Escritura del problema de la multiplicación matricial

David Enrique Palacios García[1] Karen Sofia Coral Godoy[1]

[1]Departamento de Ingeniería de Sistemas, Pontificia Universidad Javeriana
Bogotá, Colombia
{david_palacios, corallg_ksofia}@javeriana.edu.co

1 de septiembre de 2022

Resumen

En este documento se presenta la formalización del problema de la multiplicación matricial, junto con la descripción de un algoritmo de programación dinámica que lo soluciona. Además, se presenta un análisis experimental de este algoritmo. **Palabras clave:** matriz, algoritmo, formalización, experimentación, programación dinámica.

Índice

1. Introducción

Las matrices son útiles para resolver muchos tipos de problemas, que pueden derivar desde matemáticas y física, hasta finanzas y economía. Lo anterior sin dejar de lado la computación gráfica, donde se pueden generar imágenes a partir de este tipo de estructura de datos. A menudo, estas son utilizadas en videojuegos, animaciones de computadora, captura de video, edición de efectos especiales, etc; Es importante destacar una operación básica en una composición de matrices, la multiplicación. Para realizar esta operación, es necesario conocer la forma óptima de agrupar las matrices para minimizar la cantidad de multiplicaciones escalares. Este taller busca solucionar el problema previamente mencionado, utilizando una estrategia de programación dinámica, con el objetivo de presentar: la formalización del problema (sección 2), la escritura formal del algoritmo (sección 3) y un análisis experimental del mismo (sección 4).

2. Formalización del problema

Cuando se piensa en *multiplicar una serie de matrices* la solución inmediata puede ser muy simplista: inocentemente, se piensa en multiplicar cada par de matrices con la siguiente en la secuencia hasta haber multiplicado todas las matrices. Sin embargo, con un poco más de reflexión, hay tres preguntas que pueden surgir:

1. ¿Todas las matrices se pueden multiplicar?
2. ¿Qué restricciones hay para multiplicar matrices?
3. ¿Existe una manera de acelerar este proceso?

Para iniciar, recordemos que para multiplicar las matrices A y B, es necesario que la cantidad de columnas de `A` sea la misma cantidad de filas de `B`. Esta operación matricial no es conmutativa, lo que significa que $AxB \neq BxA$. El resultado de esta operación es una matriz C cuya cantidad de filas es igual a las filas de A y cantidad de columnas igual a columnas de B.

No obstante, las matrices sí cumplen con una propiedad asociativa, esto es, $(A_1)(A_2A_3) = (A_1A_2)(A_3)$, en otros términos, se pueden asociar de distintas maneras y todas estas asociaciones diferentes entre sí nos devolverán el mismo valor. Si se analiza con mayor detalle, entenderemos que la forma en que se colocan los paréntesis puede tener un fuerte impacto en el costo en qué se evalua el producto, en donde puede llegar a crecer exponencialmente, convirtiendo el problema en un proceso sumamente largo y complejo de resolver. Por ello, se busca encontrar el camino óptimo de realizar estas multiplicaciones escalares.

2.1. Definición del problema de la "multiplicación matricial"

Así, el problema de multiplicar una cadena de matrices de modo que se realicen el menor número de multiplicaciones escalares, se define a partir de:

1. Una secuencia de matrices $S = \langle A_1, A_2, ..., A_n \rangle$ donde $A_i \in R^{(r_i \times c_i)} \wedge \mathrm{r}_i = c_{i-1}$
2. $A_{i,k}A_{k+1,n} \equiv (A_1...A_k)(A_{k+1}...A_n)$ donde $1 \leq k \leq n$
3. Entonces, $M_{i,j}$ es el número óptimo de multiplicaciones escalares al agrupar las matrices $A_{i,j}$:

$$M_{i,j} = \begin{cases} 0 & ; \quad i = j \\ min_{i \leq k \leq j} \{M_{i,k} + M_{k+1,j} + m_{ikj}\} & ; \quad i \neq j \end{cases} \tag{1}$$

donde m_{ikj} es el número de multiplicaciones escalares para calcular $A_{i,k}A_{k+1,j}$.

Dado que: $A_iA_{i+1} \in R^{r_i \times c_{i+1}}$, $A_iA_{i+1}A_{i+2} \in R^{r_i \times c_{i+2}}, ..., A_{i,k} \in R^{r_i \times c_k}, A_{k+1,j} \in R^{r_{k+1} \times c_j}$ y $r_{k+1} = c_k$, entonces:

$$m_{ikj} = r_i c_k c_j \tag{2}$$

Como restricción se definirá como entrada una secuencia $D = \langle d_i : 0 \leq i \leq n \wedge d_i \in N \rangle$, donde d_0 es la cantidad de filas de la primera matriz, d_1 es la cantida de columnas de la primera matriz y la cantidad de filas de la segunda matriz y, en general, d_{i-1}, es la cantidad de filas de A_i y la cantidad de columnas de A_{i-1}, entonces se simplificaría:

$$m_{ikj} = d_{i-1}d_k d_j \tag{3}$$

Producir una cadena de caracteres W que indique la forma óptima de agrupar las matrices de S, para que el número de multiplicaciones escalares para encontrar el producto total de todas las matrices sea el mínimo.

- Entradas:
  - $D = \langle d_1, d_2, d_3, ..., d_n \rangle \mid d_{i-1} = r_i \in A_i \wedge d_{i-1} = c_i \in A_{i-1}$
- Salidas:
  - $W = \langle A_i \in S \rangle \mid W$ es una cadena de caracteres que cumple la ecuación (1).

3. Algoritmo de solución

3.1. Programación dinámica- Backtracking

La idea de este algoritmo es: calcular la cantidad de multiplicaciones escalares entre la matriz A_1 hasta la matriz A_n, calculando todas las posibles combinaciones y respetando la regla asociativa. Los valores se irán guardando en una matriz auxiliar M para evitar cálculos repetidos y una matriz adicional B para conocer los valores que generaron la solución.

Además, como se pretende evaluar hasta un tope de mil matrices, el algoritmo usando progrmación dinámica permite evitar el colapso de la pila de ejecución del procesador por usar la iteración en lugar de la recursión.

El resultado final (óptima cantidad de multiplicaciones escalares) quedará guardado en la casilla $[\ 1, |M|\ ]$ por lo que será necesario aplicar el *Backtracking* sobre la matriz B en búsqueda de la manera en que se deben agrupar las matrices, esto mediante la estrategia de dividir y vencer.

Algoritmo 1 Optimización de multiplicaciones escalares

Require: $D = \langle d_1, d_2, d_3, ..., d_n \rangle \mid d_{i-1} = r_i \in A_i \wedge d_{i-1} = c_i \in A_{i-1}$

```
1: procedure CadenasMatriciales(D)
2:     n ← |D| − 1
3:     M ← [n][n] = {0}
4:     B ← [n][n] = {−1}
5:     for i ← n − 1 to 1 step − 1 do
6:         for j ← i + 1 to n do
7:             q ← ∞
8:             m ← 1
9:             for k ← i to j − 1 do
10:                left ← M[i, k]
11:                right ← M[k + 1, j]
12:                x ← 0
13:                if i − 1 = 0 then
14:                    x ← D[ |D| ]
15:                else
16:                    x = D[i − 1]
17:                end if
18:                v ← left + right + (x ∗ D[k] ∗ D[j])
19:                if v < q then
20:                    q ← v
21:                    m ← k
22:                end if
23:            end for
24:            M[i, j] ← q
25:            B[i, j] ← m
26:        end for
27:    end for
28:    return M, B
29: end procedure
```

Algoritmo 2 Obtener cadena de matrices

```
1: procedure AGREGARPARENTESIS(B, i, j)
2:     if i = j then
3:         print(A_i)
4:     else
5:         print( '(' )
6:         q ← B[i, j]
7:         agregarParentesis(B, i, q)
8:         agregarParentesis(B, q + 1, j)
9:         print( ')' )
10:    end if
11: end procedure
```

Algoritmo 3 Generar cadenas matriciales

```
1: procedure IMPRIMIRCADENASMATRICIALES(D)
2:     M, B ← CADENASMATRICIALES(D)
3:     agregarParentesis(B, 1, |B|)
4: end procedure
```

3.1.1. Análisis de complejidad

Por inspección de código: Se puede observar que, el algoritmo más complejo es $CadenasMatriciales$, con hasta 3 ciclos anidados. Lo cierto es que por la naturaleza de la programación dinámica, se ahorran cálculos repetidos, por lo que se acelera el proceso y se busca evitar llenar la pila de ejecución del procesador. Esto nos resulta en una complejidad de $O(ijk)$, siendo esta una expresión cúbica. A pesar de que la solución es un poco demorada, se debe a la complejidad del problema en sí, puesto que se necesitan calcular todas las posibles combinaciones de multiplicaciones escalares.

3.1.2. Invariante

La invariante al igual que la de todos los algortimos de programación dinámica es que se asegura que la tabla en cada iteración se esta completando correctamente.

4. Análisis experimental

En esta sección se presentarán los resultados del experimento para confirmar el orden de complejidad de los algoritmos presentados en la sección 3 y la precisión de la solución.

4.1. 10 Números de matrices

Acá se presentan los experimentos cuando el algoritmo se ejecuta tomando como párametro de entrada, números de matrices diferentes para cada iteración.

4.1.1. Protocolo

1. Definir una muestra de 10 iteraciones o números de entrada.
2. El algoritmo se ejecutará 10 veces con un parámetro de entrada diferente entre 2 y 1000, que representará el número de matrices a multiplicar en cada iteración.
3. Se genera el gráfico necesario para realizar un análisis.

4.1.2. Experimentación

1. Se utilizó el programa *cadenasMatricialesBacktracking* al cual se le pasa una secuencia de 10 números que corresponden al número de matrices que se contemplan en cada iteración, estos son:

   - `3,12,44,63,85,115,364,543,754,929`

   Adicionalmente, el programa teniendo en cuenta el número de matrices, define internamente la secuencia D con $|D| + 1$ números aleatorios que pertenezcan al rango definido entre 1 y 100.

2. El programa logró finalizar sin complicaciones, con un tiempo aproximado de minuto y medio.

3. Se obtuvo las 10 parentizaciones óptimas de las iteraciones con el número de múltiplicaciones que se deben realizar para llegar a dicha solución (Ver tabla 1), logrando así crear la siguiente gráfica: (ver Figura 1)

Número de matrices	Mínimo de múltiplicaciones
3	77.700
12	146.255
44	280.056
63	473.145
85	505.326
115	254.867
364	962.908
543	1.290.649
754	1.808.228
929	2.285.030

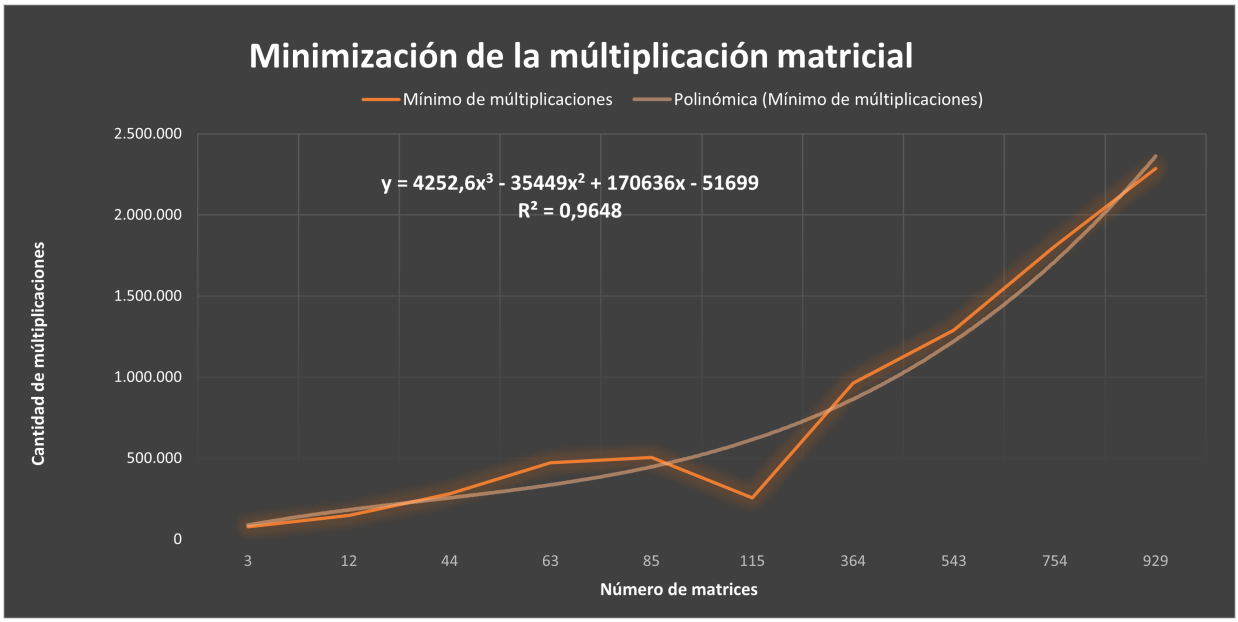

Figura 1: Cantidad mínima de múltiplicaciones

4.1.3. Análisis del experimento

Como se expresó previamente, el algoritmo de múltiplicación matricial tiene una complejidad de $O(ijk)$, sin embargo, en términos de tiempo de ejecución, es un proceso que llega a tomar mucho tiempo cuando la cantidad de matrices es muy alta. Así mismo, si todas las matrices son del mismo tamaño se alcanza una complejidad $\Theta(n^3)$. Esto pudo confirmarse, utilizando herramientas de Microsoft Excel, en donde se aplicó una regresión cúbica, hallando que la línea de tendencia del algoritmo, su ecuación y valor R^2 estaban muy cercanos a 1, lo cual indica que se empleó el modelo más preciso, y por ende se reafirma la complejidad algorítmica de la solución presentada.

Por otro lado, para este experimento, se realizó la una prueba de escritorio de la iteración con un número de matrices más pequeña para poder tener una comparación o evaluación del algoritmo. Los resultados se expresan a continuación:

- `Iteración: 1`
- `Número de matrices: 3`
- `D: [21, 53, 37, 47]`
- `M:`

```
[0 52311 77700
 0 0 41181
 0 0 0]
```

- `B:`

```
[-1 1 1
 -1 -1 2
 -1 -1 -1]
```

- `Mínimo de múltiplicaciones: 77700`
- `Solución: ( ( A1 A2 ) A3 )`

Para una múltiplicación de 3 matrices, se tienen 2 posibles soluciones de agrupación:

1. (A1 (A2 A3)) → calculando (21*53*47) + (53*37*47) = 144478
2. ((A1 A2) A3) → calculando (21*53*37) + (21*37*47) = 77700

Con esta iteración se puede validar que el algoritmo funciona, ya que se agregaron los paréntesis teniendo en cuenta el mínimo de múltiplicaciones escalares entre sí. De la misma forma, se observa que la solución óptima corresponde al valor almacenado en la esquina superior derecha de la matriz M, cumpliendo con el análisis y planteamiento dado en el aula de clases para el algoritmo *BottomUp*.

5. Conclusiones

El experimento fue útil para concluir:

1. Al aumentar el número de elementos en la cadena de matrices, el número de formas distintas en que puede parentizarse el producto de dichas matrices también aumenta.
2. Para plantear una solución es importante considerar los diferentes tamaños de entrada, como en el caso del problema de múltiplicación matricial en donde, posiblemente usar un algoritmo inocente es razonable cuando trabajamos con un universo pequeño, sin embargo, para valores grandes, es la peor estrategia para determinar la parentización óptima de una composición de matrices.

3. La programación dinámica se puede aplicar en los problemas de optimización ya que se basa en la idea de recursividad. En esta, se resuelven problemas descomponiéndolos en subproblemas similares más pequeños y usando dichas soluciones parciales de los subproblemas para llegar a una respuesta final. Cada subproblema se resuelve sólo una vez y su resultado es almacenado, evitando el trabajo de recalcular cada vez que el subproblema es nuevamente encontrado.

4. En general, el tiempo de un algoritmo de programación dinámica depende del número de subproblemas y del número de opciones a revisar para cada subproblema.