ANÁLISIS Y DISEÑO DE ALGORITMOS CC371-A
Prof. Jaime Osorio Ubaldo

DIVIDE Y VENCERÁS
MULTIPLICACIÓN DE MATRICES

—

Por: Guillermo Ronie Salcedo Alvarez

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix}, \quad \mathbf{B} = \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1p} \\ b_{21} & b_{22} & \cdots & b_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{np} \end{pmatrix}$$

DISEÑO APLICANDO DIVIDE Y VENCERÁS

Al momento de diseñar un algoritmo de multiplicación de matrices, nos podemos encontrar ante la definición misma de ella, se van multiplicando los elementos de las filas de una matriz *A* por los elementos de las columnas de la otra matriz *B*. Obteniendo así una nueva matriz *C* de la siguiente forma:

$$\mathbf{C} = \begin{pmatrix} a_{11}b_{11} + \cdots + a_{1n}b_{n1} & a_{11}b_{12} + \cdots + a_{1n}b_{n2} & \cdots & a_{11}b_{1p} + \cdots + a_{1n}b_{np} \\ a_{21}b_{11} + \cdots + a_{2n}b_{n1} & a_{21}b_{12} + \cdots + a_{2n}b_{n2} & \cdots & a_{21}b_{1p} + \cdots + a_{2n}b_{np} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1}b_{11} + \cdots + a_{mn}b_{n1} & a_{m1}b_{12} + \cdots + a_{mn}b_{n2} & \cdots & a_{m1}b_{1p} + \cdots + a_{mn}b_{np} \end{pmatrix}$$

Por lo que se podría pensar en resolver al realizar tres iteraciones, yendo por cada cada elemento de las filas de A, por cada elemento de las columnas de B, para un respectivo elemento de la matriz C. Sin embargo, al implementarlo de esta manera caeríamos ante el paradigma de diseño de algoritmos “fuerza bruta”, y tendríamos una complejidad **$O(n^3)$**.

Por este motivo, se busca optimizar por medio del paradigma “divide y vencerás”:

Bajo este paradigma, el algoritmo a utilizar se desarrollará en esta sección. Este algoritmos se aplicará únicamente entre dos matrices cuadradas de igual tamaño $2^n \times 2^n$. Y lo que iremos haciendo será dividir las matrices en 4 submatrices de tamaño $2^{n-1} \times 2^{n-1}$. Luego de ello, se irán realizando las multiplicaciones de submatrices recursivamente. Sea la multiplicación de matrices $A \times B = C$.

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

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

Cabe destacar que si las matrices que queremos multiplicar no son de tamaño $2^n \times 2^n$, buscaremos esta forma al rellenar con ceros hasta obtenerlas.

Luego, el pseudocódigo sería la siguiente función que tomará como parámetros dos matrices y el tamaño de las matrices.

```
multMatrix(matrix A, matrix B, int n)
	si n = 1:
		matrix C de 1x1
		C[0][0] = A[0][0] · B[0][0]
		devuelve C
	sino
		half = n / 2
		Obtener las 4 submatrices de A (Aij)
		Obtener las 4 submatrices de B (Bij)
		matrix Cij de
		C11 = multMatrix(A11, B11, half) + multMatrix(A12, B21, half)
		C12 = multMatrix(A11, B12, half) + multMatrix(A12, B22, half)
		C21 = multMatrix(A21, B11, half) + multMatrix(A22, B21, half)
		C22 = multMatrix(A21, B12, half) + multMatrix(A22, B22, half)

	matrix C de half x half
	Para i = 0 hasta half
		Para j = 0 hasta half
			Llenar la matriz C con las submatrices
			C11, C12, C21, C22
	Devuelve C
fin-multMatrices
```

El pseudocódigo iniciará con nuestro caso base, en donde el tamaño de las matrices insertadas ambas son de tamaño igual a 1. En ese caso, se realizará el producto de dichos elementos y se devuelve dicha matriz de 1x1 con tal valor.

Caso contrario, las matrices sean de mayor tamaño, definiremos half como la mitad del tamaño de nuestras matrices, esto nos indicará el tamaño de las submatrices. Inmediatamente, calculamos dichas submatrices Aij y Bij.

Una vez tengamos las submatrices, se realizan las operaciones que se mostraron previamente. Tratamos cada submatriz como si fuesen elementos de una matriz 2x2. Y guardamos dichas operaciones en Cij. (i, j = 1, 2).

Finalmente, combinamos todos los resultados que obtengamos en una matriz C de tamaño half x half. En donde los elementos a insertar serán aquellos que obtengamos de las operaciones entre las submatrices (Cij).

COMPLEJIDAD DEL ALGORITMO

A continuación se presenta el desarrollo de cómo obtener la complejidad Big O del algoritmo “divide y vencerás” de la multiplicación de matrices. Para ello, partiremos por obtener el costo temporal de la misma:

Tendremos nuestra función como un T(n), el costo temporal, que dependerá únicamente de n, el tamaño de las matrices. Y, sabemos, que para n igual a 1, el costo temporal será una constante; dado el pseudocódigo tenemos T(1) = 4.

Ahora, procedemos a obtener el costo temporal para el caso contrario. Es necesario notar que el proceso de obtener las submatrices tomará un tiempo T_1 y el de sumar matrices será otro tiempo T_2. De este modo, el costo temporal nos queda como el costo temporal de aplicar 8 veces la función con un tamaño n / 2 más el resto de operaciones (asignaciones y loop final):

$$T(n) = 8T(n/2) + 2 + 2T_1 + 2 + 4T_2 + 4 + 1 + 4(n/2)^2$$

$$T(n) = 8T(n/2) + 2T_1 + 4T_2 + 9$$

Calculamos el tiempo temporal T_1 y T_2:

- Para el proceso de obtención de submatrices, basta saber que por cada submatriz se debe iterar 2 veces un tamaño n/2 de la matriz original. Por este motivo, sabemos que

  $$T_1(n) = a(n + k)^2 + b$$

  Dado el programa que se realizará en la siguiente sección, el costo temporal será exactamente:

  $$T_1(n) = n^2 + 2n + 16$$

- Para el proceso de suma de matrices, sabemos que es necesario iterar 2 veces, elemento por elemento de A y B, y sumarlos. Del mismo modo que para T_1, tendremos:

  $$T_2(n) = a'(n + k')^2 + b'$$

  Dado el programa que se realizará en la siguiente sección, el costo temporal será exactamente:

  $$T_2(n) = 2n^2 + 2$$

Reemplazando T_1 y T_2 en la expresión T(n):

$$T(n) = 8T(n/2) + 2(n^2 + 2n + 16) + 4(2n^2 + 2) + 9$$

$$T(n) = 8T(n/2) + 10n^2 + 4n + 49$$

De nuestro algoritmo, si ingresamos matrices de tamaño n, dividiremos entre dos hasta encontrar un n = 1. Supondremos que tras k divisiones encontraremos dicho n.

$$n / 2^k = 1 \rightarrow k = \log_2 n$$

Luego, acotamos nuestro T(n), y después de las k divisiones obtendremos

$$T(n) < 8T(n/2) + 11n^2$$

$$T(n) < 8^2T(n/4) + (11 + 11/4)n^2$$

...

$$T(n) < 8^kT(n/2^k) + mn^2$$

$$T(n) < 8^{\log_2 n}(4) + mn^2$$

$$T(n) < 4n^3 + mn^2$$

Por lo tanto, nuestro algoritmo es del orden **$O(n^3)$**.

Una manera más sencilla de calcular la complejidad de este algoritmo es la siguiente:

- Encontramos que tanto el costo temporal de obtener las submatrices como el de sumar matrices son del orden $O(n^2)$. Por lo que podemos reescribir T(n):

  $$T(n) = 8T(n/2) + O(n^2)$$

- Aplicamos el teorema maestro (*master theorem*) que nos indica que para alguna forma recursiva $T(n) = aT(n/b) + f(n)$, $a \geq 1$, $b > 1$.

  Si $f(n) = O(n^c)$ con un $c < \log_b a$ entonces, $T(n) = O(n^{\log_b a})$.

  Como $f(n) = O(n^2)$, entonces, se verifica que $c = 2 < \log_2 8 = 3$, y se puede calcular:

  $$T(n) = O(n^{\log_2 8}) = O(n^3).$$

PROGRAMA DEL ALGORITMO

La implementación del algoritmo “divide y vencerás” para la multiplicación de matrices se realizó en el lenguaje de programación C++. En él, se nos solicitará las dimensiones de nuestras matrices a multiplicar. Se validará si es posible la multiplicación, y de ser así, se ingresan los datos y se buscará que las matrices sean de tamaño $2^n \times 2^n$. De no serlo, se rellenará con ceros hasta encontrar dicha forma. Se realiza la multiplicación respectiva y se muestran los resultados en consola.

Fig. 1: Resultados en consola para dimensiones inválidas.

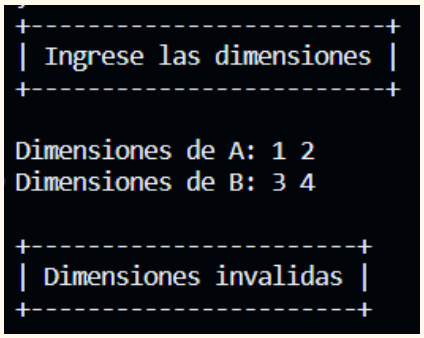
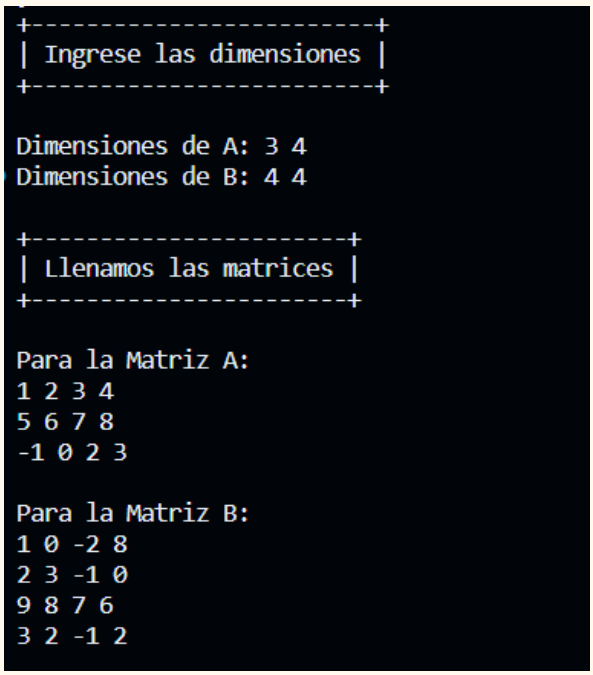
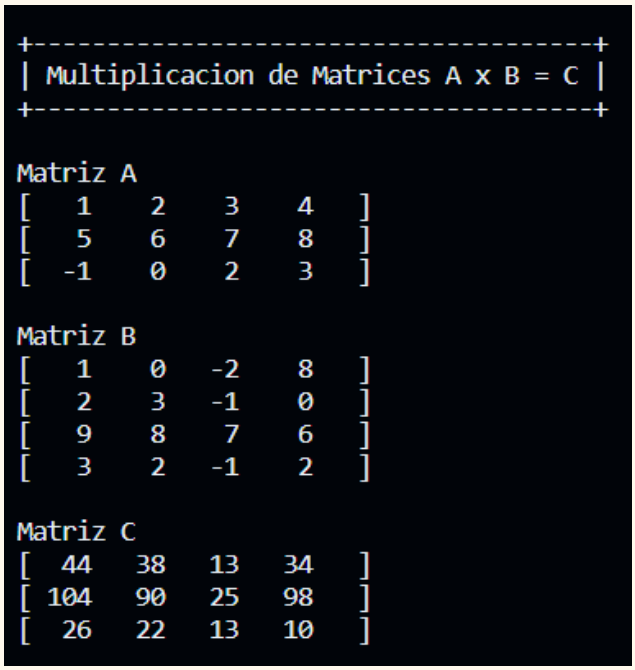


Fig. 2: Resultados en consola para dimensiones válidas.





UNA NUEVA ALTERNATIVA

Como hemos notado de este enfoque de “divide y vencerás” para la multiplicación de matrices, la complejidad que hemos obtenido es la misma que se obtuvo por el enfoque de “fuerza bruta”, por lo que se diría que no hemos encontrado una mejor alternativa. No obstante, el algoritmo de Strassen es un enfoque similar al de “divide y vencerás” en el que se construyen las siguientes 7 matrices para nuestro algoritmo:

- $M_1 = (A_{11} + A_{22}) \times (B_{11} + B_{22})$
- $M_2 = (A_{21} + A_{22}) \times B_{11}$
- $M_3 = A_{11} \times (B_{12} - B_{22})$
- $M_4 = A_{22} \times (B_{21} - B_{11})$
- $M_5 = (A_{11} + A_{12}) \times B_{22}$
- $M_6 = (A_{21} - A_{11}) \times (B_{11} + B_{12})$
- $M_7 = (A_{12} - A_{22}) \times (B_{21} + B_{22})$

Luego, la multiplicación $A \times B = C$ nos queda como:

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} M_1 + M_4 - M_5 + M_7 & M_3 + M_5 \\ M_2 + M_4 & M_1 - M_2 + M_3 + M_6 \end{bmatrix}.$$

De donde se nota que solo utilizaremos 7 multiplicaciones a comparación de las 8 del algoritmo que propusimos. De ahí se desprende que nuestra nueva forma recursiva es de la forma:

$$T(n) = 7T(n/2) + O(n^2)$$

Y aplicando el teorema maestro, encontraremos que la complejidad de este nuevo algoritmo es $O(n^{\log_2 7}) = O(n^{2.8074})$. Del cual se puede decir que tiene una complejidad “asintóticamente mejor” que el $O(n^3)$.

Referencias

- Wikipedia contributors. (2023a). Strassen algorithm. Wikipedia. https://en.wikipedia.org/wiki/Strassen_algorithm.
- Wikipedia contributors. (2023b). Master theorem (analysis of algorithms). Wikipedia. https://en.wikipedia.org/wiki/Master_theorem_(analysis_of_algorithms).