

PRÁCTICA 8: MULTIPLICACIÓN DE UNA SUMA DE MATRICES.

Hernández Castellanos César Uriel, Aguilar Garcia Mauricio

Escuela Superior de Cómputo
Instituto Politécnico Nacional, México
uuriel12009u@gmail.com, mauricio.aguilar.garcia,90@gmail.com

Resumen: Se implementa el algoritmo de multiplicación de una secuencia de matrices el cual utiliza la técnica "programación dinámica" que es el de descomponer en subproblemas y resolverlos de la mejor manera, y al regresar evaluar todas las soluciones para encontrar la óptima".

Palabras Clave: Algoritmo, Programacion dinámimca, Matriz, óptimo.

1. Introducción

La programación dinámica es una técnica matemática que se utiliza para la solución de problemas matemáticos seleccionados, en los cuales se toma un serie de decisiones en forma secuencial.

Proporciona un procedimiento sistemático para encontrar la combinación de decisiones que maximice la efectividad total, al descomponer el problema en etapas, las que pueden ser completadas por una o más formas (estados), y enlazando cada etapa a través de cálculos recursivos.

La programación dinámica no cuenta con una formulación matemática estándar, sino que se trata de un enfoque de tipo general para la solución de problemas, y las ecuaciones específicas que se usan se deben desarrollar para que representen cada situación individual. Comúnmente resuelve el problema por etapas, en donde cada etapa interviene exactamente una variable de optimización (u optimizadora).

La teoría unificadora fundamental de la programación dinámica es el Principio de Optimalidad, que nos indica básicamente como se puede resolver un problema adecuadamente descompuesto en etapas utilizando cálculos recursivos.

“Una política óptima tiene la propiedad de que, independientemente de las decisiones tomadas para llegar a un estado particular, en una etapa particular, las decisiones restantes deben constituir una política óptima para abandonar ese estado”. [1]

2. Conceptos Básicos

2.1. Cota ajustada asintótica

En análisis de algoritmos una cota ajustada asintótica es una función que sirve de cota tanto superior como inferior de otra función cuando el argumento tiende a infinito. Usualmente se utiliza la notación $\theta(g(x))$ para referirse a las funciones acotadas por la función $g(x)$. [2]

2.2. Cota inferior asintótica

En análisis de algoritmos una cota inferior asintótica es una función que sirve de cota inferior de otra función cuando el argumento tiende a infinito. Usualmente se utiliza la notación $\Omega(g(x))$ para referirse a las funciones acotadas inferiormente por la función $g(x)$. [2]

2.3. Cota superior asintótica

En análisis de algoritmos una cota superior asintótica es una función que sirve de cota superior de otra función cuando el argumento tiende a infinito. Usualmente se utiliza la notación de Landau: $O(g(x))$, Orden de $g(x)$, coloquialmente llamada Notación O Grande, para referirse a las funciones acotadas superiormente por la función $g(x)$. [2]

2.4. Algoritmos

2.4.1. Multiplicación de una secuencia de matrices

El producto de un número n de matrices es optimizable en cuanto al número de multiplicaciones escalares requeridas. A la hora de multiplicar una serie de matrices se puede elegir en que orden queremos realizar las multiplicaciones entre estas. Se pueden realizar en un orden cualquiera dada la propiedad asociativa de la multiplicación.

El orden en que se decide multiplicar no afecta al resultado, es decir, el resultado siempre es el mismo. La diferencia esta en el número de multiplicaciones que implica elegir un orden u otro. Al multiplicar dos matrices M_1 de tamaño $n \times m$ y M_2 de tamaño $m \times k$ el número de multiplicaciones escalares es $n \cdot m \cdot k$. La cantidad total de multiplicaciones será, la suma de todas las multiplicaciones que hacen falta para multiplicar cada submatriz obtenida como resultado con la siguiente en el orden escogido.

```

MATRIX-CHAIN-ORDER( $p$ )
1   $n \leftarrow \text{length}[p] - 1$ 
2  for  $i \leftarrow 1$  to  $n$ 
3      do  $m[i, i] \leftarrow 0$ 
4  for  $l \leftarrow 2$  to  $n$        $\triangleright l$  is the chain length.
5      do for  $i \leftarrow 1$  to  $n - l + 1$ 
6          do  $j \leftarrow i + l - 1$ 
7               $m[i, j] \leftarrow \infty$ 
8              for  $k \leftarrow i$  to  $j - 1$ 
9                  do  $q \leftarrow m[i, k] + m[k + 1, j] + p_{i-1}p_kp_j$ 
10                     if  $q < m[i, j]$ 
11                         then  $m[i, j] \leftarrow q$ 
12                              $s[i, j] \leftarrow k$ 
13  return  $m$  and  $s$ 

```

Figura 1: Algoritmo de secuencia de matrices

```

PRINT-OPTIMAL-PARENS( $s, i, j$ )
1  if  $i = j$ 
2      then print " $A$ " $_i$ 
3      else print "("
4          PRINT-OPTIMAL-PARENS( $s, i, s[i, j]$ )
5          PRINT-OPTIMAL-PARENS( $s, s[i, j] + 1, j$ )
6          print ")"

```

Figura 2: Algoritmo para imprimir la secuencia de matrices

3. Experimentación y Resultados

3.1. Multiplicación de una secuencia de matrices

- 1.- Implementar el algoritmo de la multiplicación de una secuencia de matrices.

```
void imprimeParenti(long ** s, long i, long j){
    //cout << i << ":" << j;
    if(i == j)
        cout << "A" << i+1 << " ";
    else{
        cout << "(";
        imprimeParenti(s, i, s[i][j]);
        cout << " * ";
        imprimeParenti(s, s[i][j]+1, j);
        cout << ")";
    }

    //return;
}

tablas secuenciamatrices(vector <long> P){
    long n = P.size()-1;
    tablas tab;

    tab.m = (long **) calloc((n), sizeof(long *));
    for(long i = 0; i < n; i++)
        tab.m[i] = (long *) calloc((n), sizeof(long));

    tab.s = (long **) calloc((n), sizeof(long *));
    for(long i = 0; i < n; i++)
        tab.s[i] = (long *) calloc((n), sizeof(long));

    for(long l = 2; l <= n; l++){
        for(long i = 0; i+l-1 < n; i++){
            long j = i + l - 1;
            tab.m[i][j] = LONG_MAX;
            for(long k = i; k < j; k++){
                long q = tab.m[i][k] + tab.m[k+1][j] + (P[i]*P[k+1]*P[j+1]) ;
                if(q < tab.m[i][j]){
                    tab.m[i][j] = q;
                    tab.s[i][j] = k;
                }
            }
        }
    }

    return tab;
}
```

Figura 3: Código de algoritmo

- (i). Como entrada el algoritmo tendrá n matrices A_i de tamaño $p_{i-1} * p_i$:
- (ii). Como salida, se mostrará la configuración de paréntesis que genera el algoritmo.

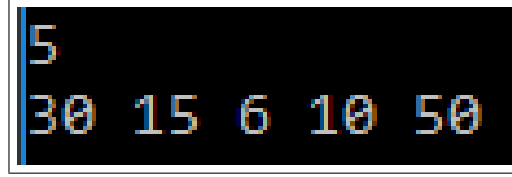


Figura 4: Entrada con n tamaños para las $n-1$ matrices junto con sus tamaños respectivos del ejemplo 1




Figura 5: Entrada con n tamaños para las $n-1$ matrices junto con sus tamaños respectivos del ejemplo 2

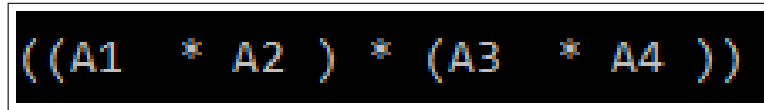


Figura 6: Salida del programa con la configuración de los paréntesis del ejemplo 1




Figura 7: Salida del programa con la configuración de los paréntesis del ejemplo 2

(iii) Además se mostrarán todas las configuraciones de paréntesis y se corroborará que la generada en el punto ii) en efecto, es la óptima.
 Por fuerza bruta, para un caso de $n=4$ y $p = \{30,15,6,10,50\}$, se tienen las siguientes combinaciones:

$$(A_1 A_2)(A_3 A_4) \Rightarrow \# \text{ de operaciones} = 14,700$$

$$A_1(A_2(A_3 A_4)) \Rightarrow \# \text{ de operaciones} = 30,000$$

$$A_1((A_2 A_3) A_4) \Rightarrow \# \text{ de operaciones} = 30,900$$

$$((A_1 A_2) A_3) A_4 \Rightarrow \# \text{ de operaciones} = 19,500$$

$$(A_1(A_2 A_3)) A_4 \Rightarrow \# \text{ de operaciones} = 20,400$$

Como se puede ver, para este ejemplo la configuración óptima, ya que solo hace 14,700 operaciones, es: $(A_1 A_2)(A_3 A_4)$ que coincide con lo obtenido en el programa como se muestra

en la Figura 6.

Ahora para un caso de $n=4$ y $p = \{30,25,40,100,10\}$, se tienen las siguientes combinaciones,

$$(A_1 A_2)(A_3 A_4) \Rightarrow \# \text{ de operaciones} = 82,000$$

$$A_1(A_2(A_3 A_4)) \Rightarrow \# \text{ de operaciones} = 57,500$$

$$A_1((A_2 A_3) A_4) \Rightarrow \# \text{ de operaciones} = 126,000$$

$$((A_1 A_2) A_3) A_4 \Rightarrow \# \text{ de operaciones} = 180,000$$

$$(A_1(A_2 A_3)) A_4 \Rightarrow \# \text{ de operaciones} = 205,000$$

Como se puede ver, para este ejemplo la configuración óptima, ya que solo hace 57,500 operaciones, es: $A_1(A_2(A_3 A_4))$ que coincide con lo obtenido en el programa como se muestra en la Figura 7.

4. Conclusiones

En esta práctica pudimos verificar que hay problemas que requieren otro tipo de solución, esto para poder reducir su complejidad en gran medida, como es el caso de este algoritmo el cual encuentra la parentización óptima en $O(n^2)$ que es mucho mejor que las otras implementaciones, aunque para que podamos lograr esto entramos en el campo de la programación dinámica, que es buscar la solución óptima de todo un conjunto de soluciones posibles, y este algoritmo nos ayuda bastante a entender las diferencias y entender lo que es la programación dinámica.

4.1. Conclusión Aguilar Garcia Mauricio

En la práctica pudimos ver como la programación dinámica es de gran ayuda para resolver problemas matemáticos como la multiplicación de matrices, puesto que la secuencia que elijamos para realizar la operación depende inmensamente de el tiempo que se vaya a tardar el algoritmo en obtener la solución por lo cual es esencial obtener la secuencia más óptima, es decir, la que contenga menor cantidad de operaciones, para poder asegurar que nuestra multiplicación sea óptima también.

4.2. Conclusión Hernández Castellanos César Uriel

La programación dinámica es una técnica muy útil para tomar una sucesión de decisiones interrelacionadas, lo que requiere una formulación de una relación recursiva apropiada para cada problema individual, pero nos proporciona el ahorro computacionales.

En cuanto a la práctica no se tuvo ningún inconveniente al momento de resolver el problema de encontrar la parentización óptima para una secuencia de matrices de manera

dinámica, donde se complicó fue al momento de intentar programarlo por fuerza bruta.

5. Anexo

5.1. Función 1

Problema: Utilizando método por sustitución, muestre que la siguiente ecuación de recurrencia es $\Omega(2^n)$.

$$P(n) = \sum_{k=1}^{n-1} P(k)P(n-k), \text{ si } n \geq 2, \text{ y } P(n=1) = 1.$$

Sustituyendo se tiene que:

Para $n = 2$

$$P(2) = \sum_{k=1}^1 P(k)P(2-k) = P(1)P(1) = 1$$

$$P(2) = 1 \geq c * 2^2 = 4 * c \quad \forall c \leq 1/4$$

De manera análoga calculamos para 3, 4, 5, 6 y 7.

$$P(3) = P(1)P(2) + P(2)P(1) = 1 * 1 + 1 * 1 = 2$$

$$P(3) = 2 \geq c * 2^3 = 8 * c \quad \forall c \leq 1/4$$

$$P(4) = P(1)P(3) + P(2)P(2) + P(3)P(1) = 1 * 2 + 1 * 1 + 1 * 1 = 5$$

$$P(4) = 5 \geq c * 2^4 = 16 * c \quad \forall c \leq 1/4$$

$$P(5) = P(1)P(4) + P(2)P(3) + P(3)P(2) + P(4)P(1)$$

$$P(5) = 1 * 5 + 1 * 2 + 2 * 1 + 5 * 1 = 14$$

$$P(5) = 14 \geq c * 2^5 = 32 * c \quad \forall c \leq 1/4$$

$$P(6) = P(1)P(5) + P(2)P(4) + P(3)P(3) + P(4)P(2) + P(5)P(1)$$

$$P(6) = 1 * 14 + 2 * 5 + 2 * 2 + 5 * 2 + 14 * 1 = 52$$

$$P(6) = 52 \geq c * 2^6 \vee c \leq 1/4$$

$$P(7) = P(1)P(6) + P(2)P(5) + P(3)P(4) + P(4)P(3) + P(5)P(2) + P(6)P(1)$$

$$P(7) = 1 * 52 + 1 * 14 + 2 * 5 + 5 * 2 + 14 * 1 + 52 * 1 = 152$$

$$P(7) = 152 \geq c * 2^7 \vee c \leq 1$$

Recordando que para que $P(n) = \Omega(2^n)$ significa que existe una c , tal que:

$$P(n) \geq c * 2^n$$

Tomando las evaluaciones anteriores y tomando a c como un valor menor que $1/4$, se tiene que:

$$P(n) \geq c * 2^n \vee c \leq 1/4$$

Ya que se cumple esta condición, podemos decir que, en efecto, $P(n) = \Omega(2^n)$.

Referencias

- [1] "PROGRAMACIÓN DINÁMICA", Ingenieria UNAM, 2017. [Online]. Disponible:
http://www.ingenieria.unam.mx/sistemas/PDF/Avisos/Seminarios/SeminarioV/Sesion6_IdaliaFlores
- [2] Introduction to Algorithms, Second Edition by Thomas H. Cormen, Charles E. Leiserson,
Ronald L. Rivest, Clifford Stein