MT404/MS993

Métodos computacionais de álgebra linear

Terceiro Projeto – 2023

Algoritmos subcúbicos para multiplicação matricial

Resumo

Nesta terceira atividades projeto, vamos introduzir as principais noções sobre complexidade algorítmica e apresentar os algoritmos de Strassen para multiplicação matricial. Faremos diversos "experimentos" computacionais com esses algoritmos.

1 Preliminares

A noção fundamental que exploraremos nesta atividade é a de complexidade algorítmica. Tentaremos manter este texto o mais auto-contido possível, mas para detalhes extras e pré-requisitos, recomenda-se [1], uma das obras básicas da literatura do curso. Este é um assunto vasto e aqui o exploraremos apenas superficialmente. Grosso modo, complexidade algorítmica é uma medida do "custo" em tempo, energia, ou qualquer outra quantidade de interesse, necessário para que um algoritmo seja executado completamente. Vamos tomar como exemplo a operação que nos interessa agora, a multiplicação matricial $AB = C$. Se A e B são, respectivamente, matrizes $m \times p$ e $p \times n$, a matriz produto será uma matriz $m \times n$. Da definição do produto matricial, sabemos que as entradas de C, em termos das entras de A e B, são dadas por

$$c_{ij} = \sum_{k=1}^{p} a_{ik} b_{kj}, \tag{1}$$

com $1 \leq i \leq m$ e $1 \leq j \leq n$. Podemos facilmente propor um algoritmo tal que, dada duas matrizes A e B, calcule seu produto $C = AB$. Basta, por exemplo, calcular a soma (1) para cada uma das entradas de C. Serão obviamente mn somas dessas. Para os nossos propósitos, contaremos apenas as operações aritméticas elementares, aquelas realizadas com as entradas das matrizes dadas A e B, que podem estar em formato floating-point ou de inteiros. Vamos ignorar todos os outros "custos" associados ao algoritmo, como por exemplo o custo de se montar e executar instruções em loop, o custo associado a registrar e recuperar valores de variáveis na memória, etc. Esta é uma aproximação comum na análise de complexidade de algoritmos numéricos, mas que obviamente não é adequada para aplicações computacionais não numéricas como, por exemplo, organização de listas, etc. Cada operação do tipo (1) envolve p produtos e $p-1$ adições elementares. Assim, temos que o "custo aritmético"[1] para obtermos o produto das matrizes A e B consistirá em mnp produtos e $mn(p-1)$ adições elementares. Para efeitos desta atividade, essa será a complexidade do algoritmo descrito por (1). Podemos sempre manter as operações de multiplicação e de adição separadas, explicitando o número de cada uma delas, ou o que é mais frequente, vamos definir o custo em termos das operações de adição, definindo-se ω como a razão entre o custo de um produto e uma adição elementares. Neste caso, o custo total, em termos do custo de adições elementares, será $(1+\omega)mnp-mn$. O valor de ω para aplicações práticas depende, em última instância, do processador utilizado. Esse tipo de parâmetro pode ser obtido na documentação dos processadores de interesse. Tipicamente, as instruções para soma e multiplicação de números em formato floating-point são, respectivamente, `FADD` e `FMUL`, e seus principais parâmetros de execução (latency, throughput, etc) são sempre apresentados em números de ciclos do clock do processador. Nosso parâmero ω é a razão entre o custo de uma instrução `FMUL` e uma `FADD` e, obviamente, deve ser entendido estatisticamente, *i.e.*, em média. No passado não muito distante, as instruções `FMUL` eram bastante mais custosas que as `FADD` e não era raro encontrar valores de ω da ordem de 10. No caso do Z80, processador extremamente comum no inicio dos anos 80, nem sequer havia uma instrução do tipo `FMUL`, e a multiplicação era implementada via software, podendo implicar, na prática, em ω superiores a 50. Graças a avanços consideráveis nos circuitos dos processadores, hoje em dia é comum termos ω da ordem 1, o que na prática nos leva a considerar equivalentes, do ponto

[1] É(ra) comum empregar a expressão FLOP (*float-point operation*) para designar esse custo. A unidade FLOPS, *float-point operation per second*, ainda é comum para designar o desempenho de CPUS.

de vista de custo, somas e multiplicações elementares. Uma ótima referência on-line sobre processadores e o desempenho de suas principais instruções é [2], procurem por *Instruction Tables*.

Vamos restringir nossas análises, por simplicidade, ao produto de matrizes quadradas $n \times n$. Nesse caso, a complexidade da multiplicação matricial (1) será $(1+\omega)n^3 - n^2$. Nosso interesse maior será sempre no comportamento assintótico da complexidade algorítmica. Para este fim, vamos introduzir a notação $O(g(x))$, chamada *big-O*, para denotar comportamentos assintóticos de funções. Dizemos que $f(x) = O(g(x))$ se existirem uma constante positiva c e um x_0 tal que

$$0 \leq f(x) \leq cg(x), \quad \text{para todo } x > x_0. \tag{2}$$

Estamos interessados basicamente em comparar funções não negativas. Desta definição, é evidente que $f(x) = O(g(x))$ implica que $g(x)$ é um limite superior para $f(x)$ para valores de x grandes, um limite superior para o crescimento de $f(x)$. Rigorosamente, $O(g(x))$ define uma classe de equivalência, e portanto, seria mais adequado afirmar que $f(x) \in O(g(x))$, e não $f(x) = O(g(x))$. Vamos ignorar estas sutilezas aqui, não serão relevantes para nossos propósitos. Como sempre, exemplos serão uteis para fixarmos os conceitos. A complexidade da multiplicação de matrizes quadradas é do tipo

$$(1+\omega)n^3 - n^2 = O(n^3), \tag{3}$$

pois é fácil ver que

$$0 \leq (1+\omega)n^3 - n^2 \leq (1+\omega)n^3 \tag{4}$$

para todo n. Tecnicamente, com a nossa definição, temos que toda a função que é $O(n^k)$ será também $O(n^{k+1})$, para todo k inteiro não negativo, quer dizer, o conjunto das funções $O(n^k)$ está contido no conjunto das funções $O(n^{k+1})$. Porém, para os nossos interesses, vamos sempre considerar o **menor** desses conjuntos, quer dizer, nosso interesse é no limite superior mais estrito possível. Muitas vezes se utiliza a notação $\Theta(g(x))$, definida de maneira análoga, mas se existirem duas constantes positivas c_1 e c_2 e um x_0 tal que

$$0 \leq c_1 g(x) \leq f(x) \leq c_2 g(x), \quad \text{para todo } x > x_0. \tag{5}$$

Aqui vamos adotar a notação $O(g(x))$, sempre no sentido do limite superior estrito.

O nosso algoritmo de multiplicação tem complexidade $O(n^3)$ e isso nos dá informação sobre o custo de sua execução. Por exemplo, podemos estimar que a multiplicação de duas matrizes 100×100 será 10^3 vezes mais custosa que o caso da multiplicação de duas matrizes 10×10. Esse é um típico algoritmo polinomial, pois sua complexidade é um polinômio no parâmetro associado à quantidade total de dados, que aqui obviamente é o tamanho n da matriz quadrada. Algoritmos polinomiais são tipicamente bons. Há, obviamente, algoritmos "ruins" do ponto de vista de complexidade, e convém dar um exemplo ilustrativo. Considerem o seguinte problema.

Problema (Problema da soma de subconjuntos)**.** *Seja $S = \{s_1, s_2, s_3, \ldots, s_n\}$ um conjunto de n inteiros $s_k \in \mathbb{Z}$. Determine se há algum subconjunto de S cuja a soma seja M.*

Este exemplo, um clássico na teoria de complexidade, vejam por exemplo a seção 35.5 de [1], nos permite ilustrar vários pontos sutis. Vamos começar **supondo** que tenhamos uma resposta ao problema, quer dizer, que sabemos que há um subconjunto e o temos. Qual é o custo para verificarmos que, de fato, temos a resposta? Bem, é o custo associado com somar os inteiros do subconjunto e verificar que o resultado é M. No pior dos casos, teremos que o subconjunto seria o próprio conjunto inicial S e, nesse caso, teríamos que fazer $n-1$ somas. Portanto, temos que, no pior dos casos, a complexidade da **verificação** da solução é $O(n)$ e, portanto, polinomial em n. Porém, e se de fato não sabemos se há ou não tal subconjunto e temos que procurá-lo? Qual seria a complexidade desta operação? Bem, basicamente teríamos que fazer uma busca exaustiva sobre todos os possíveis subconjuntos de S, realizar as somas, e verificar se o resultado é M. Quantos subconjuntos de k inteiros podemos formar a partir da nossa lista de n inteiros? Sabemos a resposta, ela será dada pelo número de combinações possíveis

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}. \tag{6}$$

Para cada subconjunto (combinação) de k inteiros, teremos que fazer $k-1$ somas para verificar se o resultado é M. Portanto, o número total de operações para "varrer" todas as possibilidades será[2]

$$\sum_{k=1}^{n}(k-1)\binom{n}{k} = \frac{n!}{k!(n-k)!} = \left(\frac{n}{2}-1\right)2^n + 1, \tag{7}$$

i.e., esta estratégia tem uma complexidade $O(n2^n)$. Os problemas cuja solução exigem algoritmos de complexidade polinomial, como o nosso problema de verificar se uma solução do problema é de fato correta, são ditos problemas da classe P, de polinomial. O problema de calcular o produto de duas matrizes A e B também é da classe P. Notem que o problema da soma de subconjuntos é curioso, pois a tarefa de verificar uma solução é de complexidade polinomial, mas procurar essas soluções com a nossa estratégia acima é de complexidade muito maior que polinomial. Denota-se por NP (*nondeterministic polynomial*) os problemas cujas soluções podem ser verificadas por algoritmos polinomiais. A pergunta que dá origem ao talvez mais famoso problema da computação é se os problemas do tipo NP possuiriam algoritmos polinomiais, ainda desconhecidos, para sua solução, ou de maneira informal, se P = NP. Para mais detalhes, ver o Capítulo 34 de [1]. Não nos aventuraremos mais nesse campo riquíssimo, sutil e escorregadio (para mim!).

2 Multiplicação matricial

Vamos retornar a multiplicação matricial (1), restringindo-nos as matrizes quadradas $n \times n$. Vimos que o algoritmo baseado na definição (1) requer $(1+\omega)n^3 - n^2$ operações (adições) elementares. Porém, fica a pergunta: poderíamos fazer melhor, com um menor número de operações? A resposta é sim. Vamos explorar uma ideia que está proposta como o exercício 42 da seção 4.6, Volume 2, do sempre inspirador Knuth [3]. A questão envolve o produto de dois números complexos

$$(a+bi)(c+di) = (ac-bd) + (ad+bc)i. \tag{8}$$

O produto de dois números complexos, vistos como pares de números reais, envolve quatro produtos e duas somas reais. A pergunta do exercício é: poderíamos fazer esse produto usando apenas **três** produtos reais? A resposta

[2]Este resultado pode ser provado explorando a definição dos binômios de Newton: $(x+y)^n = \sum_{k=0}^{n}\binom{n}{k}x^{n-k}y^k$ e sua derivada, e tomando-se $x = y = 1$. Provem!

é sim, e envolve os seguintes produtos

$$m_1 = ac, \quad m_2 = bd, \quad m_3 = (a+b)(c+d), \tag{9}$$

de onde temos

$$(a+bi)(c+di) = \overbrace{(ac-bd)}^{m_1-m_2} + \underbrace{(ad+bc)}_{m_3-m_1-m_2} i. \tag{10}$$

Vemos que é possível fazer o produto de dois números complexos a partir de 3 multiplicações e 5 adições reais, quer dizer, para ter menos multiplicações, pagamos o preço de termos mais adições. Estratégias deste tipo podem resultar num custo total menor se ω for grande o suficiente.

Este exemplo nos inspira a tentar aplicar construções semelhantes em outros casos. Vamos considerar o problema do cálculo do produto escalar de dois vetores $\vec{V}, \vec{W} \in \mathbb{R}^n$,

$$\vec{V} \cdot \vec{W} = \sum_{k=1}^{n} v_k w_k. \tag{11}$$

Esta operação, como definida, requer n produtos e $n-1$ adições para o cálculo do produto escalar. Poderíamos fazer de outra forma? A resposta, como já devem ter adivinhado, é sim. Vamos supor momentaneamente que n seja par. Teremos a seguinte identidade

$$\sum_{k=1}^{n} v_k w_k = \sum_{k=1}^{\frac{n}{2}} \left(v_{2k-1} + w_{2k}\right)\left(v_{2k} + w_{2k-1}\right) - \sum_{k=1}^{\frac{n}{2}} v_{2k-1} v_{2k} - \sum_{k=1}^{\frac{n}{2}} w_{2k-1} w_{2k}. \tag{12}$$

Notem a estrutura desse produto. Tomamos a soma das entradas ímpares de $\vec{V}$ com as pares de $\vec{W}$ e calculamos o produto com a soma das entradas pares de $\vec{V}$ com as ímpares de $\vec{W}$. Obviamente, aparecerão termos espúrios no produto, e os subtraímos para obter o produto escalar original desejado. Uma possível vantagem é que agora temos que calcular $\frac{n}{2}$ produtos no lugar de n. Será que houve algum ganho? Vamos conferir. O primeiro somatório envolve $\frac{3n}{2} - 1$ adições e $\frac{n}{2}$ produtos. O segundo e o terceiro somatórios envolvem, cada um deles, $\frac{n}{2}$ produtos e $\frac{n}{2} - 1$ adições. O total de operações será $(3\omega + 5)\frac{n}{2} - 1$, o que, desgraçadamente, não representa ganho nenhum em relação ao calculo usual. Porém, a mesma ideia pode ser vantajosa para matrizes, como veremos a seguir. Notem que a hipótese de n par, para efeitos de complexidade, não implica em perda de generalidade. Se n for impar, basta considerar $n-1$ no algoritmo (12) e adicionar o produto extra $v_n w_n$, o que implicaria em uma soma e um produto a mais para termos o produto escalar desejado.

O algoritmo de Winograd para multiplicação matricial explora a identidade (12) para matrizes. A motivação principal é que as componentes c_{ij} do produto matricial (1) são, de fato, produtos escalares dos vetores linhas e colunas de A e B, respectivamente. Teremos

$$c_{ij} = \sum_{k=1}^{\frac{n}{2}} \left(a_{i,2k-1} + b_{2k,j}\right)\left(a_{i,2k} + b_{2k-1,j}\right) - \sum_{k=1}^{\frac{n}{2}} a_{i,2k-1} a_{i,2k} - \sum_{k=1}^{\frac{n}{2}} b_{2k-1,j} b_{2k,j}. \quad (13)$$

Por que o algoritmo de Winograd (13) pode representar, de fato, um ganho, se o produto escalar (12) é pior que o usual? Porque os dois últimos somatórios em (13) são calculados separadamente e uma única vez para as linhas de A e para as colunas de B. Vejamos. O primeiro somatório envolve, como no caso anterior, $\frac{3n}{2} - 1$ adições e $\frac{n}{2}$ produtos. Como são n^2 entradas de C, vamos precisar de $\frac{3+\omega}{2} n^3 - n^2$ operações para calcular o primeiro somatório para toda a matriz C. Cada um dos outros dois somatório requer apenas $\frac{1+\omega}{2} n^2 - n$ operações para serem calculados. Lembrando que há ainda duas subtrações (que contam como adições para efeitos de complexidade) para cada entrada de C, temos que o número total de operações elementares envolvidas em (13) é

$$\frac{3+\omega}{2} n^3 + (2+\omega) n^2 - 2n = O(n^3). \quad (14)$$

Para que não haja dúvidas, o algoritmo 1 implementa o produto de Winograd (13). Notem que estamos ignorando os custos associados as variáveis necessárias para armazenar os dois últimos somatórios de (13). Continuamos com um algoritmo cúbico, *i.e.*, um algoritmo de complexidade $O(n^3)$, mas comparando-se com (3), vemos que economizamos $\frac{n^3}{2} - n^2$ multiplicações, mas pagamos o preço de termos $\frac{n^3}{2} + 3n^2 - 2n$ adições extras. A razão entre o número de operações do algoritmo de Winograd e o usual (1) para calcular o produto de duas matrizes grandes ($n \to \infty$) será

$$\frac{3+\omega}{2+2\omega}, \quad (15)$$

de onde temos que, efetivamente, o algoritmo de (13) só irá representar um ganho em relação à multiplicação usual (1) se $\omega > 1$. Chegamos a primeira atividade deste projeto.

Algoritmo 1: Produto de Winograd (13).

Entrada: Matrizes $n \times n$ A e B, com n par.
Saída: Matriz produto $C = AB$.

```
início
    para j = 1, ..., n faça
        v_j ← a_{j,1} · a_{j,2} ;
        w_j ← b_{1,j} · b_{2,j} ;
        para k = 2, ..., n/2 faça
            v_j ← v_j + a_{j,2k-1} · a_{j,2k} ;
            w_j ← w_j + b_{2k-1,j} · b_{2k,j} ;
        fim
    fim
    para i = 1, ..., n faça
        para j = 1, ..., n faça
            c_{ij} ← (a_{i,1} + b_{2,j}) · (a_{i,2} + b_{1,j}) ;
            para k = 2, ..., n/2 faça
                c_{ij} ← c_{ij} + (a_{i,2k-1} + b_{2k,j}) · (a_{i,2k} + b_{2k-1,j}) ;
            fim
            c_{ij} ← c_{ij} - v_i - w_j ;
        fim
    fim
fim
```

Algoritmo de Winograd. Implemente os algoritmos de multiplicação matricial usual e o de Winograd 1 para matrizes quadradas de ordem n par[3]. Explore seus algoritmos para estimar o valor "efetivo" de ω para o seu "computador"[4]. Há diversas maneiras de fazer isto, uma delas é a seguinte. Gere aleatoriamente várias matrizes A e B relativamente grandes (100×100 parece ser suficiente, mas eu espero que vocês explorem mais esse ponto). Calcule diversos produtos AB usando o algoritmo usual e o de Winograd e registre o tempo de execução **apenas** do produto. Quase todas as linguagens de programação tem instruções do tipo `time()` que podem ser usadas para isso. Com esses tempos e as previsões (3) e (14), estime ω com todos os cuidados estatísticos que puderem. Ao final desta atividade, tentem responder se vale a pena ou não usar o algoritmo de Winograd em situações práticas. Eu tenho uma opinião, mas não a adiantarei... ☺

Além de representar ganhos modestos, o algoritmo de Winograd (13) tem uma característica problemática, como veremos a seguir. Ele tacitamente assume que os produtos elementares são comutativos, vejam com cuidado o primeiro somatório de (13). Não poderíamos utilizá-lo, por exemplo, para multiplicar recursivamente matrizes em bloco.

2.1 O mundo subcúbico

O algoritmo de Winograd (13) não representa um ganho real de complexidade para a multiplicação matricial. A verdadeira revolução veio com o algoritmo de Strassen [4], proposto em 1969. Para mais detalhes, ver o Capítulo 28 de [1]. O ponto fundamental do algoritmo de Strassen é a observação que o produto de duas matrizes 2×2

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}\begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} = \begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{pmatrix}, \tag{16}$$

[3]O caso de n impar pode ser implementado adicionando-se a (13) os produtos faltantes, como no caso do produto escalar. Isso não nos interessa aqui.

[4]Está entre aspas porque você não vai avaliar efetivamente o custo das multiplicações `FMUL` do seu processador, mas de todo o ambiente, que envolve detalhes do sistema operacional, linguagem, interpretador (no caso do Python ou Matlab), etc. Para piorar um pouco mais, os processadores modernos podem "alterar" o fluxo de instruções de seus programas, contanto que o resultado final seja inalterado. É realmente difícil avaliar o desempenho de baixo nível de processadores a partir de aplicações de alto nível. Trataremos disso em aula.

que exigiria 8 produtos e 4 adições elementares usando-se o algoritmo habitual, pode ser calculado com apenas 7 produtos. São eles

$$\begin{aligned} m_1 &= (a_{11}+a_{22})(b_{11}+b_{22}), & m_2 &= (a_{21}+a_{22})b_{11}, \\ m_3 &= a_{11}(b_{12}-b_{22}), & m_4 &= a_{22}(b_{21}-b_{11}), \\ m_5 &= (a_{11}+a_{12})b_{22}, & m_6 &= (a_{21}-a_{11})(b_{11}+b_{12}), \\ m_7 &= (a_{12}-a_{22})(b_{21}+b_{22}), \end{aligned} \tag{17}$$

de onde temos

$$\begin{aligned} c_{11} &= m_1+m_4-m_5+m_7, & c_{12} &= m_3+m_5, \\ c_{21} &= m_2+m_4, & c_{22} &= m_1-m_2+m_3+m_6. \end{aligned} \tag{18}$$

Contando-se cuidadosamente, vemos que são 18 adições. Passar de $4(1+2\omega)$ para $18+7\omega$ operações não parece um grande ganho, mas como veremos, é um ganho substancial, revolucionário de fato. Há um refinamento do algoritmo de Strassen que nos permite utilizar apenas 15 adições. É a chamada variante de Strassen-Winograd, e será a que empregaremos aqui. Começamos com estas 8 adições

$$\begin{aligned} p_1 &= a_{21}+a_{22}, & q_1 &= b_{12}-b_{11}, \\ p_2 &= p_1-a_{11}, & q_2 &= b_{22}-q_1, \\ p_3 &= a_{11}-a_{21}, & q_3 &= b_{22}-b_{12}, \\ p_4 &= a_{12}-p_2, & q_4 &= b_{21}-q_2. \end{aligned} \tag{19}$$

Agora, as 7 multiplicações

$$\begin{aligned} m_1 &= a_{11}b_{11}, & m_5 &= p_3q_3, \\ m_2 &= a_{12}b_{21}, & m_6 &= p_4b_{22}, \\ m_3 &= p_1q_1, & m_7 &= a_{22}q_4, \\ m_4 &= p_2q_2, \end{aligned} \tag{20}$$

mais três adições

$$\begin{aligned} r_1 &= m_1+m_4, & r_3 &= r_1+m_3, \\ r_2 &= r_1+m_5, \end{aligned} \tag{21}$$

e finalmente teremos

$$\begin{aligned} c_{11} &= m_1+m_2, & c_{12} &= r_3+m_6, \\ c_{21} &= r_2+m_7, & c_{22} &= r_2+m_3. \end{aligned} \tag{22}$$

Como vocês devem imaginar, é muito fácil cometer erros com estes cálculos. A melhor maneira de certificar que de fato não há erros é apelando para pacotes algébricos. Aqui está a verificação do algoritmo de Strassen-Winograd feita no `sympy`, o pacote simbólico do Python. É bastante claro e será fácil, se alguém quiser, re-escrever usando Mathematica ou Maple. Por certo, os usuários do Matlab podem utilizar também o Maple, eles tem interfaces de comunicação.

O ganho expressivo do algoritmo de Strassen vem do seu uso recursivo. Notem que todas as operações citadas acima não envolveram a hipótese de comutatividade, quer dizer, ao contrário do que ocorreu com o algoritmo de Winograd (13), não se usou em nenhum momento aqui que os produtos elementares são comutativos. Portanto, o produto de Strassen de matrizes 2×2 pode ser usado também para matrizes em bloco! Quer dizer, poderíamos supor em (16) que as matrizes A, B e C são $2n\times 2n$, e que suas entradas a_{ij}, b_{ij} e c_{ij} são matrizes $n\times n$, e poderíamos continuar o raciocínio recursivamente. Esta é a essências das abordagens *divide and conquer*, vejam o Capítulo 4 de [1]. Com essa abordagem, podemos reduzir o cálculo do produto de matrizes $2^n\times 2^n$ ao uso recursivo de multiplicações 2×2 de Strassen. Passemos agora a determinação da complexidade dessa construção. Vamos começar pelo caso da multiplicação usual.

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}\begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} = \begin{pmatrix} a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{12} + a_{12}b_{22} \\ a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} \end{pmatrix}. \tag{23}$$

Como vemos, pensando em matrizes de bloco, a multiplicação de duas matrizes $2n\times 2n$ via o algoritmo usual envolve 8 produtos e 4 somas de matrizes $n\times n$. Notem que uma soma de matrizes $n\times n$ corresponde a n^2 somas elementares. Sejam P_k e S_k, respectivamente, o número de produtos e somas necessários para multiplicarmos duas matrizes $2^k\times 2^k$. Aplicando o algoritmo usual de multiplicação, teremos

$$P_k = 8P_{k-1}, \quad S_k = 8S_{k-1} + 4\cdot 4^{k-1}, \tag{24}$$

cuja solução, sabendo-se que $P_1 = 8$ e $S_1 = 4$, é[5]

$$P_k = 8^k, \quad S_k = 8^k - 4^k. \tag{25}$$

[5] As equações a diferenças são muito parecidas às equações diferenciais. Por exemplo, a solução geral para S_k será a soma da solução de equação homogênea $S_k = 8S_{k-1}$, que corresponde a $S_k = A8^k$, com A arbitrário, com uma solução particular, que pode ser, por exemplo, $S_k = -4^k$. A constante A é determinada da condição $S_1 = 4$.

Escrevendo-se em termos de $n = 2^k$, temos o resultado conhecido de n^3 multiplicações e $n^3 - n^2$ adições.

Para o caso do algoritmo de Strassen-Winograd, temos

$$P_k = 7P_{k-1}, \quad S_k = 7S_{k-1} + 15 \cdot 4^{k-1}, \tag{26}$$

cuja solução é

$$P_k = 7^k, \quad S_k = 5(7^k - 4^k), \tag{27}$$

o que nos dá uma complexidade, em termos de $n = 2^k$, da forma

$$(5 + \omega)n^{\log_2 7} - 5n^2 = O\left(n^{\log_2 7}\right). \tag{28}$$

Como $\log_2 7 = 2.807\ldots$ temos, efetivamente, um ganho de complexidade em relação à multiplicação usual. O algoritmo de Strassen, com ou sem variante de Winograd, foi o primeiro algoritmo subcúbico para a multiplicação matricial.

As vezes é conveniente alternar entre a multiplicação usual e a de Strassen nas recursões. Neste caso, teremos

$$P_k = a_k P_{k-1}, \quad S_k = a_k S_{k-1} + b_k \cdot 4^{k-1}, \tag{29}$$

sendo $a_k = 8$ e $b_k = 4$ se a k-ésima operação for do tipo usual, ou $a_k = 7$ e $b_k = 15$ se for do tipo Strassen. Estes algoritmos são chamados híbridos e cada uma das recursões é identificada pelas letras s (Strassen) ou n (normal). Por exemplo, o produto $snsn$ corresponde a uma recursão de 4 níveis para a multiplicação de matrizes 16×16. Da esquerda para a direita, os produtos são: Strassen-Winograd ($a_4 = 7$ and $b_4 = 15$), normal ($a_3 = 8$ and $b_3 = 4$), Strassen-Winograd ($a_2 = 7$ and $b_2 = 15$) e finalmente um normal ($a_1 = 8$ and $b_1 = 4$), aplicados recursivamente. Neste caso, teremos $P_4 = 3136$ and $S_4 = 6336$. Os produtos $nnnn$ e $ssss$ correspondem, respectivamente, à multiplicação usual e ao algoritmo de Strassen-Winograd "puro". Chegamos a segunda tarefa.

Algoritmos de Strassen-Winograd. A partir da análise da complexidade dos algoritmos de multiplicação matricial usual e de Strassen-Winograd, determine o valor de n mínimo para que a multiplicação de Strassen-Winograd seja vantajosa, do ponto de vista de operações aritméticas elementares, para multiplicação de matrizes $2^n \times 2^n$, supondo $\omega = 1$. Implemente os algoritmos de multiplicação matricial usual e o de Strassen-Winograd **recursivamente** para essas matrizes e compare seus resultados.

Referências

[1] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, *Introduction to Algorithms*, (2009).

[2] `https://www.agner.org/optimize/`

[3] D. Knuth, *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*, (1997).

[4] V. Strassen, *Gaussian Elimination is not Optimal*, Numer. Math. **13**, 354 (1969)