que encontra, em tempo $O(n \log n)$, o par mais-próximo de uma coleção de n pontos em $\mathbb{E}^1$. A técnica de divisão-e-conquista que veremos será estendida na próxima seção para obtermos um algoritmo que atinge a mesma complexidade de tempo, i.e. $O(n \log n)$, para encontrar um par de pontos mais-próximo no plano.

3.1. **Descrição do algoritmo.** O algoritmo é recursivo. Na base da recursão, se P tem poucos pontos (menos que uma constante k_0) então o problema é resolvido diretamente via força-bruta. Caso contrário, a idéia do algoritmo é, em cada nível da recursão, executar as três fases a seguir:

Dividir: Compute a mediana m da coleção P. Particione a coleção P em subcoleções E e D tal que $e \leq m < d$ para todo ponto e em E e d em D.

Conquistar: Resolva, separadamente, o problema do par mais-próximo para E e D. Sejam $\{e_1, e_2\}$ um par mais-próximo em E e $\{d_1, d_2\}$ um par mais próximo em D.

Combinar: Seja $\delta = \min\{|e_1 - e_2|, |d_1 - d_2|\}$. Um par mais-próximo em P é $\{e_1, e_2\}$ ou $\{d_1, d_2\}$ ou $\{e, d\}$, onde e é um ponto em E e d é um ponto de D. Neste último caso a distância entre e e d deve ser menor que δ. Quantos pares de pontos são candidatos a $\{e, d\}$? A reposta é no máximo 1! A saber, $e = m$ (lembremos que m pertence a E) e $d := \min\{p | p \in D\}$. (Veja Figura 1.)

Abaixo encontra-se uma descrição um pouco mais detalhada do algoritmo (`C-like`). Para simplicar a exposição, o algoritmo abaixo devolve somente a distância mínima entre um par de pontos mais-próximos de P.

Algoritmo `ParMaisPróximo1`(P)
`Recebe:` Uma coleção $P = \{p_1, \ldots, p_n\}$ pontos em $\mathbb{E}^1$.

`Devolve:` δ, a distância entre um par mais-próximo em P.

1. **se** $n = 1$ **então devolva** $+\infty$;
2.   **senão se** $n = 2$ **então devolva** distância(p_1, p_2);
3.   **senão**
4. $m \leftarrow$ mediada de P;

5   Sejam $E := \{p \in P \mid p \leq m\}$ e $D := \{q \in P \mid q > m\}$;

6. $\delta_E \leftarrow$ `ParMaisPróximo1`(E);
7. $\delta_D \leftarrow$ `ParMaisPróximo1`(D);
8. $e \leftarrow m (= \max\{p : p \in E\}(= m)$;

9             $d \leftarrow \min\{p : p \in D\}$;

10. $\delta \leftarrow \min\{\delta_E, \delta_D, d - e\}$;
11.   **devolva** δ;

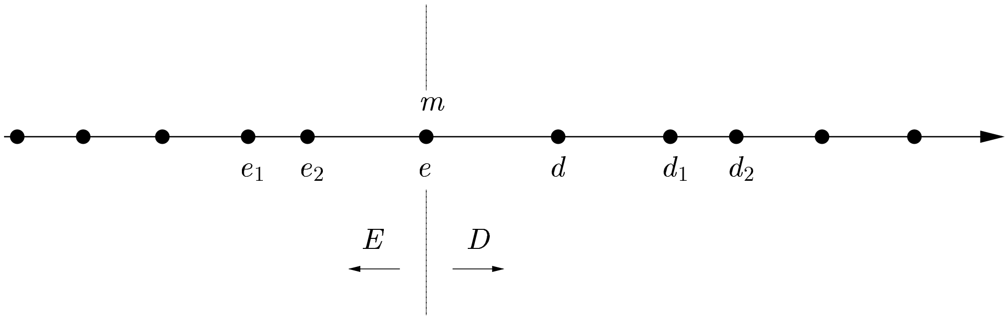


FIGURA 1. Ilustração da fase *Divisão* e *Combinar* do algoritmo.

3.2. **Correção do algoritmo.** Não é difícil ver que o algoritmo está correto. A observação importante é que, na fase COMBINAR, somente precisamos computar a distância entre um par de pontos $\{e, d\}$ com e em E e d em D.

3.3. **Análise do algoritmo.** Se, em um pré-processamento, ordenarmos os pontos, então as Linhas 4,5, 8, 9 e 10 do algoritmo podem ser executas em tempo constante. Assim, a complexidade de tempo $T(n)$ do algoritmo, excluindo o custo do pré-processamento, que mede o número de operações primitivas (ou passos) feitos pelo algoritmo no pior caso satisfaz a recorrência

$$T(n) \leq \begin{cases} T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + c & \text{se } n > 2 \\ b & \text{se } n = 2 \\ a & \text{se } n = 1 \end{cases},$$

onde a, b e c são constantes positivas. Logo, $T(n) = O(n)$. Portanto, o consumo de tempo do algoritmo é $\Theta(n \log n)$, já que o tempo gasto com o pré-processamento é $\Theta(n \log n)$.

EXERCÍCIO. Mostre que $T(n) = O(n)$.

4. Par mais-próximo no plano: Um algoritmo por divisão-e-conquista

Veremos agora a generalização do algoritmo da seção anterior para o espaço $\mathbb{E}^2$. A descrição que veremos é a mesma que se encontra no Capítulos 5 de Preparata e Shamos [5], e Capítulo 3 de Resende e Stolfi [3]. A fase Combinar que veremos é um pouco diferente da apresentada no Capítulo 35 de Cormen, Leiserson e Rivest [2] e Capítulo 8 de Laszlo [4] (veja os exercícios).

4.1. **Descrição do algoritmo.** Denotaremos por $P = \{p_1, \ldots, p_n\}$ a coleção dada de pontos de $\mathbb{E}^2$. Se o número de pontos for menor que uma constante k_0 (digamos, $k_0 = 3$) então o algoritmo é resolvido através de força-bruta em tempo contante. Caso contrário devemos, em cada nível da recursão, executar as fases Dividir, Conquistar e Combinar, como passamos a descrever a seguir.

4.1.1. *Dividir.* Nesta fase do algoritmo a coleção P de pontos é particionada em coleções E e D tal que cada ponto de E esteja à esquerda de cada ponto em D. Utilizando um pré-processamento que ordena os pontos de P segundo as suas x-coordenadas podemos fazer com que, em cada nível da recursão, está fase gaste tempo $O(|P|)$, ou mesmo constante, dependendo da implementação (Veja Figura 2.)

4.1.2. *Conquistar.* O problema deve ser resolvido recursivamente para E e D, obtendo assim δ_E e δ_D, as distâncias entre um par mais-próximo pontos $\{e_1, e_2\}$ em E e um par mais-próximo $\{d_1, d_2\}$ em D, respectivamente.

4.1.3. *Combinar.* Agora sim, chegamos ao **coração do algoritmo**. Para que o algoritmo gaste tempo $O(n \log n)$ é necessário que esta fase consuma uma quantidade de tempo proporcional ao número de pontos em P (complexidade de tempo linear).

Seja $\delta := \min\{\delta_E, \delta_D\}$ e seja l uma reta tal que todo ponto de E está à esquerda ou sobre l e todo ponto de D está à direita ou sobre l. A exemplo do que ocorreu no caso unidimensional, se o par mais-próximo está em E ou D a recursão nos devolve o par e não temos com que nos preocupar. Entretanto, pode ocorrer que o par mais-próximo de P seja da forma $\{e, d\}$ com e em E e d em D. Se este é o caso temos que o par está na faixa M de largura 2δ que tem ao centro a reta l. A situação encontra-se ilustrada na Figura 2.

Na versão unidimensional do problema tinhamos um candidato a e e um candidado a d. Quantos candidatos a par mais-próximo $\{e, d\}$ com e em E e d em D nós temos? Fazendo uma análise apressada pode parecer que podemos ter que, no pior caso, calcular as distâncias entre cada par de pontos $\{e, d\}$ com e em E e d em D, o que nos levaria a uma complexidade de tempo de $\Theta(n^2)$ para o algoritmo (imagine a situação em que todos os pontos de P estão na faixa M de largura 2δ). Através de uma análise um pouco mais cuidadosa podemos reduzir substancialmente este número.

Sejam E' a subcoleção dos pontos de E que estão a uma distância menor do que δ da linha l e seja D' a correspondente subcoleção de pontos de P'_R. Seja e um ponto qualquer de E'. Já encontrarmos, através da recursão, um

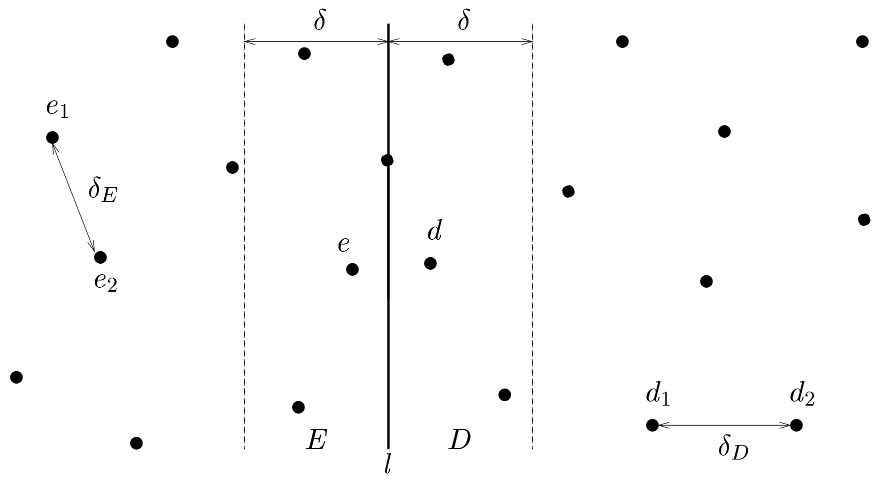


FIGURA 2. Ilustração da fase DIVIDIR e COMBINAR do algoritmo.

par de pontos cuja distância entre eles é δ. Logo, precisamos apenas calcular a distância entre e e todos os pontos d em D' que têm alguma chance de estar a uma distância menor que δ. Assim, os candidatos a d são os pontos de D' que estão no retângulo R_e de altura 2δ e largura δ ilustrado na Figura 3.

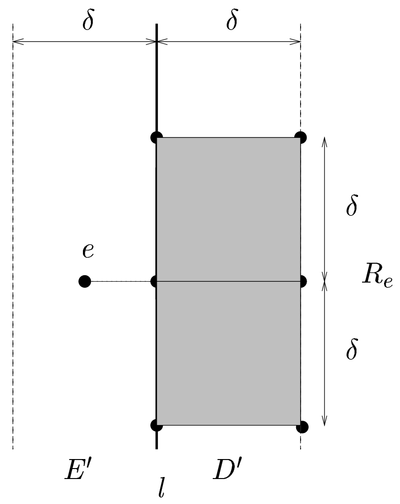


FIGURA 3. Precisamos calcular a distância entre e e os pontos de D que estão em R_e.

No máximo, quantos pontos de D' podem estar contidos no retângulo R_e? No máximo 6 pontos! Isto significa que para cada ponto e de E' precisamos calcular a distância entre e e no máximo 6 pontos de D'. Logo, o número de distância que teremos que calcular na fase COMBINAR não é superior a $6n$. Portanto, o número total de distâncias $D(n)$ que devemos calcular ao longo da execução do algoritmo, no pior caso, satisfaz a relação de recorrência

$$D(n) \leq \begin{cases} D(\lceil n/2 \rceil) + D(\lfloor n/2 \rfloor) + cn & \text{se } n > k_0 \ (k_0 = 3) \\ a & \text{se } n \leq k_0 \end{cases},$$

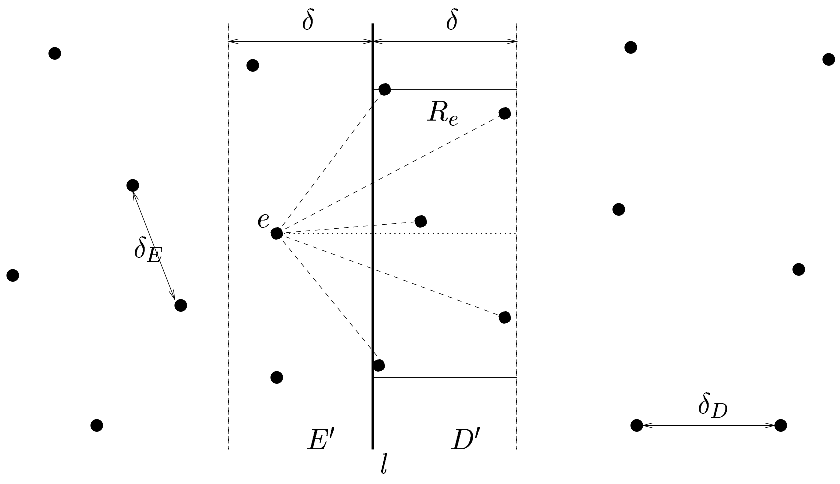


FIGURA 4. Enquanto o apontador para e está fixo um apontador para D' oscila em um intervalo de largura 2δ.

onde a e b são constantes positivas. Como sabemos $D(n) = O(n \log n)$.

Apesar de estarmos indo bem até agora, nós não temos ainda um algoritmo de complexidade de tempo $O(n \log n)$, pois não sabemos, para cada ponto e em E', quais são esses 6 pontos que temos que calcular a distância até e. Mas isto, não será grande problema, nós podemos proceder como segue.

Projetemos e e todos os pontos de D' em l (na realidade, em termos de implementação, nós só precisamos ordenar os pontos de acordo com as suas y-coordenadas). Considere os pontos de D' cuja projeção em l está a uma distância de no máximo δ da projeção do ponto e (sabemos que são no máximo 6 pontos). Ou seja, para cada ponto $e = (d_x, d_y)$ em E' basta calcularmos a distância entre e e os pontos $d = (d_x, d_y)$ em D' que satisfazem $|e_y - d_y| < \delta$.

Suponha que os pontos de E' e D' estão ordenados em relação a suas y-coordenadas. Então podemos, em tempo linear, percorrer os pontos de E' enquanto computamos as distâncias entre pontos de E' e D': um apontador para a lista de pontos E' avança enquanto um apontador para D' oscila em um intervalo de 2δ. A situação está ilustrada na Figura 4.

Seja $\{e, d\}$ o par mais-próximo na faixa de largura δ e centrada em l onde e está em E' e d está em D'. Este par foi determinado como descrito acima. Finalmente, só resta o algoritmo devolver o par de pontos mais-próximo entre os pares $\{e_1, e_2\}, \{d_1, d_2\}$ e $\{e, d\}$ e a distância δ entre esse par de pontos.

4.1.4. *Correção.* A correção do algoritmo pode ser demonstrada por indução no número n de pontos. A base da indução é $n < k_0$ onde o algoritmo encontra o par mais-próximo através de força-bruta.

Suponha por hipótese de indução que o algoritmo determina, corretamente, o par mais-próximo de uma coleção com até $\lceil n/2 \rceil$ pontos no plano, $n \geq 3$. Mostraremos agora que o algoritmo encontra o par mais-próximo de uma coleção com até n pontos, $n \geq 4$. A fase DIVIDIR particiona P

em duas subcoleções E e D com $\lceil n/2 \rceil$ e $\lfloor n/2 \rfloor$ elementos, respectivamente, onde todos os pontos de E estão à esquerda ou sobre uma reta l e os pontos de D estão à direita ou sobre a mesma reta l. Por hipótese de indução, as chamadas recursivas na fase CONQUISTAR retornam o par mais-próximo $\{e_1, e_2\}$ de E e um par mais-próximo $\{d_1, d_2\}$ de D. Seja

$$\delta := \min\{\text{distância}(e_1, e_2), \text{distância}(d_1, d_2)\}.$$

Considere a faixa vertical de largura 2δ centrada na reta l. O algoritmo, na sua fase COMBINAR, determina um par de pontos mais-próximo $\{e, d\}$ nesta faixa tal que e está em E e d está em D. Ao final da fase COMBINAR o algoritmo devolve corretamente o par mais-próximo da coleção P de n pontos (veja discussão acima sobre a fase COMBINAR).

4.1.5. *Implementação e análise do algoritmo.* . Seja $T(n)$ o número total de operações primitivas feitas pelo algoritmo acima, no pior caso, para encontrar o par mais-próximo de um coleção P com n pontos. Queremos mostrar que $T(n) = O(n \log n)$, para isto precisamos que $T(n)$ satisfaça a seguinte relação de recorrência

$$T(n) \leq \begin{cases} T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + cn & \text{se } n > k_0 \ (k_0 = 3) \\ a & \text{se } n \leq k_0. \end{cases},$$

para constantes positivas a e c. A dificuldade aqui é mostra que as fases DIVIDIR e COMBINAR podem, em cada nível da recursão, ser executadas em tempo linear. Se gastarmos tempo $O(n \log n)$ um pré-processamento ordenando os pontos de P de acordo com as suas x-coordenadas então não é difícil ver que a fase DIVIDIR pode ser realizada em tempo linear. De fato, tome l como sendo a reta vertical que passa pelo ponto $m = (m_x, m_y)$, onde m_x é a mediana das abscissas dos pontos em P. Se amarzenarmos os pontos em P, ordenados de acordo com as suas x-coordenadas, em um vetor X, então ao longo da execução do algoritmo não precisaremos ter cópias desse vetor X, basta passarmos como parâmetros dois apontadores para denotar as subcoleções E e D.

Para realizarmos a fase combinar também será conveniente guardarmos os pontos P, ordenados segundo as suas y-coordenadas, em um vetor Y. Isto também será feito durante a fase de pré-procesamento. Tendo este vetor Y em mãos precisamos, em cada execução da fase COMBINAR construir vetores Y_E e Y_D contendo os pontos em E e D, respectivamente, ordenados segundo as suas y-coordenadas. Ademais, isto precisa ser feito em tempo linear. Para tal podemos fazer o oposto do que faz o algoritmo INTERCALA (MERGE) no MERGESORT: particionamos um vetor ordenado em dois vetores ordenados, com descrito a seguir.

comprimento(Y_E) $\leftarrow$ comprimento(Y_D) $\leftarrow 0$;
para $i = 1, \ldots, n-1$ **faça**
    se $Y[i] \in P_L$ **então**
        comprimento(Y_L) $\leftarrow$ comprimento(Y_L) $+ 1$;
        Y_E[comprimento(Y_E)] $\leftarrow Y[i]$;
    senão
        comprimento(Y_D) $\leftarrow$ comprimento(Y_D) $+ 1$;

8. $\quad Y_D[\text{comprimento}(Y_D)] \leftarrow Y[i]$;

Estes vetores ordenados Y_E e Y_D podem ser passados durante a chamada recursiva. Tendo estes vetores ordenados a fase COMBINAR pode ser realizada em tempo linear. De fato, a fim de determinar o par mais-próximo $\{e, d\}$, com e em Y_E e d em Y_D, podemos percorrer, em tempo linear, os vetores os vetores Y_E e Y_D, como foi descrito anteriormente.

O tempo gasto com pré-processamento (ordenação dos pontos de acordo com as x-coordenadas e y-coordenadas) é $O(n \log n)$ e o tempo gasto, em cada nível da recursão, com as fases DIVIDIR e COMBINAR é linear. Logo, $T(n) = O(n \log n)$.

Se $T'(n)$ denota o número total de operações primitivas feitas pelo algoritmo tempos então que $T'(n) = O(n \log n) + T(n) = O(n \log n)$, como desejavamos mostrar. O seguinte teorema resume toda está discussão.

Teorema 2. *O par mais-próximo de um conjunto de n pontos no plano pode ser encontrado em tempo $O(n \log n)$.* ∎

Observação. O algoritmo por divisão-e-conquista visto nesta seção pode ser estendido para $\mathbb{E}^d$. O algoritmo resultante tem complexidade de tempo $O(dn \log n)$ (cf. Capítulo de Preparata e Shamos [5]). O ponto chave do algoritmo é que os pontos envolvidos na fase COMBINAR são 'esparço', no seguinte sentido: dados um número real $\delta > 0$ e um inteiro $c \geq 1$, uma coleção de pontos P de $\mathbb{E}^d$ tem *esparcidade* c para um dado δ se qualquer hypercubo de lado 2δ tem no máximo c pontos de S.

EXERCÍCIO. Mostre que na fase COMBINAR do algoritmo que vimos para encontra o par mais-próximo no plano, a coleção P tem esparcidade 12.

5. COTA INFERIOR

Nesta seção observaremos que $\Omega(n \log n)$ é cota inferior para o Problema do Par Mais Próximo e, portanto, o algoritmo visto na seção anterior tem complexidade de tempo, no pior caso, de $\Theta(n \log n)$ e é ótimo.

Lembremos que o Problema da Unicidade de Elementes consiste em: dados n números reais determinar se eles são todos distintos. este problema tem uma cota inferior de $\Omega(n \log n)$, no pior caso (cf. Ben-Or [1]).

Para resolvermos o Problema da Unicidade de Elemento para uma dada coleção de números P podemos, simplesmente, fornecer esses número para um algoritmo que resolve o Problema do Par Mais-Próximo. Se a distância entre o par mais-próximo é zero então P tem elementos repetidos, caso contrário todos os numeros de P são distintos. Com isto vemos que se existisse um algoritmo de complexidade de tempo $o(n \log n)$ para o Problema do Par Mais-Próximo, então teriamos um algoritmo de complexidade de tempo $o(n \log n)$ para o Problema da Unicidade de Elementos, contrariando a cota inferior de $\Omega(n \log n)$ para este último problema.

Teorema 3. *Qualquer algoritmo (no modelo de árvore de decisão algébrica) que determina o par mais-próximo de uma coleção de n pontos requer, no pior caso, $\Omega(n \log n)$ operações primitivas.* ∎

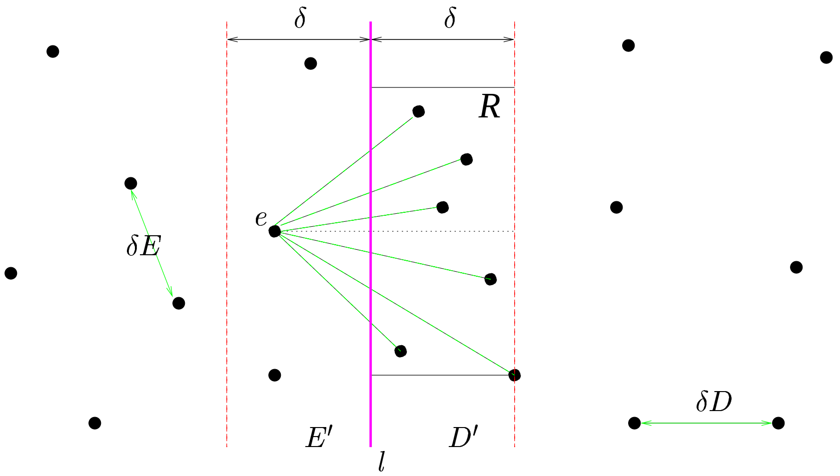
Uma conseqüência imediata deste teorema é o seguinte corolário.

Corolário 4. *O algoritmo por divisão-e-conquista da seção anterior é ótimo.*

∎

6. Exercícios

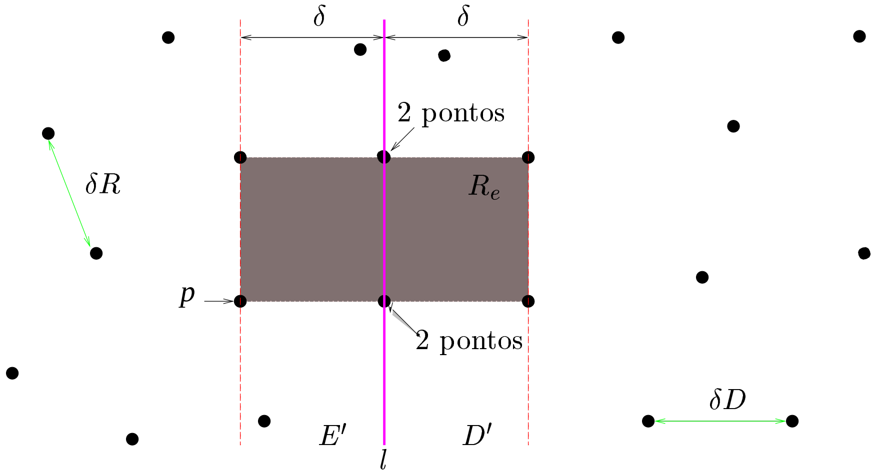
1. (Suficiente) Considere a fase Combinar do algoritmo de divisão-e-conquista para o Problema do Par Mais-Próximo, visto em sala de aula. Mostre que, nesta fase, para cada ponto $e \in E'$ é **suficiente** o algoritmo calcular a distância entre e e no máximo 6 pontos em D'.
2. (Necessário) Considere a fase Combinar do algoritmo por divisão-e-conquista para o Problema do Par Mais-Próximo, visto em sala de aula. Nesta fase, para cada ponto $e \in E'$ o algoritmo calcula a distância entre e e no máximo 6 pontos em D'. Você consegue mostrar um exemplo onde é realmente **necessário** calcularmos a distância entre e e 6 pontos $d_1, \ldots, d_6$ de D'? (Ou seja, o seu exemplo deve mostrar um ponto e em E e pontos $d_1, \ldots, d_6$ contidos no retângulo R_e de tal forma que se o algoritmo calcula a distância entre e e cada um dos pontos (digamos) $d_1, \ldots, d_5$ então o algoritmo não devolve o par mais-próximo (que por azar é o par $\{e, d_6\}$.) Se você não conseguir encontrar um tal exemplo, então tente mostrar que é **suficiente** o algoritmo calcular a distância entre cada ponto de E' e menos do que 6 pontos de D'.



3. Considere agora a seguinte pequena variação da fase Combinar do algoritmo. Seja M a lista dos pontos em $E' \cup D'$ ordenados em relação as suas y-coordenadas (ordenadas). Percorra a lista M e para cada ponto $m \in M$ calcule a distância entre m e os 7 pontos que seguem m na lista M. (Note que algumas vezes poderemos estar calculando a distância entre pontos de E' ou entre pontos de D'.) Mostre que com esta variação da fase Combinar o algoritmo corretamente encontra um par mais-próximo.

   [Note que com esta variação não precisamos mais ficar testando as y-coordenadas dos pontos que terão as suas distâncias calculadas.

   Sugestão: Veja a figura abaixo ou de uma olhada em [2] página 910.]



4. Considere ainda a variação da fase COMBINAR proposta no exercício anterior. Você mostrou que é **suficiente** calcularmos as distâncias entre cada ponto $m \in M$ e os 7 pontos que seguem m na lista M para que o algoritmo esteja correto. Será que é **necessário** calcularmos essas 7 distâncias? Ou seja, será que não poderiamos calcular a distância entre cada ponto $m \in M$ e os (digamos) 6 pontos que seguem m na lista M e ainda assim o algoritmo estaria correto? Apresente um exemplo mostrando que é necessário calcularmos essas 7 distâncias ou mostre que é possível encontrarmos o par mais próximo calculando menos do que 7 distâncias para cada $m \in M$.

   [Note que para a ordem de complexidade do algoritmo tanto faz se para cada $m \in M$ calculamos 7, 6 ou 100000 distâncias—o algoritmo continua tendo complexidade de tempo $O(n \log n)$.]
5. (Exercício 33.4-1 de [2]) Considere ainda a variação da fase COMBINAR proposta no Exercício 3. O professor Maqui Sperto teve uma idéia genial e veio com um novo esquema para que o algoritmo encontre o par mais-próximo verificando, na sua fase COMBINAR, somente a distância entre cada ponto $m \in M$ e os 5 pontos que estão a seguir de m em M. A idéia é sempre colocar os pontos da reta l no conjunto E. Então, não poderá haver um par de pontos coincidentes sobre a reta l com um ponto em E e outro ponto em D. Portanto, no máximo 6 pontos podem estar na retâgulo de dimensões $\delta \times 2\delta$. Onde está a bobagem no esquema proposto pelo professor Sperto.
6. (Exercício 33.4-2 de [2]) Sem aumentar a complexidade de tempo assintótica do algoritmo, mostre como garantir que o conjunto de pontos passados para a primeira chamada recursiva do algoritmo não contenha pontos coincidentes. Prove que então é suficiente que o algoritmo verifique os 6 (e não 7) pontos que seguem cada ponto em M. Por que não é suficiente verificar somente 5 pontos? Ou é suficiente?
7. (Exercício 33.4-3 de [2]) A distância entre dois pontos pode ser definida de diversas maneiras além da Euclidiana. No plano, a L_m-distância entre dois pontos $p = (p_x, p_y)$ e $q = (q_x, q_y)$ é dada por $(|p_x - q_x|^m + |p_y - q_y|^m)^{1/m}$. Portanto, a distância Euclidiana é a L_2-distância. Modifique o algoritmo de divisão-e-conquista para o Problema do Par Mais Próximo

de tal forma que ele devolva o par de pontos mais-próximo em relação a L_1-distância (também conhecida como *Manhattan distance*).

8. (Exercício 33.4-4 de [2]) Dados dois pontos p e q no plano, a L_∞-distância entre eles é dada por $\max\{|p_x - q_x|, |p_y - q_y|\}$. Modifique o algoritmo para o par mais próximo para que ele encontre um par mais-próximo de acordo com a L_∞-distância.

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] 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.

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

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

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

[6] M.I. Shamos e D. Hoey, Closest point problems, *Proc. 16th Annual IEEE Symposium in Foundations of Computer Science*, 1975, pp. 151–162.