*INTERMATHS, VOL. 5, NO. 2 (**2024**)*, **117–133**
https://doi.org/10.22481/intermaths.v5i2.15564
Artigos Gerais
licença creative commons

Algoritmos Subcúbicos para Multiplicação Matricial

Subcubic Algorithms for Matrix Multiplication

Tomy Felixon [a,*], **Fabiana Correia Pereira** [a], **João Socorro Pinheiro Ferreira** [b]

[a] Universidade Estadual de Campinas, Campinas - SP, Brasil; [b] Universidade Federal do Amapá, Macapa - AP, Brasil

* Autor Correspondente: t123735@dac.unicamp.br

Resumo: Este trabalho apresenta os resultados da pesquisa bibliográfica e uso de ambientes computacionais sobre Complexidade Algorítmica. Na primeira parte, abordamos algumas propriedades da multiplicação matricial, além de apresentar o algoritmo simples de dividir e conquistar. Na segunda parte do trabalho, apresentamos os resultados e discussões dando ênfase principalmente no algoritmo de Winograd e algoritmo de Strassen para multiplicação de matrizes.

Palavras-chave: Matrizes. Strassen. Strassen-Winograd.

Abstract: This paper presents the results of the bibliographic research and use of computational environments on Algorithmic Complexity. In the first part, we address some properties of matrix multiplication, in addition to presenting the simple divide and conquer algorithm. In the second part of the paper, we present the results and discussions with emphasis mainly on the Winograd algorithm and Strassen algorithm for matrix multiplication.

keywords: Matrices. Strassen. Strassen-Winograd.

1 Introdução

O tema desta pesquisa é estudar complexidade algorítmica de multiplicação de matrizes.Em resumo, tal termo está relacionado a analisar e classificar o desempenho dos algoritmos em termos de tempo e espaço. Serão analisadas a complexidade de quatro tipos de algoritmos de multiplicação de matrizes: Usual, Usual Recursivo, Winograd, Strassen e Strassen-Winograd.

O termo subcúbico provém da ordem de complexidade de multiplicação usual de duas matrizes, que é de $\mathcal{O}(n^3)$ operações algébricas, por exemplo, se duas matrizes são de ordem dois, então são necessárias oito multiplicações e quatro adições para para efetuar o produto entre ambas, ou seja, oito é potência de dois elevado ao cubo (2^3), já o termo subcúbico, é proveniente do resultados dos estudos de Winograd e Strassen, entre outros pesquisadores, sobre à realidade de se multiplicar duas matrizes com ordem menor de que três, isto é, $\mathcal{O}\left(n^{\log_2 7}\right) = \mathcal{O}\left(n^{2.81}\right)$ - faz com que sejam necessárias somente sete multiplicações.

E para iniciar nosso estudo, principalmente no que concerne aos algoritmos subcúbicos para multiplicação de matrizes, apresentamos algumas definições e conceitos relacionados às multiplicações de matrizes, bem como apresentamos sua complexidade computacional, para

Submetido em: 10 de Junho de 2024 Aprovado em: 24 de Dezembro de 2024 Publicado em: 31 de Dezembro de 2024
ISSN 2675-8318 ©2024 INTERMATHS. Publicado por Edições UESB. Este é um artigo de acesso aberto sob a licença CC BY 4.0.

tanto, mostraremos os Algoritmos 1, 2, 3, 4 e 5, com a função time para "medir" a complexidade algorítmica.

Na Subseção 2.1, apresentamos o conceito do chamado algoritmo simples de dividir e conquistar, tal definição está referenciada em [1].

Na Seção 3, apresentamos os resultados e discussões da pesquisa, e a princípio, definimos o algoritmo de Winograd como uma alternativa mais "econômica" ao produto usual de matrizes, mas que precisa atender alguns critérios específicos para garantir sua efetividade.

Na Subseção 3.2, analisamos um algoritmo para multiplicar matrizes quadradas de tamanho (número de linhas e colunas) n com complexidade de $\mathcal{O}(n^3)$ multiplicações. Um algoritmo mais eficiente foi proposto por Strassen (1969). A idéia do algoritmo é: subdividir as matrizes num produto $A \times B = C$ em quatro submatrizes com a metade do tamanho (e, portanto, um quarto de elementos). Este método aparenta ser mais vantajoso que o algoritmo de Winograd. Mas a partir de alguns aperfeiçoamentos foi possível chegar a um algoritmo mais "econômico" do ponto de vista da complexidade numérica, o algoritmo Strassen-Winograd, como veremos, na Subseção 3.3.

Os algoritmos em Python foram processados em um LAPTOP-CT2I1N34, com Processador 11th Gen Intel(R) Core(TM) i5-1135G7 2.40GHz 2.42 GHz RAM instalada 8,00 GB (utilizável: 7,73 GB), Tipo de sistema Sistema operacional de 64 bits, processador baseado em x64.

2 Multiplicação matricial

Sejam $\mathbf{A}$: $m \times p$ e $\mathbf{B}$: $p \times n$, a matriz produto será a matriz $\mathbf{C}$: $m \times n$ da definição de produto matricial, temos que as entradas de $\mathbf{C}$, em termos das entradas de $\mathbf{A}$ e $\mathbf{B}$, são dadas por

$$c_{ij} = \sum_{k=1}^{p} a_{ik} b_{kj}, \tag{2.1}$$

com $1 \leq i \leq m$ e $1 \leq j \leq n$.

Desenvolvemos o produto, temos que:

$$\mathbf{AB} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1p} \\ a_{21} & a_{22} & \cdots & a_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m-1,1} & a_{m-1,2} & \cdots & a_{m-1,p} \\ a_{m1} & a_{m2} & \cdots & a_{mp} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1,n-1} & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2,n-1} & b_{2n} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ b_{p1} & b_{p2} & \cdots & b_{p,n-1} & b_{pn} \end{bmatrix}$$

$$\mathbf{C} = \begin{bmatrix} c_{11} & c_{12} & \cdots & c_{1n} \\ c_{21} & c_{22} & \cdots & c_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{m-1,1} & a_{m-1,2} & \cdots & a_{m-1,n} \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} =$$

$$\mathbf{C} = \begin{bmatrix} \sum_{k=1}^{p} a_{1k}b_{k1} & \sum_{k=1}^{p} a_{1k}b_{k2} & \cdots & \sum_{k=1}^{p} a_{1k}b_{kn} \\ \sum_{k=1}^{p} a_{2k}b_{k1} & \sum_{k=1}^{p} a_{2k}b_{k2} & \cdots & \sum_{k=1}^{p} a_{2k}b_{kn} \\ \vdots & \vdots & \ddots & \vdots \\ \sum_{k=1}^{p} a_{m-1,k}b_{k1} & \sum_{k=1}^{p} a_{m-1,k}b_{k2} & \cdots & \sum_{k=1}^{p} a_{m-1,k}b_{kn} \\ \sum_{k=1}^{p} a_{mk}b_{k1} & \sum_{k=1}^{p} a_{mk}b_{k2} & \cdots & \sum_{k=1}^{p} a_{mk}b_{kn} \end{bmatrix}_{m\times n}. \tag{2.2}$$

A complexidade computacional da multiplicação de matrizes retangulares é expressa matematicamente como:

$$\mathcal{O}(m \cdot n \cdot p),$$

onde:

- "m" é o número de linhas da matriz $\mathbf{A}$.
- "n" é o número de colunas da matriz $\mathbf{A}$ (e também o número de linhas da matriz $\mathbf{B}$).
- "p" é o número de colunas da matriz $\mathbf{B}$.

Esta fórmula representa o número de operações de multiplicações necessárias para calcular o produto de duas matrizes retangulares. É uma representação do custo computacional em termos do tamanho das matrizes envolvidas.

Por exemplo, para as matrizes

$$\mathbf{A} = \begin{bmatrix} 2 & 3 \\ 3 & 5 \end{bmatrix} \qquad e \qquad \mathbf{B} = \begin{bmatrix} 7 & 11 \\ 11 & 13 \end{bmatrix}, \tag{2.3}$$

a matriz

$$\mathbf{C} = \begin{bmatrix} 2\cdot 7 + 3\cdot 11 & 2\cdot 11 + 3\cdot 13 \\ 3\cdot 7 + 5\cdot 11 & 3\cdot 11 + 5\cdot 13 \end{bmatrix} = \begin{bmatrix} 14 + 33 & 22 + 39 \\ 21 + 55 & 33 + 65 \end{bmatrix} = \begin{bmatrix} 47 & 61 \\ 76 & 98 \end{bmatrix}. \tag{2.4}$$

A complexidade computacional do produto matricial (2.4) é de 8 multiplicações e 4 adições, totalizando 12 operações computacionais. No caso da matriz quadrada de ordem n, é calculado por $C(n) = n \cdot n \cdot (2n-1) = 2n^3 - n^2$ e tem ordem de grandeza $\mathcal{O}(n^3)$. Para $n = 2$, $C(2) = 12$ operações algébricas computacionais. O tempo gasto para o computador realizar este produto usual (cf. Equação (2.1)), é de aproximadamente: `3.86238e-05` segundos.

Para obter a matriz (2.2), deve-se implementar a Equação (2.1) através do Algoritmo 1.

A seguir o código fonte em Phyton do Algoritmo 1, com a função `time` para medir a complexidade do algoritmo usual de produto entre duas matrizes:

```
import numpy as np
import time

def multiplicarMatrizes(A, B):
    # Verificar se as matrizes podem ser multiplicadas
    if A.shape[1] != B.shape[0]:
```

Algorithm 1 Multiplicação de Matrizes a partir da Equação (2.1)

function MULTIPLICARMATRIZES($\mathbf{A}, \mathbf{B}$)
clc; ▷ Inicialize as matrizes A e B com valores de exemplo
if size$(A, 2) \neq$ size$(B, 1)$ **then**
error('Erro: As matrizes não podem ser multiplicadas. O número de colunas de A deve ser igual ao número de linhas de B.')
$[m, n] =$ size(A);
$[n, p] =$ size(B);
$C =$ zeros(m, p);
for $i = 1$ até m **do**
for $j = 1$ até p **do**
for $k = 1$ até size$(B, 1)$ **do**
$C(i, j) = C(i, j) + A(i, k) \cdot B(k, j)$;
return C;

```
        raise ValueError('Erro: As matrizes não podem ser multiplicadas.
        O número de colunas de A deve ser igual ao número de linhas de B.')

    m, n = A.shape
    n, p = B.shape
    C = np.zeros((m, p))

    # Iniciar o temporizador
    start_time = time.time()

    # Realizar a multiplicação das matrizes
    for i in range(m):
        for j in range(p):
            for k in range(n):
                C[i, j] += A[i, k] * B[k, j]

    # Parar o temporizador e calcular o tempo decorrido
    elapsed_time = time.time() - start_time

    return C, elapsed_time

# Exemplo de uso
A = np.random.randn(10, 10)  # simular para diversos
#tamanhos de matriz (substitua pela sua matriz)
B = np.random.randn(10, 10)  simular para diversos
#tamanhos de matriz (substitua pela sua matriz)

C, elapsed_time = multiplicarMatrizes(A, B)
print(f"Tempo de execução: {elapsed_time} segundos")
```

Para imprimir a matriz produto **C**, usar o seguinte comando:

```
print("Matriz C:")
```

```
print(C)
```

Na Tabela 1, estão os registros de seis simulações de produtos entre duas matrizes randômicas normalizadas e usamos a função `time` para medir a complexidade de tais produtos. Utilizamos matrizes de ordem n^2, com o propósito de comparar com outros algoritmos de multiplicação, ao longo deste trabalho.

Tabela 1 – Tempo de execução do produto usual de matrizes, a partir da Equação (2.1) e Algoritmo 2.1.

n	2	8	64	128	512	1024
Tempo (s)	0.000030	0.000648	0.175758	1.393882	103.128729	840.871537

Fonte: elaborado pelos autores.

Para a matriz 512×512 o tempo que o computador utilizou para processar o algoritmo foi de aproximadamente 2 minutos; e o tempo que gastou para a matriz 1024×1024 foi de aproximadamente 17 minutos.

A complexidade da multiplicação de matrizes quadradas é do tipo

$$(1+\omega)n^3 - n^2 = \mathcal{O}(n^3), \tag{2.5}$$

pois é fácil ver que

$$0 \leq (1+\omega)n^3 - n^2 \leq (1+\omega)n^3, \tag{2.6}$$

para todo n.

Em resumo, o número total de operações aritméticas no programa (Algoritmo 1) depende das dimensões das matrizes $\mathbf{A}$ e $\mathbf{B}$. Se as matrizes $\mathbf{A}$ e $\mathbf{B}$ forem $n \times n$, o número total de operações aritméticas será $2 \times n^3$,já que o *loop* triplo executa n^3 iterações e, para cada iteração, realizamos duas operações (multiplicação e adição).

2.1 Um algoritmo simples de divisão e conquista

Para simplificar, quando usamos um algoritmo de dividir e conquistar para calcular o produto da matriz $\mathbf{C} = \mathbf{A} \cdot \mathbf{B}$, assumimos que n é uma potência exata de 2 em cada uma das $n \times n$ matrizes. Fazemos essa suposição porque em cada etapa de divisão, dividiremos $n \times n$ matrizes em quatro matrizes $n/2 \times n/2$, e assumindo que n é uma potência exata de 2, temos a garantia de que, enquanto $n \geq 2$, a dimensão $n/2$ é um número inteiro. [1]

Suponha que particionemos cada uma das matrizes $\mathbf{A}$, $\mathbf{B}$ e $\mathbf{C}$ em quatro matrizes $\frac{n}{2} \times \frac{n}{2}$

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}, \qquad B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}, \qquad e \qquad C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}, \tag{2.7}$$

Então reescrevemos a equação $\mathbf{C}$=$\mathbf{A}$·B como

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \cdot \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}. \tag{2.8}$$

A Equação (2.8) corresponde às quatro equações

$$\begin{array}{rcl} C_{11} & = & A_{11}B_{11} + A_{12}B_{21}, \\ C_{12} & = & A_{11}B_{12} + A_{12}B_{22}, \\ C_{21} & = & A_{21}B_{11} + A_{22}B_{21}, \\ C_{22} & = & A_{21}B_{12} + A_{22}B_{22}. \end{array} \tag{2.9}$$

Cada uma dessas quatro equações especifica duas multiplicações de matrizes $n/2 \times n/2$ e a adição de seus produtos $n/2 \times n/2$. Podemos usar essas equações para criar um algoritmo simples e recursivo de dividir e conquistar, visto no Algoritmo 2. Este método torna-se mais vantajoso computacionalmente em relação ao usual, tendo em vista que a m

Algorithm 2 Multiplicação de Matrizes Quadradas de Forma Recursiva

function SQUAREMATRIXMULTIPLYRECURSIVE(A, B)
$n \leftarrow$ Número de linhas de A
Deixe **C** ser uma nova matriz $n \times n$
if $n == 1$ **then**
$C[1,1] \leftarrow A[1,1] \cdot B[1,1]$
else
Particione **A**, **B** e **C** conforme as equações (4.9)
$C_{11} \leftarrow$ squareMatrixMultiplyRecursive(A_{11}, B_{11}) + squareMatrixMultiplyRecursive(A_{12}, B_{21})
$C_{12} \leftarrow$ squareMatrixMultiplyRecursive(A_{11}, B_{12}) + squareMatrixMultiplyRecursive(A_{12}, B_{22})
$C_{21} \leftarrow$ squareMatrixMultiplyRecursive(A_{21}, B_{11}) + squareMatrixMultiplyRecursive(A_{22}, B_{21})
$C_{22} \leftarrow$ squareMatrixMultiplyRecursive(A_{21}, B_{12}) + squareMatrixMultiplyRecursive(A_{22}, B_{22})
Junte as submatrizes **C** como mostrado abaixo:
$C_{\text{superior esquerda}} \leftarrow C_{11}$
$C_{\text{superior direita}} \leftarrow C_{12}$
$C_{\text{inferior esquerda}} \leftarrow C_{21}$
$C_{\text{inferior direita}} \leftarrow C_{22}$
return C

Tabela 2 – Tempo de execução do produto na forma recursiva de matrizes, a partir da Equação (2.9) e Algoritmo 2.

n	2	8	64	128	512	1024
Tempo (s)	0.000218	0.002130	0.592232	6.292380	335.786021	2689.4999435

Fonte: elaborado pelos autores.

Ao compararmos os tempos entre as Tabelas 1 e 2, concluímos o seguinte fato: para as matrizes randômicas $2^n \times 2^n$ para $n = 1, 3, 6, 7, 9, 10$, os tempos não são os mesmos, pois todos os tempos do algoritmo recursivo aumentaram, $u \longmapsto r$, conforme as comparações na primeira linha da Tabela 8.

Outro exemplo, para as matrizes (2.4), a sua forma particionada em potência de 2 é $2^2 = 4$ submatrizes 1×1, tais que $A_{11} = [2]$, $A_{12} = [3]$, $A_{21} = [3]$ e $A_{22} = [5]$, $B_{11} = [7]$, $B_{12} = [11]$,

$B_{21} = [11]$ e $B_{22} = [13]$. O produto $\mathbf{C}$ é obtido pelas Equações (2.9) será:

$$\begin{array}{rcrcrcl} C_{11} & = & 2 \cdot 7 + 3 \cdot 11 & = & 14 + 33 & = & 47, \\ C_{12} & = & 2 \cdot 11 + 3 \cdot 13 & = & 22 + 39 & = & 61, \\ C_{21} & = & 3 \cdot 7 + 5 \cdot 11 & = & 21 + 55 & = & 76, \\ C_{22} & = & 3 \cdot 11 + 5 \cdot 13 & = & 33 + 65 & = & 98. \end{array} \tag{2.10}$$

Portanto, a matriz

$$\mathbf{C} = \begin{bmatrix} 47 & 61 \\ 76 & 98 \end{bmatrix}. \tag{2.11}$$

O tempo de execução: `0.0001697540283203125` segundos, pelo algoritmo recursivo é aproximadamente 4.6 vezes mais lento de que o produto usual, encontrado na matriz (2.4) .

3 Resultados e Discussões

3.1 Algoritmo de Winograd

Uma operação fundamental envolvendo matrizes é, obviamente, a multiplicação de matrizes: se $\mathbf{A} = (a_{ij})$ é uma matriz $m \times p$, $\mathbf{B} = (b_{jk})$ é uma matriz $p \times n$ e $\mathbf{C} = (c_{ik})$ é uma matriz $m \times n$, então a fórmula $\mathbf{C} = \mathbf{AB}$ corresponde à Equação (2.1). Vamos restringir às matrizes quadradas $n \times n$. [3] O algoritmo baseado na definição (2.1) requer $(1 + \omega)n^3 - n^2$ operações (adições) elementares. Cf. (2.5).

Sejam $\vec{V}$, $\vec{W} \in \mathbb{R}^n$, o produto escalar entre eles é definido por

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

Esta operação, como definida, requer n produtos e $n - 1$ adições para o cálculo do produto escalar. Poderíamos fazer de outra forma? A resposta é sim. Vamos supor momentaneamente que n seja par. Teremos a seguinte identidade[3]

$$\sum_{k=1}^{n} v_k w_k = \sum_{k=1}^{n/2} (v_{2k-1} + w_{2k})\,(v_{2k} + w_{2k-1}) - \sum_{k=1}^{n/2} v_{2k-1} v_{2k} - \sum_{k=1}^{n/2} w_{2k-1} w_{2k}. \tag{3.1}$$

Esta equação pode ser considerada como o cálculo de polinômios simultâneos em variáveis n; cada polinômio é o "produto interno" de dois vetores de n coordenadas. Um cálculo direto envolveria várias multiplicações e adições; mas Shmuel Winograd (1936 - 2019) descobriu em 1967 que existe uma maneira de trocar cerca de metade das multiplicações por adições.[2]

O algoritmo de Winograd para multiplicação matricial explora a identidade (3.1) para matrizes. A motivação principal é que as componentes c_{ij} do produto matricial (2.1) são, de fato, produtos escalares dos vetores linhas e colunas de $\mathbf{A}$ e $\mathbf{B}$, respectivamente. Teremos

$$c_{ij} = \sum_{k=1}^{n/2} (a_{i,2k-1} + b_{2k,j})(a_{i,2k} + b_{2k-1,j}) - \sum_{k=1}^{n/2} a_{i,2k-1} a_{i,2k} - \sum_{k=1}^{n/2} b_{2k-1,j} b_{2k,j}. \tag{3.2}$$

[2]

Por exemplo, para a Equação (3.1), em particular com $n = 2$, $\vec{V} = (v_1, v_2)$ e $\vec{W} = (w_1, w_2)$,

temos que:

$$\begin{aligned}
\sum_{k=1}^{2} v_k w_k &= \sum_{k=1}^{1} (v_{2k-1} + w_{2k})(v_{2k} + w_{2k+1}) - \sum_{k=1}^{1} v_{2k-1}v_{2k} - \sum_{k=1}^{1} w_{2k-1}w_{2k} \\
v_1w_1 + v_2w_2 &= (v_1 + w_2)(v_2 + w_1) - v_1v_2 - w_1w_2 \\
&= v_1v_2 + v_1w_1 + w_2v_2 + w_2w_1 - v_1v_2 - w_1w_2 \\
&= v_1w_1 + v_2w_2,
\end{aligned}$$

a igualdade verificada é o produto escalar entre os vetores $\vec{V}$ e $\vec{W}$. O produto escalar, comumente usado na geometria euclidiana, é um caso especial de produto interno.

Sejam as matrizes $\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}$ e $\mathbf{B} = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$, então a partir da Equação (3.2) pode-se escrever os elementos da matriz produto $\mathbf{C}$, para $n = 2$, é:

$$c_{ij} = \sum_{k=1}^{1}(a_{i,2k-1} + b_{2k,j})(a_{i,2k} + b_{2k-1,j}) - \sum_{k=1}^{1} a_{i,2k-1}a_{i,2k} - \sum_{k=1}^{1} b_{2k-1,j}b_{2k,j}.$$

O elemento c_{11} para $k = 1$, é:

$$\begin{aligned}
c_{11} &= (a_{1,1} + b_{2,1})(a_{1,2} + b_{1,1}) - a_{1,1}a_{1,2} - b_{1,1}b_{2,1} \\
&= a_{11}a_{12} + a_{11}b_{11} + b_{21}a_{12} + b_{21}b_{11} - a_{11}a_{12} - b_{11}b_{21} \\
&= a_{11}b_{11} + a_{12}b_{21};
\end{aligned}$$

O elemento c_{12} para $k = 1$, é:

$$\begin{aligned}
c_{12} &= (a_{1,1} + b_{2,2})(a_{1,2} + b_{1,2}) - a_{1,1}a_{1,2} - b_{1,2}b_{2,2} \\
&= a_{11}a_{12} + a_{11}b_{12} + b_{22}a_{12} + b_{22}b_{12} - a_{11}a_{12} - b_{12}b_{22} \\
&= a_{11}b_{12} + a_{12}b_{22};
\end{aligned}$$

O elemento c_{21} para $k = 1$, é:

$$\begin{aligned}
c_{21} &= (a_{2,1} + b_{2,1})(a_{2,2} + b_{1,1}) - a_{2,1}a_{2,2} - b_{1,1}b_{2,1} \\
&= a_{21}a_{22} + a_{22}b_{11} + b_{21}a_{22} + b_{21}b_{11} - a_{21}a_{22} - b_{11}b_{21} \\
&= a_{22}b_{11} + a_{22}b_{21};
\end{aligned}$$

O elemento c_{22} para $k = 1$, é:

$$\begin{aligned}
c_{22} &= (a_{2,1} + b_{2,2})(a_{2,2} + b_{1,2}) - a_{2,1}a_{2,2} - b_{1,2}b_{2,2} \\
&= a_{21}a_{22} + a_{21}b_{12} + b_{22}a_{22} + b_{22}b_{12} - a_{21}a_{22} - b_{12}b_{22} \\
&= a_{22}b_{12} + a_{22}b_{22}.
\end{aligned}$$

Com os elementos c_{11}, c_{12}, c_{21} e c_{22} formamos o produto usual de duas matrizes 2×2.

Agora vamos escrever o algoritmo de Winograd a partir da expressão (3.2):

Tabela 3 – Tempo de execução do Algoritmo 3 Produto de Winograd.

n	2	8	64	128	512	1024
Tempo (s)	0.000013	0.000672	0.463794	1.547172	105.914162	–

Fonte: elaborado pelos autores.

Algorithm 3 Produto de Winograd

```
1:  procedure WinogradProduct(A, B, C, n)
2:      Entrada: Matrizes n × n A e B, com n par.
3:      Saída: Matriz produto C = AB.
4:      for j = 1 to n do
5:          v_j ← a_{j,1} · a_{j,2}
6:          w_j ← b_{1,j} · b_{2,j}
7:          for k = 2 to n/2 do
8:              v_j ← v_j + a_{j,2k−1} · a_{j,2k}
9:              w_j ← w_j + b_{2k−1,j} · b_{2k,j}
10:     for i = 1 to n do
11:         for j = 1 to n do
12:             c_{i,j} ← (a_{i,1} + b_{2,j}) · (a_{i,2} + b_{1,j})
13:             for k = 2 to n/2 do
14:                 c_{i,j} ← c_{i,j} + (a_{i,2k−1} + b_{2k,j}) · (a_{i,2k} + b_{2k−1,j})
15:             c_{i,j} ← c_{i,j} − v_i − w_j
```

Analisando os tempos da Tabela 3 e comparando com o método usual da Tabela 1, os tempos do algoritmo de Winograd diminuem -57% na matriz 2×2 e para as outras cinco matrizes o tempo de execução é maior, atingindo 164% aproximadamente para a matriz 64×64. Em comparação com o método recursivo, Tabela 2, os tempos para a execução do algoritmo nos cinco produtos são menores, i.e., o programa Winograd é muito mais rápido. Deixamos registrado que o arquivo não processou para a matriz 1024, pois o tempo de execução do algoritmo é maior de que a capacidade de processamento do computador. Os percentuais de comparação do algoritmo de Winograd com o usual e o recursivo encontram-se respectivamente nas linhas 2 e 4 da Tabela 8.

3.2 Algoritmo de Strassen para multiplicação de matrizes

Volker Strassen (1936 - está com 87 anos), publicou o algoritmo em 1969. Apesar de seu algoritmo ser apenas um pouco mais rápido do que o método padrão para a multiplicação de matrizes, ele foi o primeiro a observar que a abordagem usual, Equação (2.1), não é ótima.

A chave do método de Strassen é tornar a árvore de recursão um pouco menos espessa. Ou seja, em vez de realizar *oito* multiplicações recursivas de matrizes $n/2 \times n/2$, ele realiza apenas *sete*. O custo de eliminar uma multiplicação de matrizes será de várias novas adições de $n/2 \times n/2$ matrizes, mas ainda assim apenas um número constante de adições. Como antes, o número constante de adições de matrizes será representado pela notação Θ, quando configurarmos a equação de recorrência para caracterizar o tempo de execução. [1]

O método de Strassen não é nada óbvio. Ele tem quatro etapas:

1. Dividir as matrizes de entrada **A** e **B** e a matriz de saída **C** em submatrizes $n/2 \times n/2$, como na equação (2.7). Esta etapa leva $\Theta(1)$ tempo para cálculo do índice, apenas como em MATRIZ QUADRADA-MULTIPLICADA-RECURSIVA.

2. Criar 10 matrizes $S_1, S_2, \cdots, S_{10}$, cada uma das quais é $n/2 \times n/2$ e é a soma ou diferença de duas matrizes criadas na etapa 1. Podemos criar todas as 10 matrizes em $\Theta(n^2)$.

3. Usar as submatrizes criadas na etapa 1 e as 10 matrizes criadas na etapa 2, calcular recursivamente sete produtos de matrizes $P_1, P_2, \cdots, P_7$. Cada matriz P_i é $n/2 \times n/2$.

4. Calcular as submatrizes desejadas $C_{11}, C_{12}, C_{21}, C_{22}$ da matriz resultante C adicionando e subtraindo várias combinações das matrizes P_i. Podemos calcular todas as quatro submatrizes em tempo $\Theta(n^2)$.

Em 1969, Strassen publicou o artigo científico *Gaussian Elimination is not Optimal*[1][4] demonstrando que o Algoritmo 4 calcula o produto de duas matrizes de ordem n com com menos de $4.7n^{\log_2 7}$ operações aritméticas; o método usual requer aproximadamente $2n^3$ operações aritméticas. O algoritmo gera algoritmos para inverter uma matriz de ordem n, resolver um sistema de n equações lineares em n incógnitas, calcula um determinante de ordem n etc., todos exigindo menos que a constante $n^{\log_2 7}$ operações aritméticas.

Algorithm 4 Algoritmo $\alpha_{n,k}$ para multiplicação de matrizes

procedure MATRIXMULTIPLY($\mathbf{A}, \mathbf{B}$)
  if $k = 0$ **then**
    Realize a multiplicação usual de matrizes A e B e retorne o resultado.
  else
    Divida as matrizes **A** e **B** em submatrizes A_{ij} e B_{ij} de ordem $n2^k$.
    Calcule as seguintes submatrizes intermediárias:
    $I = (A_{11} + A_{22})(B_{11} + B_{22})$
    $II = (A_{21} + A_{22})B_{11}$
    $III = A_{11}(B_{12} - B_{22})$
    $IV = A_{22}(-B_{11} + B_{21})$
    $V = (A_{11} + A_{12})B_{22}$
    $VI = (-A_{11} + A_{21})(B_{11} + B_{12})$
    $VII = (A_{12} - A_{22})(B_{21} + B_{22})$
    Calcule as submatrizes resultantes:
    $C_{11} = I + IV - V + VII$
    $C_{21} = II + IV$
    $C_{12} = III + V$
    $C_{22} = I + III - II + VI$
    Retorne a matriz resultante C.

O código fonte no Phyton para compilar o Algoritmo 4 é o seguinte:

```
import numpy as np
import time

def matrix_multiply_standard(A, B):
    # Algoritmo padrão para multiplicação de matrizes
    return np.dot(A, B)

def alpha_matrix_multiply(A, B, k):
    if k == 0:
        # Caso base: Algoritmo padrão de multiplicação de matrizes
        return matrix_multiply_standard(A, B)

    m = len(A)
    n = len(A[0])
```

[1] A eliminação gaussiana não é otimizada.

```
    # Dividir as matrizes em submatrizes menores
    half_m = m // 2
    half_n = n // 2

    A11 = A[:half_m, :half_n]
    A12 = A[:half_m, half_n:]
    A21 = A[half_m:, :half_n]
    A22 = A[half_m:, half_n:]

    B11 = B[:half_n, :half_n]
    B12 = B[:half_n, half_n:]
    B21 = B[half_n:, :half_n]
    B22 = B[half_n:, half_n:]

    # Calcula submatrizes intermediárias
    I = (A11 + A22).dot(B11 + B22)
    II = (A21 + A22).dot(B11)
    III = A11.dot(B12 - B22)
    IV = A22.dot(-B11 + B21)
    V = (A11 + A12).dot(B22)
    VI = (-A11 + A21).dot(B11 + B12)
    VII = (A12 - A22).dot(B21 + B22)

    # Calcula submatrizes resultantes
    C11 = I + IV - V + VII
    C12 = III + V
    C21 = II + IV
    C22 = I + III - II + VI

    # Concatena submatrizes resultantes
    C = np.vstack((np.hstack((C11, C12)), np.hstack((C21, C22))))

    return C

# Função para medir o tempo de execução
def measure_execution_time(A, B, k):
    start_time = time.time()
    result = alpha_matrix_multiply(A, B, k)
    end_time = time.time()
    execution_time = end_time - start_time
    return result, execution_time

# Tamanho da matriz
m = 2
n = 2
k = 2  # Você pode ajustar k conforme necessário
```

```
# Gere matrizes de números aleatórios para teste
A = np.random.randn(m, n)
B = np.random.randn(n, m)

# Matriz A
#A = np.array([[2, 3], [3, 5]])

# Matriz B
#B = np.array([[7, 11], [11, 13]])

# Medir o tempo de execução e realizar a multiplicação
result, execution_time = measure_execution_time(A, B, k)

print(f"Resultado da multiplicação:\n{result}")
print(f"Tempo de execução: {execution_time:.6f} segundos")
```

A seguir, na Tabela 4 calculamos seis produtos matriciais utilizando o algoritmo de Strasssen, para comparar com os outros métodos estudados.

Tabela 4 – Tempo de execução do Algoritmo 4 de Strassen.

n	2	8	64	128	512	1024
Tempo (s)	0.000215	0.001467	0.479345	3.352598	182.802909	1229.282250

Fonte: elaborado pelos autores.

Ao compararmos o tempo de execução do produto matricial 2×2 na Tabela 4 com o da Tabela 1, o mesmo é 7 vezes maior, porém ao compararmos com o produto recursivo, ele é 1% menor.

3.3 Algoritmo de Strassen-Winograd

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 [3]

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

Agora, as 7 multiplicações:

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

Mais 3 adições

$$\begin{aligned} r_1 &= m_1 + m_4, & r_3 &= r_1 + m_3, \\ & & t_2 &= r_1 + m_5, \end{aligned}$$

e finalmente teremos

$$c_{11} = m_1 + m_2, \qquad c_{12} = r_3 + m_6,$$
$$c_{21} = r_2 + m_7, \qquad c_{22} = r_2 + m_7.$$

As adições e multiplicações acima, escrevemos um algoritmo no phyton abaixo para calcularmos a complexidade computacional do refinamento do algoritmo de Strassen-Winograd. Na Tabela 5 registramos os referidos tempos para matrizes de ordem $2^n \times 2^n$, para $n = 1, 3, 6, 7, 9, 10$.

```
import numpy as np
import time

def strassen_winograd(a, b):
    n = len(a)

    # Caso base para a matriz 1x1
    if n == 1:
        return np.dot(a, b)

    # Dividir as matrizes em quatro submatrizes
    a11 = a[:n//2, :n//2]
    a12 = a[:n//2, n//2:]
    a21 = a[n//2:, :n//2]
    a22 = a[n//2:, n//2:]

    b11 = b[:n//2, :n//2]
    b12 = b[:n//2, n//2:]
    b21 = b[n//2:, :n//2]
    b22 = b[n//2:, n//2:]

    # Calcular as variáveis intermediárias usando as fórmulas fornecidas
    p1 = a21 + a22
    q1 = b12 - b11

    p2 = p1 - a11
    q2 = b22 - q1

    p3 = a11 - a21
    q3 = b22 - b12

    p4 = a12 - p2
    q4 = b21 - q2

    m1 = a11 * b11
    m5 = p3 * q3

    m2 = a12 * b21
    m6 = p4 * b22

    m3 = p1 * q1
```

```
    m7 = a22 * q4

    m4 = p2 * q2

    # Calcular as variáveis finais usando as fórmulas fornecidas
    r1 = m1 + m4
    r3 = r1 + m3

    r2 = r1 + m5

    t2 = r1 + m5

    c11 = m1 + m2
    c12 = r3 + m6

    c21 = r2 + m7
    c22 = r2 + m7

    # Combinar as submatrizes resultantes em uma única matriz
    c = np.zeros((n, n))
    c[:n//2, :n//2] = c11
    c[:n//2, n//2:] = c12
    c[n//2:, :n//2] = c21
    c[n//2:, n//2:] = c22

    return c

# Função para medir o tempo de execução do algoritmo
def measure_time(a, b):
    start_time = time.time()
    strassen_winograd(a, b)
    end_time = time.time()
    elapsed_time = end_time - start_time
    print(f"Tempo de execução: {elapsed_time:.6f} segundos")

# Teste com matrizes de tamanho nxn
n=1024
A = np.random.rand(n, n)
B = np.random.rand(n, n)

measure_time(A, B)
```

Tabela 5 – Tempo de execução do Algoritmo refinado de Strassen-Winograd.

n	2	8	64	128	512	1024
Tempo (s)	0.000081	0.000073	0.000107	0.000524	0.004783	0,019783

Fonte: elaborado pelos autores.

Na Tabela 6, agrupamos os tempos de execução dos algoritmos das Tabelas 1, 2, 3, 4 e 5, com a finalidade de comparar os tempos de complexidade computacional dos cinco algoritmos.

Tabela 6 – Comparativo dos tempos em segundo (s) dos quatro algoritmos estudados.

Algoritmos	2	8	64	128	512	1024
Usual (u)	0.000030	0.000648	0.175758	1.393882	103.128729	840.871537
Recursivo (r)	0.000218	0.002130	0.592232	6.292380	335.786021	2689,499944
Winograd (w)	0.000013	0.000672	0.463794	1.547172	105.914162	–
Strassen (s)	0.000215	0.001467	0.479345	3.352598	182.802909	1229.282250
Strassen-Winograd (sw)	0.000081	0.00073	0,107	1.524	140.783	1000.19783

Fonte: elaborado pelos autores.

Na Tabela 8 organizamos as variações percentuais entre os tempos de execução (complexidade temporal) dos quatro algoritmos estudados, com a perspectiva de apontarmos evidências entre os mesmos.

[3] desenvolveu o algoritmo `StrassenRecursive.ipynb` que efetua o produto matricial de duas matrizes randômicas $2^n \times 2^n$. Os tempos de complexidade estão registrados na Tabela 7.

Tabela 7 – Algoritmos Usual, Recursivo e Strassen e os respectivos tempos de complexidade computacional.

Algoritmos	2^1	2^3	2^6	2^7	2^9	2^{10}
Nmf (u)	0.000145	0.000125	0.000252	0.000634	0.010726	0.073312
Rumf (r)	0.000146	0.002036	1.716001	7.746499	552.8518	4478.807539
RSmf (s)	0.000145	0.001817	0.664369	5.938602	247.9660	1737.289177

Fonte: elaborado pelos autores.
Nota: Numpy matmul finished (Nmf), Recursive usual multiplication finished (Rumf) e Recursive Strassen multiplication finished (RSmf).

Ao compararmos as complexidades entre algoritmos e a ordem das matrizes da Tabela 6 com a Tabela 7, percebemos que os tempos apresentam as mesmas ordens de grandezas. Realizamos este procedimento com a finalidade de verificarmos o quanto os algoritmos desenvolvidos pelos autores está em concordância com o apresentado por [3].

A Tabela 8, construída a partir das comparações entre os tempos dos algoritmos da Tabela 6. Por exemplo, para calcular a variação de complexidade do algoritmo usual (u) em relação ao recursivo (r) de uma matriz 2×2, utilizamos a seguinte fórmula:

$$u \longrightarrow r = \frac{r_2 - u_2}{r_2} \cdot 100, \tag{3.3}$$

sendo que r_2 e u_2 são respectivamente, os tempos de complexidades dos algoritmos recursal e usual processaram os produtos para as mesmas matrizes.

$$u \longrightarrow r = \frac{0.000218 - 0.00003}{0.000218} \cdot 100 = 86\%,$$

isto significa que a complexidade computacional aumentou em 86%, quando compara-se o u com o r . Do mesmo modo, todos as células da primeira linha da Tabela 8 foram preenchidas. Analogamente, com a mesma relação e as devidas adequações preenchemos as demais células da referida tabela.

Tabela 8 – Variação percentual dos quatro algoritmos estudados em referência à Tabela 6.

Variação	2	8	64	128	512	1024
u $\longmapsto$ r	86	70	70	78	69	69
u$\longmapsto$ w	−131	4	62	10	3	*
u$\longmapsto$ s	86	56	63	58	44	32
u$\longmapsto$ s w	63	-216	-64	9	27	16
r $\longmapsto$ w	−1577	−217	−28	−307	−217	*
r$\longmapsto$ s	−1	−45	−24	−88	−84	-119
r $\longmapsto$ sw	−169	−939	−453	−313	−139	−169
w $\longmapsto$ s	94	54	3	54	42	100
w $\longmapsto$ sw	84	−228	−333	−2	25	100
s $\longmapsto$ sw	−165	−616	−348	−120	−30	−23

Fonte: elaborado pelos autores.
Nota: * Não foi possível calcular esta porcentagem de processamento do algoritmo de Winograd para a matriz randômica 1024×1024.

Na Tabela 8, os valores com o sinal negativo (−) indicam que o algoritmo da ponta da seta obteve vantagem de complexidade computacional em relação ao que está ao início da seta.

Por conseguinte, o algoritmo de Strassen-Winograd (sw) obteve vantagem quase que na totalidade dos casos medidos.

Analogamente à Tabela 8, construímos a Tabela 9 com as variações de tempo de complexidade.

Tabela 9 – Variação percentual dos três algoritmos estudados em referência à Tabela 7.

Variação	2^1	2^3	2^6	2^7	2^9	2^{10}
u $\longmapsto$ r	1	1529	680853	1691892	5154215	6109142
u $\longmapsto$ s	0	1354	263538	1274278	2311722	2369620
r $\longmapsto$ s	−1	−11	−61	−25	−55	−61

Fonte: elaborado pelos autores.

Na Tabela 9, o algoritmo $r \longmapsto s$ obteve vantagem de complexidade computacional em relação as outras duas variações.

4 Conclusão

Neste trabalho, realizamos um estudo sobre complexidade algarítmica com foco na análise do desempenho de algoritmos em termos de tempo e espaço.

Iniciamos com uma exploração das multiplicações de matrizes, destacando a complexidade computacional e apresentando o Algoritmo 1, que será usado para avaliar essa complexidade.

Na Subseção 2.1, introduzimos o conceito do algoritmo de dividir e conquistar.

Na Seção 3, concluímos o trabalho apresentando os resultados e discussões, onde descrevemos o algoritmo de Winograd como uma alternativa econômica para a multiplicação de matrizes, desde que atendam a critérios específicos. Além disso, discutimos o algoritmo de Strassen, desenvolvido por Volker Strassen em 1969, e mencionamos o algoritmo Strassen-Winograd, que representa um aperfeiçoamento em termos de complexidade numérica.

De um modo geral, a partir da análise da Tabela 8, concluímos que a vantagem de algoritmo em relação à complexidade ocorreu entre Winograd (w) em relação ao recursivo (r) (na linha 4) - isto é, o Winograd utilizou menos tempo de que o recursivo para efetuar a multiplicação

entre as matrizes de mesma dimensão $2^n \times 2^n$, para $n = 1, 3, 6, 7, 9, 10$; e, também do Strassem (s) para o recursivo (r) (linha 5), em que o algoritmo de Strassen utilizou menos tempo para executar as mesmas multiplicações.

O Algoritmo 1 para multiplicação de matrizes não é o melhor possível: na Seção 3.2 vimos que o algoritmo proposto por Strassen (1969) precisa somente $n^{\log_2 7} \approx n^{2.807}$ multiplicações. Este algoritmo possui desempenho real melhor somente para n grande (no caso de Strassen $n \approx 700$.

Analogamente à Tabela 9, o algoritmo de Strassen (s) obteve vantagem em relação ao recursivo (r) para todas os produtos de matrizes (linha 3) - em relação a ordem direta.

Estes tópicos fornecerão uma base sólida e teórica para futura pesquisa no assunto.

Contribuições

Todos os autores contribuíram substancialmente na concepção e/ou no planejamento do estudo; na obtenção, análise e/ou interpretação dos dados; na redação e/ou revisão crítica; e aprovaram a versão final a ser publicada.

Orcid

Tomy Felixon https://orcid.org/0000-0002-3792-9126
João Socorro Pinheiro Ferreira https://orcid.org/0000-0002-3711-3602

Referências

1. CORMEN, Thomas H.; LEISERSON, Charles E.; RIVEST, Ronald L.; STEIN, Cliford. **Introduction to algorithms.** 3.ed. MIT press, 2022.
2. KNUTH, Donald E. **The Art of Computer Programming: Seminumerical Algorithms**, Volume 2. Addison-Wesley Professional, 2014.
3. SAA, Alberto. **Algoritmos subc´ubicos para multiplica¸c ao matricial**. Campinas, SP: IMECC, 2023. Disponível em: https://vigo.ime.unicamp.br/mt404/EP3.pdf.
4. STRASSEN, Volker. Gaussian elimination is not optimal. **Numerische mathematik**, v. 13, n. 4, p. 354-356, 1969.

© INTERMATHS
CC BY 4.0