

Apostila Introdutória de Algoritmos

Guilherme D. da Fonseca
Celina M. H. de Figueiredo

Versão rascunho para o curso de Análise de Algoritmos da Unirio
2009.

18 de Setembro de 2009

Divisão e Conquista

Resolver problemas pequenos é quase sempre mais simples que resolver problemas maiores. É natural dividir um problema grande em sub-problemas menores e resolver cada um dos sub-problemas separadamente. Feito isto, temos que combinar as soluções dos problemas menores para obtermos a solução do problema total. Os algoritmos de divisão e conquista têm, então, três fases: dividir, conquistar e combinar.

Na primeira fase, a divisão, o problema é decomposto em dois (ou mais) sub-problemas. Em alguns algoritmos, esta divisão é bastante simples, enquanto em outros, é a parte mais delicada do algoritmo.

Na segunda fase, a conquista, resolvemos os sub-problemas. A beleza da técnica reside no fato de que os problemas menores podem ser resolvidos recursivamente, usando o mesmo procedimento de divisão e conquista, até que o tamanho do problema seja tão pequeno que sua solução seja trivial ou possa ser feita mais rapidamente usando algoritmos mais simples.

Na terceira fase, a combinação das soluções, temos que unir as soluções dos problemas menores para obtermos uma solução unificada. Este procedimento nem sempre é trivial, e muitas vezes pode ser simplificado se a divisão (primeira fase) for feita de modo inteligente.

3.1. Envelope Superior

O envelope superior de um conjunto de retas S no plano cartesiano é a seqüência de segmentos de retas de S com valor y máximo para x variando de $-\infty$ à $+\infty$ (figura 3.1). O nosso problema é:

PROBLEMA 6. *Dado um conjunto S de retas no plano, construa o envelope superior de S .*

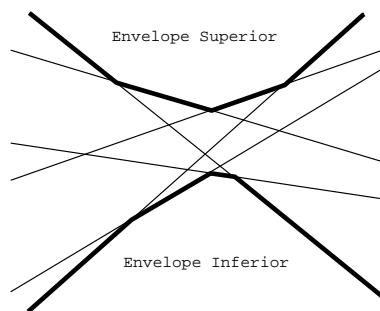


FIGURA 3.1. Envelope superior e envelope inferior de um conjunto de retas.

Nesta sessão, o nosso algoritmo fará a divisão de modo bastante simples, apenas dividimos S em S_1 e S_2 de mesmo tamanho (ou tamanhos diferindo de no máximo uma unidade, se $|S|$ for ímpar). A parte mais delicada do algoritmo consiste em combinar as duas soluções em uma solução unificada. Queremos resolver então o seguinte problema: dados dois envelopes superiores $U^1 = (U_1^1, \dots, U_{|U^1|}^1)$ e $U^2 = (U_1^2, \dots, U_{|U^2|}^2)$, obter o envelope superior $U = (U_1, \dots, U_{|U|})$ das retas de $U^1 \cup U^2$. Para combinarmos os dois envelopes superiores, usaremos uma técnica chamada de linha de varredura. Nesta técnica, vamos resolvendo o problema da esquerda para a direita.

Iniciamos comparando os coeficientes angulares das retas U_1^1 e U_1^2 , as retas que contém os segmentos mais a esquerda nos envelopes superiores U_1 e U_2 . A reta de menor coeficiente angular

dentre U_1^1 e U_1^2 será colocada na posição U_1 . Digamos que esta reta seja U_1^1 . Seguimos então descobrindo qual a primeira reta que intercepta U_1 , examinando apenas U_2^1 e U_1^2 , e colocamos esta reta na posição U_2 . Repetimos este procedimento até obtermos todo o envelope superior U . O pseudo-código do algoritmo está na figura 3.2.

Entrada:

U^1 : Vetor com retas formando um envelope superior, da esquerda para a direita.

U^2 : Idem, para outro conjunto de retas.

Saída:

U : Envelope superior de $U^1 \cup U^2$.

Observações:

$\angle(r)$: Coeficiente angular da reta r .

No caso de acessos além do limite dos vetores de entrada, considere que qualquer reta intercepta uma dada reta antes de uma reta inexistente.

CombinaEnvelopes(U^1, U^2)

$i \leftarrow i_1 \leftarrow i_2 \leftarrow 1$

Se $\angle(U^1[1]) < \angle(U^2[1])$

$U[1] \leftarrow U^1[1]$

$i_1 \leftarrow i_1 + 1$

Senão

$U[1] \leftarrow U^2[1]$

$i_2 \leftarrow i_2 + 1$

Enquanto $i_1 \leq |U^1|$ e $i_2 \leq |U^2|$

Se $U^1[i_1]$ intercepta $U[i]$ antes de $U^2[i_2]$

$i \leftarrow i + 1$

$U[i] \leftarrow U^1[i_1]$

$i_1 \leftarrow i_1 + 1$

Senão

$i \leftarrow i + 1$

$U[i] \leftarrow U^2[i_2]$

$i_2 \leftarrow i_2 + 1$

Retorne U

FIGURA 3.2. Fase de combinação do problema 6

Com este algoritmo de combinação de dois envelopes superiores concluído, é bastante simples escrever um algoritmo para resolver o problema original. Na primeira fase, dividimos S em S_1 e S_2 de mesmo tamanho (ou tamanhos diferindo de no máximo uma unidade, se $|S|$ for ímpar). Na segunda fase, resolvemos recursivamente o problema para os dois subconjuntos, a não ser que um dos conjuntos tenha apenas uma reta, quando sabemos que o envelope superior é a própria reta. Na terceira fase, combinamos as soluções com o algoritmo que acabamos de ver.

Vamos agora analisar a complexidade de tempo de nosso algoritmo. A primeira fase leva tempo constante e a terceira fase leva tempo linear. A complexidade da segunda fase é colocada na forma de recorrência

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

Para provarmos um limite superior para $T(n)$ por indução, precisamos ter uma estimativa de quanto vale $T(n)$. Vamos imaginar a execução do algoritmo como uma árvore como na figura 3.3. Cada vértice representa uma execução do procedimento e o número indicado nele representa o número de retas na entrada. Os dois filhos de um vértice correspondem às duas chamadas recursivas feitas a partir do vértice pai. O tempo gasto em todas as execuções com

uma reta na entrada, no total, é $O(n)$. O mesmo é válido para todas as execuções com 2 retas na entrada, e assim por diante. Como a altura da árvore é $O(\lg n)$, a soma das complexidades de tempo vale $O(n \lg n)$.

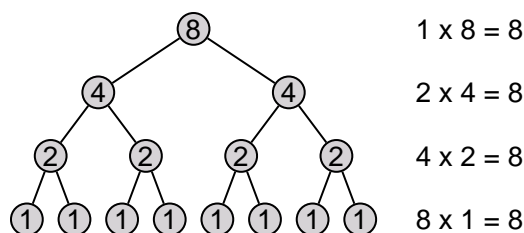


FIGURA 3.3. Árvore correspondente à execução do algoritmo de divisão e conquista em entrada de tamanho inicial 8.

Podemos então fazer provar o teorema abaixo usando indução:

TEOREMA 3.1. *A complexidade de tempo do algoritmo apresentado para determinar o envelope superior de um conjunto de n retas é $O(n \lg n)$.*

DEMONSTRAÇÃO. Como o algoritmo divide o problema em duas partes de aproximadamente o mesmo tamanho e as etapas de divisão e combinação levam tempo no máximo linear, temos a recorrência $T(n) = 2T(n/2) + n$. Por indução provamos que $T(n) \leq cn \lg n$:

$$\begin{aligned} T(n) &= 2T(n/2) + n \\ &\leq 2cn/2 \lg(n/2) + n \\ &= cn \lg(n/2) + n \\ &= cn \lg n - cn + n \\ &\leq cn \lg n. \end{aligned}$$

□

3.2. Par de Pontos Mais Próximos

PROBLEMA 7. *Dado um conjunto S de n pontos no plano, encontrar o par de pontos mais próximos.*

Um algoritmo trivial para este problema tem complexidade de tempo $\theta(n^2)$, simplesmente calculando a distância entre todos os pares de pontos e encontrando a distância mínima. Desejamos obter um algoritmo mais eficiente que este, para n grande. No problema anterior, dividimos os elementos da entrada em dois conjuntos quaisquer de tamanho aproximadamente iguais. Desta vez, seremos mais exigentes em nossa divisão. Vamos dividir S em dois conjuntos S_1 e S_2 por uma reta vertical r , de modo que S_1 e S_2 são aproximadamente do mesmo tamanho. Podemos fazer isso de modo simples se ordenarmos inicialmente os pontos de S segundo o eixo x . Note que esta ordenação só precisa ser feita uma vez no início do algoritmo.

Digamos que o par de pontos mais próximos de S_1 tenha distância d_1 e o par de pontos mais próximos de S_2 tenha distância d_2 . Como podemos obter o par de pontos mais próximos de $S_1 \cup S_2$? Certamente o par de pontos mais próximos tem distância menor ou igual a $\min(d_1, d_2)$. Mas é possível que o par que estamos procurando tenha um ponto em S_1 e outro em S_2 . Ainda assim, podemos descartar seguramente os pontos que distam mais que $\min(d_1, d_2)$ da reta vertical r que usamos para dividir os pontos (figura 3.4(a)). Vamos chamar de S' o conjunto de pontos que distam no máximo $\min(d_1, d_2)$ de r . Nosso algoritmo precisa apenas calcular o par de pontos mais próximo em S' e comparar sua distância com $\min(d_1, d_2)$, escolhendo a menor. Na maioria das situações, só isto já reduziria bastante o tempo de processamento, porém, no pior caso, pode ser que não descartemos nenhum ponto, portanto ainda teríamos de calcular $O(n^2)$ distâncias.

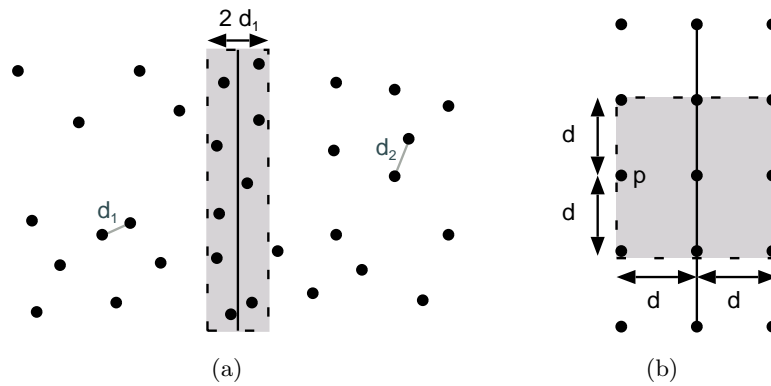


FIGURA 3.4. (a) Execução do algoritmo para encontrar o par de pontos mais próximos. (b) Número máximo de pontos que podem estar contidos no quadrado.

Para todo ponto p em S' , podemos nos limitar a calcular a distância entre p e os pontos de S' cuja diferença de coordenada y seja menor que $\min(d_1, d_2)$. Como o número desses pontos é no máximo 8, garantimos que só precisamos fazer um número linear de comparações nessa fase. Para justificarmos que o número desses pontos é no máximo 8, note que, para cada ponto p , os pontos de S' que distam até $\min(d_1, d_2)$ de p , estão dentro de um quadrado de lado $2\min(d_1, d_2)$, e não há dois pontos na mesma metade desse quadrado que distem menos de $\min(d_1, d_2)$ (figura 3.4(b)). Porém, para fazermos selecionarmos estes pontos, devemos percorrer os pontos de S' de cima para baixo, sendo necessário ordenar os pontos segundo o eixo y .

Para analisarmos a complexidade de tempo deste algoritmo, construímos a recorrência abaixo, lembrando que a complexidade de tempo para ordenar n elementos é $\theta(n \lg n)$:

$$T(n) = 2T(n/2) + n \lg n.$$

É possível provar por indução que $T(n) = \theta(n \lg^2 n)$, porém não vamos fazer esta prova, já que reduziremos ainda mais a complexidade do nosso algoritmo. Para isto, basta fazermos a ordenação dos pontos segundo o eixo y apenas uma vez, e sempre passarmos o conjunto de pontos como parâmetro ordenado pelo eixo y . Note que, na fase de divisão, não precisamos trabalhar com os pontos ordenados segundo o eixo x , mas apenas ter acesso a esta ordenação de modo a encontrar rapidamente o elemento com coordenada x mediana (de fato, é possível encontrar a mediana em tempo linear sem usar ordenação, como veremos na sessão 4.2, mas neste caso é mais eficiente na prática ordenarmos os pontos).

Assim, o nosso algoritmo começa ordenando os pontos segundo o eixo x e segundo o eixo y separadamente. Dividimos então os pontos, ordenados pelo eixo y em dois conjuntos separados por uma reta vertical. Calculamos recursivamente o par de pontos mais próximos nestes dos sub-conjuntos, sem refazermos qualquer ordenação, pois os pontos já estão ordenados segundo o eixo y e temos acesso a sua ordem segundo o eixo x . Computamos então o conjunto de pontos S' , próximos a reta vertical r , e examinamos as distâncias entre os pontos verticalmente próximos de S' , de cima para baixo, bastando examinarmos distâncias entre pontos à esquerda de r e pontos à direita de r . Retornamos então a distância mínima, dentre as distâncias entre os pontos de S' , d_1 e d_2 . O pseudo-código deste algoritmo está na figura 3.5. Alguns detalhes do algoritmo estão um pouco diferentes no pseudo-código, de modo a melhorar ainda mais a performance.

A complexidade de tempo do nosso algoritmo é $O(n \lg n)$, pois as ordenações iniciais levam tempo $O(n \lg n)$ e o restante obedece a já conhecida recorrência

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

Entrada: S : Conjunto de pontos no plano**Saída:** (p, p') : Par de pontos mais próximos**PontosMaisPróximos(S)** $S_x \leftarrow$ ordenação de S segundo o eixo x $S_y \leftarrow$ ordenação de S segundo o eixo y Retorne $\text{PMPOrdenado}(S_x, 1, |S|, S_y)$ **PMPOrdenado($S_x, inicio, fim, S_y$)**Se $fim - inicio \leq 4$

Retorne a solução do problema obtida comparando todas as distâncias

 $meio \leftarrow \lfloor (inicio + fim)/2 \rfloor$ Para i de 1 até $|S_y|$ Se $S_y[i].x \leq S_x[meio]$, então acrescenta $S_y[i]$ ao final de S_1 Senão, acrescenta $S_y[i]$ ao final de S_2 $(p_1, p'_1) \leftarrow \text{PMPOrdenado}(S_x, inicio, meio, S_1)$ $(p_2, p'_2) \leftarrow \text{PMPOrdenado}(S_x, meio + 1, fim, S_2)$ Se $|p_1 - p'_1| < |p_2 - p'_2|$, então $p \leftarrow p_1$ e $p' \leftarrow p'_1$ Senão, $p \leftarrow p_2$ e $p' \leftarrow p'_2$ $d \leftarrow |p - p'|$ Para i de 1 até $|S_1|$ Se $S_x[meio + 1].x - S_1[i].x < d$ Acrescenta $S_1[i]$ ao final de S'_1 Para i de 1 até $|S_2|$ Se $S_1[i].x - S_x[meio].x < d$ Acrescenta $S_2[i]$ ao final de S'_2 $j_2 \leftarrow 1$ Para i_1 de 1 até $|S'_1|$ Enquanto $S'_1[i_1].y - S'_2[j_2].y > d$ $j_2 \leftarrow j_2 + 1$ $i_2 \leftarrow j_2$ Enquanto $S'_2[i_2].y - S'_1[i_1].y < d$ Se $|S'_2[i_2] - S'_1[i_1]| < d$ $p \leftarrow S'_1[i_1]$ e $p' \leftarrow S'_2[i_2]$ $d \leftarrow |p - p'|$ $i_2 \leftarrow i_2 + 1$ Se $i_2 > |S'_2|$, então sai do 'enquanto'Retorne (p, p')

FIGURA 3.5. Solução do Problema 7

3.3. Conjunto Independente de Peso Máximo em Árvores

Um conjunto independente em um grafo, é um subconjunto de seus vértices que não contém nenhum par de vértices que sejam adjacentes. Chama-se de conjunto independente máximo, o maior conjunto independente do grafo (figura 3.6(a)). Na versão com pesos nos vértices, deseja-se maximizar a soma dos pesos dos vértices do conjunto. Este problema é extremamente complexo de ser resolvido, estando na categoria de problemas *NP-difíceis*, como veremos no capítulo 10. Porém, se nos restringirmos a árvores (grafos sem ciclos), podemos resolver o problema eficientemente usando divisão e conquista.

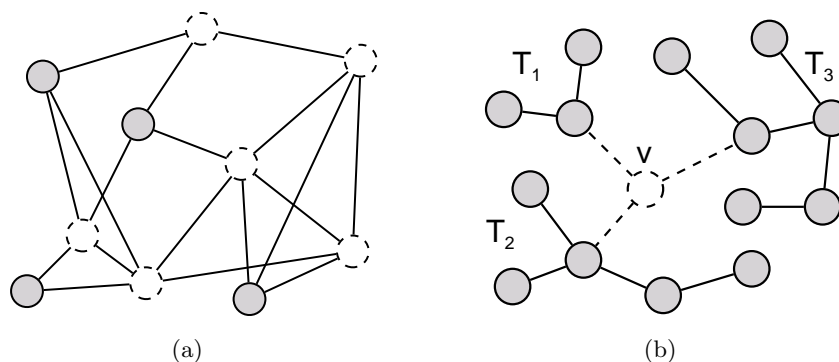


FIGURA 3.6. (a) Conjunto independente máximo de uma árvore sem pesos.
 (b) Árvore dividida em três sub-árvores pela remoção do vértice v .

PROBLEMA 8. *Dado uma árvore T , com pesos nos vértices, encontrar um conjunto independente de peso máximo de T .*

Nos outros problemas dessa sessão, nos preocupamos em fazer a divisão de modo balanceado, ou seja, queríamos obter sub-problemas de aproximadamente o mesmo tamanho. Neste caso, entretanto, veremos que isto não é necessário. Vamos começar escolhendo um vértice v da árvore T . Com este vértice, é natural dividir a árvore em algumas sub-árvores (figura 3.6(b), pois a remoção de v vai tornar a árvore desconexa (a não ser que v seja uma folha, mas, se for, também não há problema nenhum). Como podemos usar a solução dos problemas para as sub-árvores de modo a obter uma solução para o problema maior?

Pensando um pouco sobre isso, você vai notar que não há qualquer maneira óbvia de fazê-lo, pois caso algum vértice adjacente a v em T esteja no conjunto independente máximo de uma das sub-árvores, não será possível acrescentar v ao novo conjunto independente, aproveitando as soluções dos sub-problemas. Para resolver isto, vamos complicar um pouco nosso problema.

Nosso novo problema é: dados uma árvore T , com pesos nos vértices, e um vértice v , calcular: (i) um conjunto independente de maior peso dentre os conjuntos independentes que não contém v ; (ii) um conjunto independente de peso máximo. Com isto, podemos descrever nosso algoritmo de divisão e conquista.

Na primeira iteração, como não é fornecido nenhum vértice v , iniciamos escolhendo um vértice v qualquer. Para cada sub-árvore T_i obtida pela remoção de v , chamamos de v_i o vértice de T_i adjacente à v . Calculamos recursivamente os conjuntos independentes máximos de cada sub-árvore T_i , podendo conter e sem poder conter o vértice v_i . O conjunto independente máximo C_1 de T , com a restrição de não conter v , é, claramente, a união dos conjuntos independentes máximos das sub-árvores obtidas pela remoção de v . Para calcularmos o conjunto independente máximo real de T , construímos um outro conjunto independente C_2 . O conjunto independente C_2 é obtido pela união do vértice v aos conjuntos independentes máximos das sub-árvores T_i , que não contém v_i . O conjunto independente máximo de T é, então, o conjunto independente de maior peso dentre C_1 e C_2 . O pseudo-código deste algoritmo está na figura 3.7.

Provar que a complexidade de tempo do algoritmo é linear no número de vértices é simples e fica como exercício.

3.4. Multiplicação de Matrizes: Algoritmo de Strassen

PROBLEMA 9. *Dadas duas matrizes $n \times n$, A e B , obter a matriz $C = A \cdot B$.*

Uma solução bastante simples é usar a definição de produto de matrizes, que é

$$C_{ij} = \sum_{k=1}^n A_{ik} B_{kj}.$$

Entrada: T : Árvore com pesos nos vértices. v : Vértice de T , inicialmente qualquer vértice.Saída: (C^1, C^2) , onde: C^1 : Conjunto independente que não contém v de peso máximo C^2 : Conjunto independente de peso máximo de T ConjuntoIndependente(T, v)Para cada sub-árvore T_i obtida pela remoção de v de T $v_i \leftarrow$ vértice de T_i adjacente a v em T $(C_i^1, C_i^2) \leftarrow$ ConjuntoIndependente(T_i, v_i) $C^1 \leftarrow C^1 \cup C_i^2$ $C^2 \leftarrow C^1 \cup C_i^1$ $C^2 \leftarrow C^1 \cup \{v\}$ Se $\text{peso}(C^2) < \text{peso}(C^1)$ $C^2 \leftarrow C^1$ Retorne (C^1, C^2)

FIGURA 3.7. Solução do problema 8

Este algoritmo tem complexidade de tempo $O(n^3)$, pois para calcularmos cada elemento na matriz C , fazemos um número linear de operações elementares. Como podemos usar divisão e conquista neste problema, ou seja, decompor o problema em sub-problemas menores? Primeiro vamos simplificar um pouco o problema, nos restringindo a matrizes onde n é uma potência de 2. Não perdemos muito com isto, pois caso a largura de nossa matriz não seja uma potência de 2, podemos completá-la com elementos nulos.

Sabemos que o produto de duas matrizes 2×2 é dado por

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} = \begin{pmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{pmatrix}.$$

Podemos dividir cada uma das nossas matrizes $n \times n$, A e B , em quatro sub-matrizes $n/2 \times n/2$, pois consideramos que n é potência de 2. Usamos então a fórmula para multiplicação de matrizes 2×2 . Assim, teremos que fazer 8 multiplicações de matrizes $n/2 \times n/2$. Estas multiplicações são resolvidas recursivamente. Note que este algoritmo é bem diferente dos outros algoritmos de divisão e conquista que vimos antes. Não estamos apenas dividindo a entrada em conjuntos disjuntos, e resolvendo recursivamente o problema nesses conjuntos. Agora, dividimos cada uma das matrizes da entrada em 4 partes e criamos 8 sub-problemas combinando estas partes. Deste modo, a fase de divisão, onde definimos os sub-problemas a serem resolvidos, tornou-se bem mais elaborada.

Para analisarmos a complexidade de tempo deste algoritmo, vamos apenas contar o número de multiplicações elementares realizadas, já que o número de adições e outras operações é uma constante vezes o número de multiplicações. Contamos exatamente este número com a recorrência

$$T(n) = \begin{cases} 8T(n/2) & , \text{ se } n > 2 \\ 8 & , \text{ se } n = 2 \end{cases}.$$

É fácil notar que $T(n) = n^3$, portanto não ganhamos absolutamente nada com nosso algoritmo de divisão e conquista. Porém, nosso algoritmo agora é fortemente baseado em uma operação bastante simples, a multiplicação de matrizes 2×2 . Se conseguirmos descobrir uma maneira mais eficiente de multiplicarmos estas matrizes, podemos melhorar nosso algoritmo

imediatamente. Não é nem um pouco trivial, multiplicar duas matrizes 2×2 com menos de 8 multiplicações elementares, mas mostraremos aqui como fazê-lo. Considere as variáveis abaixo:

$$\begin{aligned} m_1 &= (a_{21} + a_{22} - a_{11})(b_{22} - b_{12} + b_{11}), \\ m_2 &= a_{11}b_{11}, \\ m_3 &= a_{12}b_{21}, \\ m_4 &= (a_{11} - a_{21})(b_{22} - b_{12}), \\ m_5 &= (a_{21} + a_{22})(b_{12} - b_{11}), \\ m_6 &= (a_{12} - a_{21} + a_{11} - a_{22})b_{22}, \\ m_7 &= a_{22}(b_{11} + b_{22} - b_{12} - b_{21}). \end{aligned}$$

É possível, embora um pouco trabalhoso, verificar que

$$AB = \begin{pmatrix} m_2 + m_3 & m_1 + m_2 + m_5 + m_6 \\ m_1 + m_2 + m_4 - m_7 & m_1 + m_2 + m_4 + m_5 \end{pmatrix}.$$

De fato, multiplicar matrizes 2×2 usando este método é extremamente ineficiente, pois no lugar de 4 adições e 8 multiplicações, realizamos 24 adições ou subtrações e 7 multiplicações (é possível reduzir o número de adições e subtrações para 15 usando variáveis adicionais). Porém, se as adições puderem ser realizadas muito, muito, muito mais rápido que as multiplicações, pode valer a pena. Este é o caso das operações com matrizes grandes. Como este método não se baseia na comutatividade da multiplicação, podemos considerar que os elementos são matrizes, e não números reais. Assim, se usarmos este método para escolher que sub-problemas desejamos resolver, obtemos a recorrência

$$T(n) = 7T(n/2).$$

Para resolvermos esta recorrência, vamos considerar $T(1) = 1$. Assim, obtemos $T(2) = 7$, $T(4) = 7^2$ etc. É fácil perceber que $T(2^n) = 7^n$. Substituindo n por $\lg n$, temos

$$T(n) = 7^{\lg n} = 7^{\log_7 n / \log_7 2} = n^{1/\log_7 2} = n^{\lg 7} \approx n^{2,80735}.$$

TEOREMA 3.2. *O algoritmo de Strassen calcula o produto de duas matrizes $n \times n$ em tempo $O(n^2,81)$.*

Desta forma, conseguimos reduzir a complexidade de tempo de $\theta(n^3)$ para $\theta(n^{2,81})$. Claro que, com isso, aumentamos a constante oculta pela notação O , portanto este algoritmo só é mais rápido na prática para multiplicar matrizes realmente muito grandes. Assim, quando n é menor que um certo valor (que pode ser determinado experimentalmente), é preferível chamar o algoritmo cúbico de multiplicação, e não continuar recursivamente.

3.5. Resumo e Observações Finais

Apresentamos neste capítulo a técnica de divisão e conquista, que se baseia em decompor um problema em sub-problemas menores, e resolvê-los recursivamente, combinando suas soluções depois. Apresentamos aqui quatro problemas: envelope superior de retas, par de pontos mais próximos, conjunto independente máximo em árvores e multiplicação de matrizes. Os algoritmos de divisão e conquista têm três fases: dividir, conquistar e combinar.

Na primeira fase, a divisão, o problema é decomposto em dois (ou mais) sub-problemas. No problema ‘envelope superior’, simplesmente dividimos as retas em dois conjuntos de mesmo tamanho. No problema ‘par de pontos mais próximos’, dividimos os pontos em dois conjuntos do mesmo tamanho usando uma reta vertical. No problema ‘conjunto independente máximo’, particionamos a árvore, removendo um vértice a cada iteração. No problema ‘multiplicação de matrizes’, criamos sete sub-problemas, combinando partições das duas matrizes originais e suas somas.

Na segunda fase, a conquista, resolvemos os sub-problemas, recursivamente, usando o mesmo procedimento de divisão e conquista, até que o tamanho do problema seja tão pequeno que sua solução seja trivial ou possa ser feita mais rapidamente usando algoritmos mais simples.

Na terceira fase, a combinação das soluções, também chamada de casamento, temos que unir as soluções dos problemas menores para obtermos uma solução unificada. No caso do ‘envelope superior’, usamos o paradigma de linha de varredura para computar esta combinação da esquerda para a direita. No problema ‘par de pontos mais próximos’, tivemos que usar uma técnica bem mais sofisticada, para calcular apenas um número linear de distâncias adicionais. No ‘conjunto independente máximo’, aumentamos o problema para retornar também o conjunto independente máximo sem um elemento, de modo que pudéssemos fazer a combinação. No problema ‘multiplicação de matrizes’, unimos as soluções usando equações nada triviais.

A técnica de busca binária, examinada no capítulo 2, é um caso particular do paradigma de divisão e conquista onde a divisão é feita examinando apenas um número constante de elementos e escolhendo um único sub-problema para resolver. A técnica de simplificação, que será estudada no capítulo 4, também é um caso particular da divisão e conquista, onde é resolvido apenas um sub-problema.

Uma variação do paradigma de divisão e conquista que não vimos aqui consiste em combinar antes de conquistar (ou casar antes de conquistar). Nesta variação, primeiro analisamos como as soluções serão combinadas, para depois seguirmos na conquista, nos beneficiando da informação obtida na combinação. Esta variação é útil no exercício 3.6.

Um artifício que também não comentamos neste capítulo, mas é essencial para a eficiência de alguns algoritmos de divisão e conquista, é chamado de memorização. Em alguns casos, o nosso algoritmo pode chamar duas vezes o procedimento para encontrar a solução exatamente do mesmo problema. Neste caso, devemos consultar uma tabela e recuperarmos a resposta que anotamos na tabela, sem perder tempo fazendo duas vezes o mesmo cálculo. O algoritmo do exercício 3.4 tem complexidade exponencial sem memorização, mas complexidade linear com memorização.

Uma alternativa à técnica de divisão e conquista e memorização é chamada de programação dinâmica (capítulo 5). Na técnica de programação dinâmica, no lugar de resolvermos um problema maior dividindo-o em problemas menores, resolvemos primeiro problemas menores e seguimos combinando suas soluções até chegar na solução do problema maior que desejamos resolver.

Exercícios

- 3.1) A recorrência $T(n) = T(\alpha n) + T((1 - \alpha)n) + n$, com α constante entre 0 e 1, é obtida no cálculo de complexidade caso a divisão da entrada em duas partes não seja simétrica. Prove que esta recorrência também satisfaz $T(n) = O(n \lg n)$.
- 3.2) Escreva um algoritmo que calcule o elemento máximo e o elemento mínimo de um conjunto com n elementos, usando apenas $3 \lceil n/2 \rceil - 2$ comparações.
- 3.3) Escreva um algoritmo para ordenar um vetor com n elementos em tempo $O(n \lg n)$ usando divisão e conquista.
- 3.4) Os números de Fibonacci F_i são definidos recursivamente pela recorrência $F_i = F_{i-1} + F_{i-2}$, com $F_0 = 0$ e $F_1 = 1$. Escreva um algoritmo recursivo para calcular o F_i . Use o recurso de memorização para tornar a complexidade de seu algoritmo linear em i .
- 3.5) Dado um conjunto de n pontos no plano, escreva um algoritmo para determinar seu fecho convexo em tempo $O(n \lg n)$, usando divisão e conquista. Prove que o algoritmo está correto e analise sua complexidade de tempo.
- 3.6) Dado um conjunto de n pontos no plano, escreva um algoritmo para determinar seu fecho convexo em tempo $O(n \lg h)$, onde h é o número de vértices no fecho convexo. Seu algoritmo pode usar livremente uma função que, dado um conjunto de m pontos no plano e uma reta vertical r , retorna as arestas do fecho convexo que interceptam a reta r . Prove que o algoritmo está correto e analise sua complexidade de tempo.

- 3.7) Uma triangulação de um conjunto S de pontos no plano é uma subdivisão de seu fecho convexo em triângulos disjuntos (exceto em seus bordos), onde os vértices dos triângulos são exatamente os pontos de S (figura 3.8(a)). Escreva um algoritmo para computar uma triangulação de um conjunto de pontos no plano.
- *3.8) Uma triangulação de Delaunay é uma triangulação que satisfaz a propriedade que os círculos circunscritos aos triângulos da triangulação não contém nenhum ponto em seus interiores (figura 3.8(b)). Outra definição é que uma aresta pertence a triangulação de Delaunay se e só se existe círculo com os dois pontos da aresta no seu bordo e nenhum ponto em seu interior. Escreva um algoritmo baseado em divisão e conquista que, dado um conjunto S de pontos no plano, compute sua triangulação de Delaunay em tempo $O(|S| \lg |S|)$.

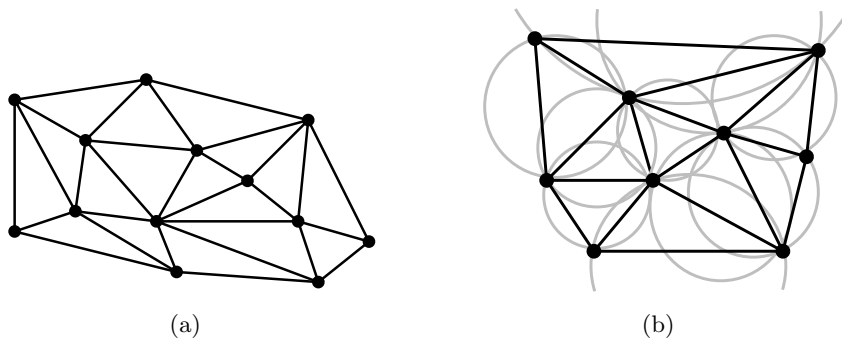


FIGURA 3.8. (a) Triangulação de um conjunto de pontos. (b) Triangulação de Delaunay de um conjunto de pontos.

- *3.9) O fecho convexo de um conjunto de pontos no espaço é o menor poliedro convexo que contém todos estes pontos. Dado um conjunto de n pontos no espaço tridimensional, escreva um algoritmo para determinar seu fecho convexo em tempo $O(n \lg n)$. Prove que o algoritmo está correto e analise sua complexidade de tempo.