MULTIPLICACION ENCADENADA DE MATRICES

Juan Carlos Flores Mora, Miguel Alejandro Flores Sotelo, Sergio de Jesús Castillo Molano
Instituto Politécnico Nacional
Unidad Profesional Interdisciplinaria de Ingeniería Campus Tlaxcala
11 de Junio del 2024
jfloresm2202@alumno.ipn.mx

mfloress2200@alumno.ipn.mx

scastillom2200@alumno.ipn.mx

I. INTRODUCCIÓN

El problema de la multiplicación encadenada de matrices es un desafío fundamental en la ciencia de la computación y en la optimización de algoritmos. Se refiere al proceso de determinar la secuencia óptima de multiplicación de un conjunto de matrices para minimizar el número total de operaciones. Este problema tiene diversas aplicaciones en áreas como computación gráfica, procesamiento de imágenes, análisis de redes, aprendizaje automático, optimización de costos y bilogía celular.

El objetivo de este trabajo es desarrollar y analizar diferentes enfoques para resolver el problema de multiplicación encadenada de matrices. Para ello, se implementará el algoritmo de fuerza bruta, que consiste en calcular todos los costos posibles entre cada matriz y encontrar el mínimo costo; así como el enfoque de "top-Down", que divide el problema en subproblemas más pequeños y resuelve cada subproblema de manera recursiva. Sin embargo, a diferencia del enfoque recursivo básico, utiliza memorización para evitar recalcular los resultados de los subproblemas que ya se han resuelto. Esto ayuda a mejorar significativamente la eficiencia del algoritmo al evitar la repetición de cálculos redundantes.

II. MARCO TEÓRICO

En el algoritmo de multiplicación encadenada de matrices hay muchos pasos a seguir para poder llegar al resultado que necesitamos, los cuales son: Fuerza bruta, programación dinámica, Matplotlib, generación de código en Phyton .

A continuación, se explica el concepto de cada uno de los pasos para poder llegar a los resultados empleados y evaluados para el desarrollo de la práctica:

Fuerza bruta: Este método implica calcular la distancia entre cada par de puntos en el conjunto y seleccionar el par con la distancia mínima. Aunque su implementación es sencilla, su complejidad computacional es O(n2), donde n es el número de puntos en el conjunto.

Programación dinámica: es un enfoque eficiente para resolver problemas de optimización donde los subproblemas se superponen. Para resolver el problema de encontrar la distancia mínima entre cada par de puntos en el conjunto utilizando programación dinámica, primero ordenamos los puntos según su posición en el eje X. Luego, calculamos recursivamente las distancias mínimas entre los pares de puntos en cada mitad del conjunto y combinamos las soluciones para encontrar la distancia mínima global.

Técnicas de diseño algorítmico:

Programación Dinámica: Esta técnica es un método para reducir el tiempo de ejecución de un algoritmo mediante la utilización de subproblemas superpuestos y subestructuras óptimas.
Matplotlib: Es una biblioteca de Python ampliamente utilizada para crear visualizaciones estáticas, animadas e interactivas en 2D. Para este caso, se utiliza para la generación de gráficos a partir de arreglos bidimensionales que se emplean dentro de los programas, logrando así la representación de las pruebas con distintas multiplicaciones de matrices y visualizar el rendimiento para cada una de sus ejecuciones.
Programación en Python: Por medio de este lenguaje de programación, se toma en consideración la precisión de las operaciones de temporización, además de que el rendimiento del código es un poco más lento al ser Python un lenguaje interpretado. Por otro lado, se asegura que la implementación del código de como resultado las soluciones óptimas.

CÓDIGO FUERZA BRUTA

```
import random
import time
def construirSecuencia(s, n):
    secuencia = [[ 0 for i in range(n)] for i in range(n)]
    for i in range(1, n):
        secuencia[i][i] = f"A{i}"
    for l in range(2, n):
        for i in range(1, n - l + 1):
            j = i + l - 1
            k = s[i][j]
            secuencia[i][j] = f"({secuencia[i][k]}{secuencia[k
+ 1][j]})"
    return secuencia
#fuerza bruta
def multiplicacionMatricesOptima(p, n):
    m = [[0] * n for _ in range(n)]
    s = [[0] * n for _ in range(n)]
    # Calcular los costos óptimos
    for l in range(2, n):
        for i in range(1, n - l + 1):  # i es el índice de inicio de
la cadena
            j = i + l - 1  # j es el índice de fin de la cadena
            m[i][j] = 0  # Inicializado a 0
            min_cost = None  # Inicializar el costo mínimo
            for k in range(i, j):  # k es el índice para dividir la
cadena
                q = m[i][k] + m[k + 1][j] + p[i - 1] * p[k] * p[j]
                if min_cost is None or q < min_cost:  # Si
encontramos un costo menor, lo actualizamos
                    min_cost = q
                    s[i][j] = k
            m[i][j] = min_cost

    print("Multiplicacion optima de matrices: ", end="")
    secuencia = construirSecuencia(s, n)
    print(secuencia[1][n - 1])

    return m[1][n - 1]

def generarDimensionesAleatorias(n):
    dimensiones = [random.randint(1, 100) for i in range(n +
1)]
    return dimensiones

def main():
    random.seed(time.time())
    numMatrices = int(input("Ingrese el numero de matrices:
"))
    p = generarDimensionesAleatorias(numMatrices)
    print("Dimensiones generadas aleatoriamente de las
matrices:")
    for i in range(1, numMatrices + 1):
        print(f"A{i}: {p[i - 1]} x {p[i]}")
    minOperaciones = multiplicacionMatricesOptima(p,
numMatrices + 1)
    print(f"Numero total de operaciones optimo:
{minOperaciones}")

if __name__ == "__main__":
    main()
```

// Explicación del código

Función multiplicacionMatricesOptima: Esta función recibe como entrada un arreglo p que contiene las dimensiones de las matrices y el número n de matrices a multiplicar. Inicializa dos matrices m y s de tamaño n x n, donde m almacenará los costos mínimos y s almacenará las divisiones óptimas para multiplicar las matrices.

Luego, calcula los costos óptimos de multiplicación iterando sobre las subcadenas de matrices. Para cada subcadena, calcula todos los posibles cortes y encuentra el que minimiza el costo total de multiplicación. Almacena este costo mínimo en `m` y el índice del corte óptimo en `s`.

Función construirSecuencia: Esta función toma la matriz s generada por multiplicacionMatricesOptima y construye la secuencia óptima de multiplicación de matrices utilizando los índices almacenados en s.

Función generarDimensionesAleatorias: Genera un vector aleatorio de dimensiones para las matrices, donde cada dimensión es un número entero aleatorio entre 1 y 100.

Función main: La función principal solicita al usuario el número de matrices que se van a multiplicar. Luego, genera las dimensiones aleatorias para estas matrices y las imprime. Llama a multiplicacionMatricesOptima para encontrar la secuencia óptima de multiplicación y calcular el número total de operaciones óptimo. Finalmente, imprime la secuencia óptima de multiplicación y el número total de operaciones.

El algoritmo de fuerza bruta para multiplicación de matrices encuentra la secuencia óptima de multiplicación considerando todas las posibles divisiones y calculando el costo de cada una. Luego, selecciona la división que minimiza el costo total de multiplicación. Aunque este enfoque es simple, tiene una complejidad computacional de O (n^2), donde n es el número de matrices, lo que lo hace impracticable para grandes conjuntos de matrices.

CÓDIGO PROGRAMACIÓN DINÁMICA CON ENFOQUE TOP – DOWN

```
#bibliotecas necesarias para la generación de números
aleatorios
import random
import time

def printMCP(solucion, i, j):
    if i == j:
        print(f"A{i+1} ", end="")
    else:
        print("(", end="")
        printMCP(solucion, i, solucion[i][j])
        printMCP(solucion, solucion[i][j] + 1, j)
        print(")", end="")

def MCP(p, tabla, solucion, i, j):
    if i == j:
```

```
        return 0
    if tabla[i][j] is not None:
        return tabla[i][j]

    tabla[i][j] = float('inf')
    for k in range(i, j):
        q = MCP(p, tabla, solucion, i, k) + MCP(p, tabla,
solucion, k + 1, j) + p[i] * p[k + 1] * p[j + 1]
        if q < tabla[i][j]:
            tabla[i][j] = q
            solucion[i][j] = k
    return tabla[i][j]

def generarDimensionesAleatorias(n):
    return [random.randint(1, 100) for _ in range(n + 1)]

def main():
    random.seed(time.time())
    numMatrices = int(input("Ingrese el número de matrices:
"))
    p = generarDimensionesAleatorias(numMatrices)

    print("Dimensiones generadas aleatoriamente de las
matrices:")
    for i in range(1, numMatrices + 1):
        print(f"A{i}: {p[i - 1]} x {p[i]}")

    tabla = [[None for _ in range(numMatrices)] for _ in
range(numMatrices)]
    s = [[0 for _ in range(numMatrices)] for _ in
range(numMatrices)]

    MCP(p, tabla, s, 0, numMatrices - 1)

    print(f"Número total de operaciones óptimo:
{tabla[0][numMatrices - 1]}")
    print("Paréntesis óptimos:")
    printMCP(s, 0, numMatrices - 1)

if __name__ == "__main__":
    main()
```

//Explicación del código

Función printMCP: El objetivo de esta función es la impresión del acomodo de los paréntesis, siendo este la forma en que se tiene que desarrollar las multiplicaciones para obtener el menor número de operaciones posibles. Cuando i y j son iguales, significa que solo existe una sola matriz en la subsolución, por lo que simplemente se imprime el número de matriz correspondiente. Por otro lado, se imprime los paréntesis y dentro de ellos se hace dos llamadas recursivas a esta función: la primera que obtiene el primer multiplicando que pertenece dentro del paréntesis y la segunda llamada para el segundo elemento de la multiplicación, que bien cualquiera de los dos puede ser una matriz o una multiplicación anidada. Esta función recibe a la matriz solución, que contiene las parentización de la solución óptima; i y j, que marcan el inicio y final del arreglo en cuestión.

Función MCP: Esta función es la parte sustancial para la resolución del problema. Recibe como parámetros p, que son los tamaños de las matrices a multiplicar; la tabla de soluciones, i y j, quienes marcan el inicio y final del arreglo p y de la matriz tabla.

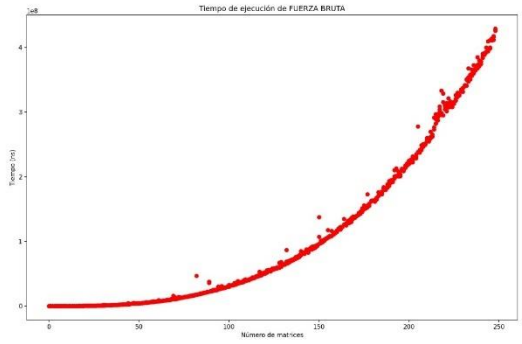
Inicialmente se tiene el caso base, cuando i==j, es decir, que el inicio es igual al final, por lo que quiere decir que solo hay un elemento en la división para (i, …, k) o (k+1, …, j), retornando un cero. Por otro lado, si el valor de la tabla en la posición de i y j es diferente de nulo, significa que ya se tiene la solución, así que simplemente se retorna el valor. Finalmente, en la posición i,j de la tabla se define un valor infinito, luego se inicializa un ciclo desde k=i hasta j-1, esto debido a que k es la pauta que permite el calculo de las posibles colocaciones de los paréntesis dependiendo del tamaño de la tabla que haya recibido la función en ese instante; en otras palabras, permite que los intervalos (i, …, k) y (k+1, …, j) puedan aumentar y disminuir su rango de matrices a evaluar. Si el número de operaciones obtenidos es menor a los registrados en la tabla con posición i, j, entonces almacena tanto el número de operaciones como la posición en donde la división de los paréntesis fue óptima. Una vez realizado todo el proceso, retorna el valor obtenido de la tabla en la posición i, j.

Función generarDimensionesAleatorias: El propósito de esta función es la generación de números aleatorios en un rango de 1 a 100, devolviendo así estos valores a la matriz en donde se hizo el llamado de la función.

Función main: Por último, pero no menos importante, se encuentra la función main. En ella se le solicita al usuario, primeramente, el número de matrices con las que se desea hacer el cálculo, luego se generan y almacenan los números aleatorios, generados con la función anterior, en la matriz p. Posteriormente, se imprime las matrices con sus respectivos tamaños y se crean las matrices bidimensionales tabla y s, establecidas ambas con un valor ya definido en todas sus celdas. Finalmente, se hace el llamado de la función, ingresando todos los parámetros ya mencionados, así como el valor de i y j, para luego imprimir los resultados: el número mínimo de operaciones y la colocación de los paréntes

III. RESULTADOS

GRAFICAS "Evaluación de la eficiencia de algoritmos de Fuerza Bruta y Top-down en función del tamaño de matrices vs tiempo"



GRAFICA 1.1 Figura 1 evaluación de eficiencia Tamaño de matrices vs tiempo (FUERZA BRUTA)

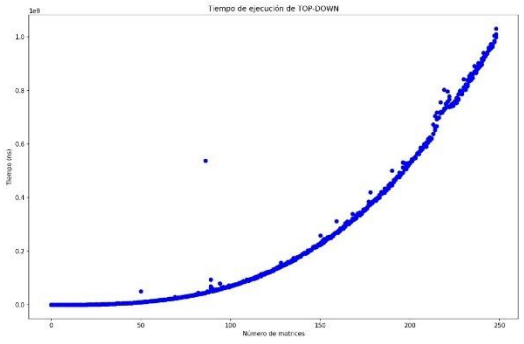
Eje X (horizontal): Representa el número de matrices

Eje Y (vertical): Representa el tiempo de ejecución en segundos.

Crecimiento no lineal: La gráfica muestra que el tiempo de ejecución crece de manera no lineal con el aumento del número de matrices. Este tipo de crecimiento sugiere una complejidad algorítmica alta, posiblemente exponencial o cuadrática.

Tendencia exponencial: La forma curvada de la gráfica indica que el tiempo de ejecución aumenta más rápidamente a medida que el número de matrices se incrementa, lo que es típico de un crecimiento exponencial.

Dispersión de puntos: Aunque hay una clara tendencia ascendente, hay variabilidad en los puntos de datos, lo que podría ser debido a diferencias en las características individuales de las matrices o variaciones en las condiciones de ejecución.



GRAFICA 1.2 Figura 2 evaluación de eficiencia Tamaño de matrices vs tiempo (TOP-DOWN)

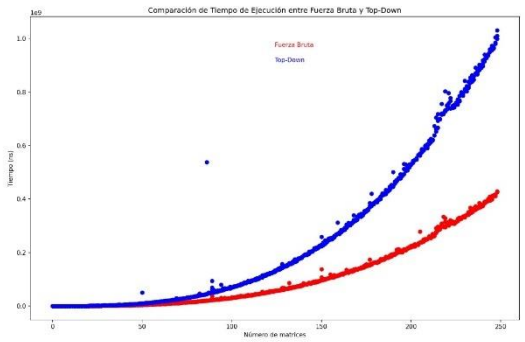
Eje X (horizontal): Representa el número de matrices

Eje Y (vertical): Representa el tiempo de ejecución en segundos.

Crecimiento Exponencial Moderado:
Al igual que la gráfica anterior, el tiempo de ejecución aumenta con el número de matrices, pero la pendiente de la curva es más moderada en comparación con la de la fuerza bruta. Esto sugiere que el algoritmo "TOP DOWN" es más eficiente.
Dispersión de Datos:
Hay cierta dispersión en los puntos de datos, especialmente en valores bajos y alrededor de los 100 matrices. Esto podría deberse a variaciones en la complejidad de las matrices o en las condiciones de ejecución del algoritmo.
Mejor Escalabilidad:
Comparando visualmente con la gráfica anterior de la fuerza bruta, el algoritmo "TOP DOWN" parece escalar mejor. El tiempo de ejecución es menor para el mismo número de matrices.

COMPARACION:

La gráfica del algoritmo "TOP DOWN" muestra una mejora en la eficiencia y escalabilidad respecto a la fuerza bruta, con un aumento exponencial del tiempo de ejecución más moderado y menos pronunciado. Esto indica que "TOP DOWN" es una mejor opción para manejar un mayor número de matrices de manera más eficiente



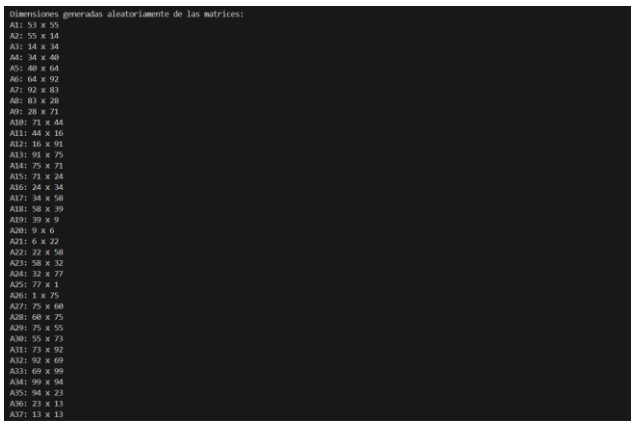
GRAFICA 1.3 Figura 3 evaluación de eficiencia Tamaño de matrices vs tiempo (FUERZA BRUTA VS TOP-DOWN)

La gráfica muestra claramente dos curvas, una para cada algoritmo.
La curva roja representa el tiempo de ejecución del algoritmo de Fuerza Bruta, mientras que la curva azul representa el tiempo de ejecución del algoritmo Top-Down. Ambas curvas muestran un crecimiento exponencial, pero la pendiente de la curva de Fuerza Bruta es más pronunciada que la del algoritmo Top-Down.

Para problemas que implican un gran número de matrices, se recomienda utilizar el algoritmo Top-Down debido a su mejor eficiencia y escalabilidad. La Fuerza Bruta puede ser

adecuada para problemas más pequeños o cuando la simplicidad del código es una prioridad, pero su rendimiento se degrada rápidamente con el tamaño del problema.

//PRUEBAS DE FUNCIONAMIENTO



V. CONCLUSIÓN

Al comparar las dos gráficas de tiempo de ejecución para los algoritmos de fuerza bruta y "TOP DOWN". El algoritmo "TOP DOWN" es significativamente más eficiente que el algoritmo de fuerza bruta. Esto se evidencia en el tiempo de ejecución considerablemente menor del "TOP DOWN" para el mismo número de matrices. Ambas gráficas muestran un crecimiento exponencial en el tiempo de ejecución a medida que el número de matrices aumenta. Sin embargo, el crecimiento es menos pronunciado en el caso del algoritmo "TOP DOWN," indicando una mejor escalabilidad. El algoritmo "TOP DOWN" demuestra una mejor capacidad de manejar un mayor número de matrices sin que el tiempo de ejecución se incremente de manera tan drástica como en el caso de la fuerza bruta. Esto lo hace más adecuado para problemas de mayor escala. Ambos algoritmos muestran dispersión en los tiempos de ejecución, lo que podría deberse a variaciones en las características de las matrices. Sin embargo, el algoritmo de fuerza bruta muestra una dispersión más amplia, especialmente en valores altos de número de matrices. Sin embargo, es importante estar consciente de posibles anomalías y variaciones en el rendimiento que pueden ocurrir en ciertos casos específicos.

IV. REFERENCIAS

[1] Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (3rd ed.). MIT Press..

[2] Knuth, D. E. (1997). *The Art of Computer Programming, Volume 3: Sorting and Searching* (2nd ed.). Addison-Wesley.

[3] Aho, A. V., Hopcroft, J. E., & Ullman, J. D. (1983). *Data Structures and Algorithms*. Addison-Wesley.

[4] Skiena, S. S. (2008). *The Algorithm Design Manual* (2nd ed.). Springer.

[5] Weiss, M. A. (2012). *Data Structures and Algorithm Analysis in C++* (4th ed.). Pearson.