Práctica 3: Multiplicación encadenada de matrices.

Acevedo Juárez Sebastián, Ávila Mejía Daniel Said

25 de Octubre de 2021

1. Introducción

En la siguiente práctica, haremos uso de nuestros conocimientos vistos en clase para comparar distintos métodos para dar solución a una multiplicación de secuencia de matrices para así decidir el mejor camino por tomar así como sus diferencias.

2. Marco teórico

Dada una secuencia de matrices, el problema consta en encontrar la manera más eficiente de multiplicar estas matrices. A pesar de que suene un tanto sencillo, el problema radica más en las decisiones que tomemos más que en el proceso. Sabemos que la multiplicación de una secuencia de matrices es asociativa. En otras palabras, no importa como apliquemos los paréntesis en el producto, el resultado es el mismo:

Dadas las matrices A, B, C y D:

- $((A \times B) \times C) \times D$
- $(A \times (B \times C)) \times D$
- $A \times ((B \times C) \times D)$
- $A \times (B \times (C \times D))$
- $(A \times B) \times (C \times D)$

De esta forma, nos es posible agrupar la secuencia en subsecuencias de multiplicación por pares sin cambiar su orden.

El problema a solucionar se encuentra en ser capaces de asociar estas matrices de tal manera que tengamos el menor número de operaciones aritméticas posibles. Todo esto se ve mejor reflejado en el siguiente ejemplo:

Dadas las matrices A(10 x 30), B(30 x 5) y C (5 x 60):

- (AB)C = (10×30×5) + (10×5×60) = 1500 + 3000 = 4500 operaciones
- A(BC) = (30×5×60) + (10×30×60) = 9000 + 18000 = 27000 operaciones

Podemos ver claramente como el orden de los paréntesis en el primer caso es menor. Esto hace que ese orden de los paréntesis sea el ideal.

3. Implementación

Para todos los programas se utilizaron las librerías *timeit*, *matplotlib* y *statistics* para tomar el tiempo de ejecución, realizar las gráficas correspondientes y tomar un tiempo promedio respectivamente.

3.1. Programación Dinámica

Para el caso de la programación dinámica, se implementó de la siguiente manera:

```
def multMatriz(dims):
    n = len(dims)
    c = [[0 for x in range(n + 1)] for y in range((n + 1))]
    for length in range(2, n + 1):
        for i in range(1, n - length + 2):
            j = i + length - 1
            c[i][j] = sys.maxsize
            k = i
            while j < n and k <= j - 1:
                cost = c[i][k] + c[k + 1][j] + dims[i - 1]
                * dims[k] * dims[j]
                if cost < c[i][j]:
                    c[i][j] = cost
                k = k + 1
    return c[1][n - 1]
```

3.2. Algoritmo Voraz

En el caso de método voraz, se implementó de la siguiente manera:

```
def greedyMatriz(p, i, j):
    if i == j:
        return 0
    min = sys.maxsize
    for k in range(i, j):
        count = (greedyMatriz(p, i, k)
                 + greedyMatriz(p, k + 1, j)
                 + p[i-1] * p[k] * p[j])
        if count < min:
            min = count
    return min
```

3.3. Very Fast Approximation of the Matrix Chain Product Problem.

En el artículo presentado en la práctica se nos presenta una solución al problema desde un enfoque orientado a la triangulación de un polígono convexo.

3.3.1. Resumen

Para poder entender mejor el tema, es necesario explicar el problema de la triangulación de un polígono convexo. Definimos a un polígono convexo como cualquier polígono en que todos sus ángulos interiores son menores a los 90 grados.

Dado un polígono convexo representado por:

$$P = (v_1, v_2, ..., v_n)$$

Este tiene pesos positivos asociados a cada vértice, divídelo en triángulos de tal manera que el costo de la partición sea el menor posible. Cabe aclarar que el costo de la triangulación es la suma del costo de todos los triángulos en la partición. Ambos problemas (Polígono convexo y multiplicación encadenada de matrices) son equivalentes.

Por otro lado podemos reducir el problema a la triangulación de polígonos básicos. Estos se definen como polígonos que su segundo y tercer vértices más pequeños son "vecinos"del más pequeño. Para querer encontrar la triangulación óptima de un polígono, los podemos dividir en subpolígonos uniendo repetidamente el vértice más pequeño con los segundos y terceros vértices más pequeños.

El algoritmo de aproximación CHI-HU-SHING plantea lo siguiente: Sea V_m el vértice más pequeño de un polígono convexo y sea V_t un vértice con V_k y V_c como sus dos vecinos. El vértice V_t es "grande"sí:

$$\frac{1}{V_k} + \frac{1}{V_c} > \frac{1}{V_t} + \frac{1}{V_m}$$

Este algoritmo sigue los siguientes pasos:

1. Mientras haya un vértice "grande"V_t, salta V_t conectando sus dos vecinos y quita V_t del polígono.

2. Si ninguno de los vértices es "grande", el vértice más pequeño se conecta con todos los vértices.

A este algoritmo también se le conoce como *Algoritmo CHS* y puede llegar a ser implementado para tener una complejidad de $O(n)$.

Empleando estos conocimientos a la aproximación de una multiplicación de matriz en cadena, lo podemos plantear de la siguiente manera: El ïnput.º entrada es una secuencia de valores enteros $(d_0, d_1, ..., d_n)$de dimensiones de matriz. La salida, es un array denomidado $Brackets = (B_1, ..., B_{n-1})$ que guarda las posiciones de los pares de paréntesis que satisfacen el orden óptimo para realizar las operaciones. Para cada, $1 \leq i \leq n-1, Bi = (j, t)$ sí y sólo sí la expresión $(M_{j+1} \times M_{j+2} \times ... \times M_t)$ está presente en el orden óptimo. Es decir, hay un par de parénteis alrededor de M_{j+1} y M_t

Sea $M = M_1 \times ... \times M_n$ donde M_i is una matriz de $d_{i-1} \times d_i$ y sea $P = (v_0, v_1, ..., v_n)$ un polígono convexo donde su peso asignado a el vértice v_i es igual a d_i. Dado un orden de multiplicación de las matrices $M_1, ..., m_n$ definido por $Brackets = (B_1, ..., B_{n-1})$ con $B_{n-1} = (0, n)$ hay una triangulación correspondiente a P definida por el arreglo $arcos = (arco_1, ..., arco_{n-2})$ tal que $1 \leq i \leq n-2, 0 \leq j < t \leq n, arcos_i = (v_j, v_t)$ sí y sólo sí $B_i = (j, t)$

La complejidad de este proceso se denomina por $O(logn)$

3.3.2. Conclusiones de la lectura

Haciendo una comparativa entre todos los métodos, es evidente que el método de triangulación de un polígono convexo es mejor respecto a su complejidad en Big O. Como se menciona, tiene un acomplejidad de *O(log n)*, mientras que utilizando programación dinámica tenemos $O(n)$. Por último, el método voraz presenta una complejidad de $O(2n + 2nlogn)$.

4. Pruebas y Resultados

Para ambos métodos, se emplearon los siguientes casos:

1. A(50x30), B(30x20) y C(20x100)

2. A(10x200), B(200x300), C(300x50), D(50x90) y E(90x10)

3. A(1x2), B(2x3), C(3x4), D(4x5), E(5x6) y F(6x7)

Cabe aclarar que se registró un tiempo promedio para todos los casos.

Cuadro 1: Valores promedio en las pruebas

	Dinámico		Greedy	
	Costo	Tiempo	Costo	Tiempo
Caso 1	160000	0.15 ms	160000	0.2 ms
Caso 2	2206000	0.58 ms	2206000	18.31 ms
Caso 3	200	1.39 ms	200	189.2 ms

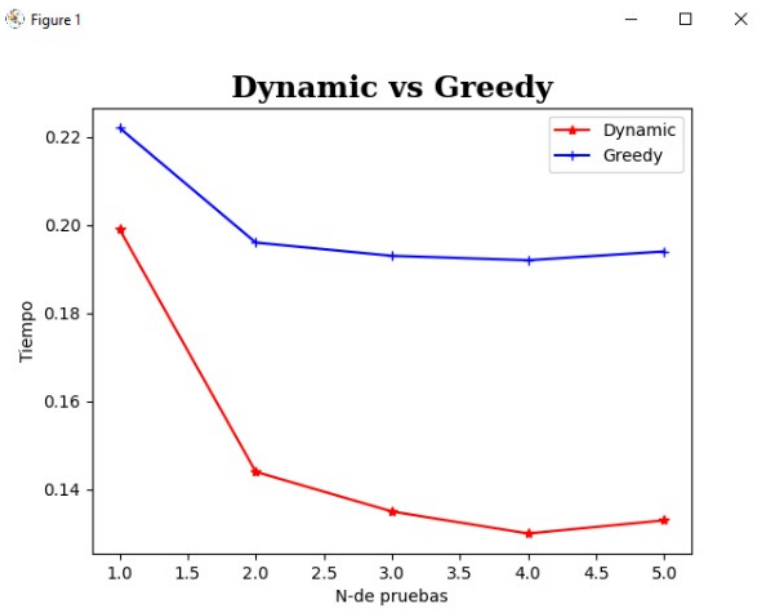

Figura 1: Gráfica comparativa entre programación dinámica y algoritmo voraz para el caso 1.

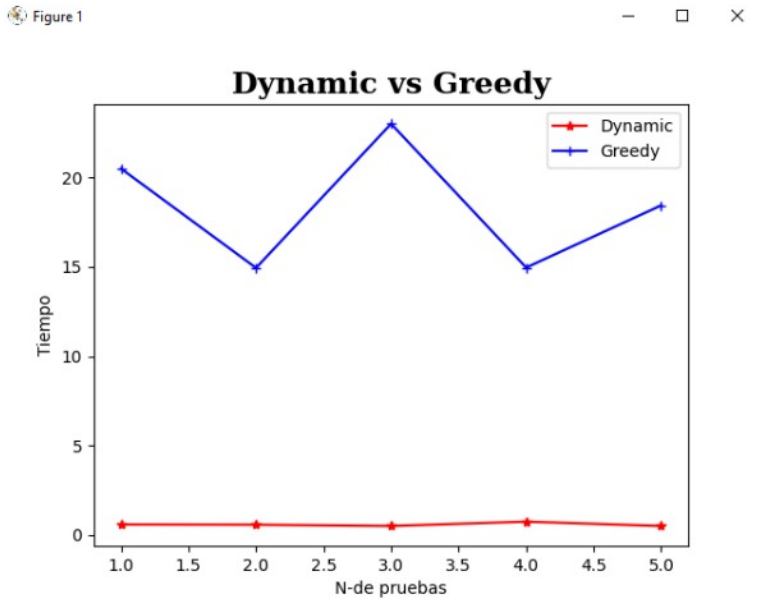

Figura 2: Gráfica comparativa entre programación dinámica y algoritmo voraz para el caso 2.

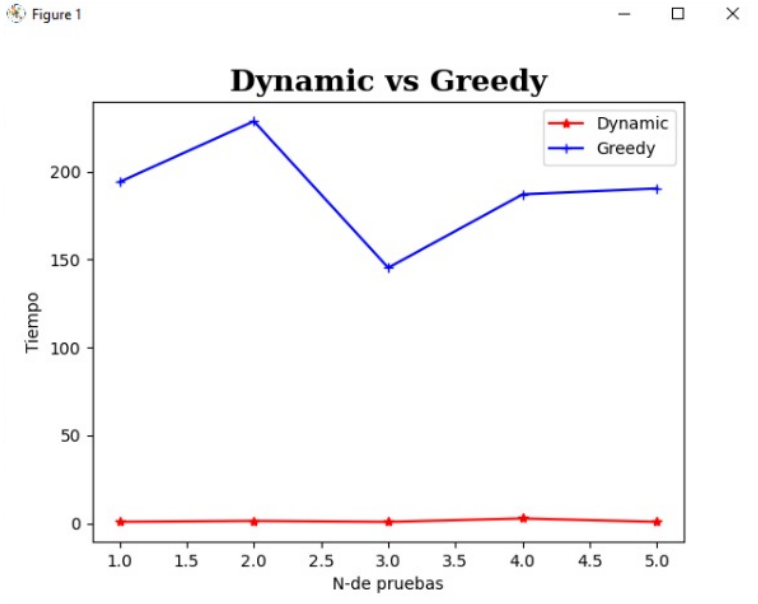

Figura 3: Gráfica comparativa entre programación dinámica y algoritmo voraz para el caso 3.

5. Conclusiones

Como es evidente, el algoritmo empleando programación dinámica es mucho más eficiente. En las gráficas, su eficiencia así como tiempo es en algunos casos muy superior. Sin embargo, si pudiéramos compararlo con la implementación con polígonos convexos, probablemente encontraríamos una mejor solución. Sin embargo, el existo no está asegurado, ya que existe un sesgo de error al cuál se expone el polígono convexo. Todo esto se menciona en el apartado de conlcusiones del texto.

Referencias

GeeksforGeeks. (2021, 7 agosto). Matrix Chain Multiplication — DP-8. https://www.geeksforgeeks.org/matrix-chain-multiplication-dp-8/