Escritura de Algoritmo para la multiplicación matricial más optima

Pablo Santander Álvarez, Steban Vanegas Camacho

Departamento de Ingeniería de Sistemas, Pontificia Universidad Javeriana
Bogotá, Colombia
santander_pjaveriana.edu.co, steban_vanegasc@javeriana.edu.co

31 de agosto de 2022

Resumen

En este documento se presenta la descripcion de un algoritmo de programación dinamica que solucion el caso de: dado una secuencia de matrices, calcular el menor número de multiplicaciones necesarias posibles e indicar donde debe posicionarse el parentesis para obtener este resultado. Además se presentan analisis experimentales. **Palabras clave:** programación dinamica, secuencia, matriz, algoritmo, formalización, experimentación, complejidad.

Índice

1. Introducción

El presente documento tiene como objetivo mostrar el proceso para resolver un algoritmo de la multiplicación de matrices más optima, utilizando la tecnica de programación dinamica donde partiendo de una ecuación recursiva se busca construir diferentes algoritmos hasta poder llegar a aquel que nos indique la menor cantidad posible de multiplicaciones escalares y que a su vez muestre donde deben estar posicionados los parentesis para que la multiplicación sea la más optima. Se pretende mostrar: la formalización del problema (sección 2), la escritura formal de dos algoritmos (sección 3) y un análisis experimental de la complejidad de ambos.

2. Formalización del problema

Para el problema de *multiplicación de matrices* inicialmente podemos partir de una matriz $A : \mathbb{R}^{(n*m)}$ y una segunda matriz $B : \mathbb{R}^{(m*p)}$, en este caso al solo tener dos matrices podemos inmediatamente concluir que la multiplicación mas óptima es $A * B$, resultando entonces en una matriz $C : \mathbb{R}^{(n*p)}$, es importante tener en cuenta que las filas de una matriz deben ser iguales a las columnas de la matriz con la cual se quiere

multiplicar. Sin embargo, la idea detras del algoritmo es que funcione en los casos donde tenemos más de dos matrices, es decir, una cadena de matrices. Para este caso partamos entonces de la suposición de que tenemos una cadena de matrices, especificamente tres matrices: $A1 : \mathbb{R}^{(n*m)}$, $A2 : \mathbb{R}^{(m*p)}$, $A3 : \mathbb{R}^{(p*r)}$. Aqui es donde entonces surge la pregunta ¿Donde deberia posicionar mis parentesis para asegurar la menor cantidad de multiplicaciones escalares entre esta cadena de matrices?. Un caso posible seria $(A1*A2)(A3)$, donde $A1$ y $A2$ son multiplicados entre si y el resultado finalmente multiplicado por $A3$. El otro caso posible seria $(A1)(A2*A3)$. Formalmente, la manera para determinar la mejor solucion es $min(M[i,k]+M[k+1,j]+m[i,(k+1,j)])$. $M[i,k]$ es la multiplicacion desde la matriz i hasta la matriz k donde se posiciona el parentesis, $M[k+1,j]$ es la multiplicacion de la matriz despues del parentesis hasta el final de la cadena de matrices y $m[i,(k+1,j)]$ es el número de multiplicaciones escalares para calcular el reusltado entre $M[i,k] * M[k+1,j]$.

2.1. Definición del problema de Multiplicación de Cadena de Matrices

Así, el problema de la optimización de multiplicación de cadena de matrices se define a partir de:

1. Una cadena de matrices $A =< A1, A2, A3....An >$, donde $A_i \in \mathbb{R}^{(r_i*c_i)} \wedge r_i = c_1 - 1$

Produciendo un número que indica la menor cantidad posible de multiplicaciones.

- Entradas:
  - La entrada se puede reducir y simplificar como: $D =< d_i : 0 \leq i \leq n \wedge d_i \in \mathbb{N} >$. Donde d_0 es la cantidad de filas de la primera matriz, d_1 es la cantidad de columnas de la primera matriz y a su vez la cantidad de filas de la segunda matriz.
- Salidas:
  - Teniendo en cuenta la simplificación, la primera salida seria entonces: $min(M_{ik} + M_{k+1j} + m_{ikj})$. Donde $m_{ikj} = d_{i-1}d_kd_j$. La segunda salida, la cual es producto del Backtracking, seria una secuencia $S = \langle a_i \in Sm \rangle$.

3. Algoritmo de solución

3.1. Multiplicación de Cadenas de Matrices - Bottom-Up con Backtracking

Este algoritmo es el resultado final de realizar los diferentes pasos necesarios para construir un algoritmo de programación dinamica, siendo estos: Algoritmo inocente, Algoritmo con Memoización, Algoritmo bottom-up y finalmente bottom-up con backtracking. La idea de este algoritmo es mostrarle al usuario cuál es la cantidad minima posible de multiplicaciones escalares para procesar la multiplicación de una cadena de matrices y a su vez, mostrarle al usuario exactamente en donde deberian estar ubicados los parentesis para poder alcanzar dicha optimización. Esto se logra mediante la implementación de un algoritmo de dividr y conquistar encargado de realizar el backtracking, construyendo entonces los parentesis y la ubicación de estos en nuestra cadena de matrices. Esto es importante ya que el solo hecho de mostrar el resultado de la cantidad de multiplicaciones escalares más optima no es suficiente para que un usuario pueda entender y saber cómo multiplicar la cadena de matrices, allí es donde entra en juego el backtracking y la posibilidad visual que brinda para entender la parentización.

Algoritmo 1 Bottom-up con Backtracking.

Require: $D =< d_i : 0 \leq i \leq n \wedge d_i \in \mathbb{N} >$
Ensure: $min(M_{ik} + M_{k+1j} + m_{ikj})$

```
1: procedure CadenaMatriz(D)
2:     n = |D| − 1
3:     let m[0...n][0...n] and s[0...n][0...n] be two matrices
4:     for L ← 2 to n do
5:         for i ← 1 to n − L + 1 do
6:             j ← i + L − 1
7:             m[i][j] ← ∞
8:             for k ← i to j do
9:                 q ← (m[i][k] + m[k + 1][j] + D[i − 1] ∗ D[k] ∗ D[j])
10:                if q < m[i][j] then
11:                    m[i][j] ← q
12:                    s[i][j] ← k
13:                end if
14:            end for
15:        end for
16:    end for
17:    Return(m[1][n − 1], Parentesis(s, 1, n))
18: end procedure
19:
20: procedure Parentesis(s, i, j)
21:    name ← A where A is char 'A'.
22:    if i = j then
23:        print(name ← name + 1)
24:        return
25:    end if
26:    print("(")
27:    Parentesis(s, i, s[i][j])
28:    Parentesis(s, s[i][j] + 1, j)
29:    print(")")
30: end procedure
```

3.1.1. Análisis de complejidad

Para determinar la complejidad del algoritmo podemos usar la tecnica de inspección de codigo, donde podemos identificar que el algoritmo posee tres ciclos que se encuentran anidados. Por esta razón, el algoritmo tiene una notación de $O(|D|^3)$

3.1.2. Invariante

Como se analizó y dijo en clase, la Invariante para los algoritmos desarrollados mediante la tecnica de Programación Dinamica es el correcto llenado de la tabla o estructura de datos a usar, donde se almacenan distintos cálculos permitiendo que los elementos que ya han sido calculados no tengan necesidad de calcularse de nuevo.

4. Análisis experimental

En esta sección se presenta el comportamiento del algoritmo. Para el experimento se decidio usar secuencias de matrices que se encuentren en un rango entre 2 y 100 matrices, con valores entre 1 y 400. El objetivo no se basa en medir la velocidad de solución del algoritmo a medida que el tamaño de la cadena de matrices

aumenta, sino medir la precisión con la cual responde sobre la cantidad minima de multiplicaciones escalares y el posicionamiento de los parentesis para garantizar esa respuesta.

Para comprobar y corrobar la respuesta obtenida por el algoritmo, se hara uso de la herramienta https://www.mimuw.edu.pl/ erykk/algovis/mcm.html la cual nos permite ingresar una cadena de matrices y obtener las diferentes multiplicaciones escalares posibles, indicando cuál es la mejor y a su vez, donde deberia ir el parentesis, todo esto de manera textual y gráfica.

Experimento 1, matriz $D < 10, 100, 5, 50 >$

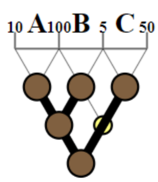
Resultado en Julia

```
      println(MultiplicacionCadenaMatrices.optimizacion([10,100,5,50]))

      end  # module MultiplicacionCadenaMatrices
(7500, "((1×2)×3)")
MultiplicacionCadenaMatrices
```

Resultado en Herramienta

10 100 5 50



The cheapest method to compute ABC is ((AB)C) with cost 7500:

- **A * BC:**
  A is a 10 x 100 matrix
  BC is a 100 x 50 matrix, computed in 25000 steps using (BC)
  total cost: 0 + 25000 + 50000 = 75000
- **AB * C:**
  AB is a 10 x 5 matrix, computed in 5000 steps using (AB)
  C is a 5 x 50 matrix
  total cost: 5000 + 0 + 2500 = 7500

Experimento 2, matriz $D < 195, 50, 69, 70, 365, 121, 323, 57, 184, 397, 310, 245, 218, 183, 333, 242, 92, 140, 122, 264, 231... >$ (Tamaño 50)

Resultado en Julia

```
      end  # module MultiplicacionCadenaMatrices
WARNING: replacing module MultiplicacionCadenaMatrices
(4024656, "((1×(2×(3×(4×(5×(6×(7×(8×(9×(10×(11×(12×(13×(14×(15×(16×(17×(18×(19×(20×(21×(22×(23×(24×(25×(26×(27×(28×(29×(30×(31×(32×(33×(34×(35×(36×(37×(38×(39×(40×(41×(42×(43×(44×(
48×49))")
MultiplicacionCadenaMatrices
```

Resultado en Herramienta

The cheapest method to compute ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopq is ((A(B(C(D(E(F(G(H(I(J(K(L(M(N(O(P(Q(R(S(T(U(V(W(X(Y(Z([(\(](^(_(`(a(b(c(d(e(f(g(
cost 4024656:

Experimento 3, matriz $D < 107, 145, 4, 81, 167, 114, 49, 170, 169, 87, 136, 51, 50, 90, 48, 120, 153, 67, 196, 24, 182, 124, 113, 155$ (Tamaño 80)

Resultado en Julia

```
      end  # module MultiplicacionCadenaMatrices
WARNING: replacing module MultiplicacionCadenaMatrices
(1703204, "((1×(2×(3×(4×(5×(6×(7×(8×(9×(10×(11×(12×(13×(14×(15×(16×(17×(18×(19×(20×(21×(22×(23×(24×(25×(26×(27×(28×(29×(30×(31×(32×(33×(34×(35×(36×(37×(38×(39×(40×(41×(42×(43×(44×(
9×60))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))×((((((((((((((((61×62)×63)×64)×65)×66)×67)×68)×69)×70)×71)×72)×73)×74)×75)×76)×77)×78)×79))")
MultiplicacionCadenaMatrices
```

Resultado en Herramienta

The cheapest method to compute ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~ ������������������ is ((A(B(C(D(E(F(G(H(I(J(K(L(M(N(O(P(Q(R(S(T(U(V(W(X(Y(Z([[\\[](^{_(`(a(b(c(d(e(f(g(h(i(j(k(l(m(n(o(p(q(r(s(t(u(v(w(x(y(z({|}~))��)��)��)��)��)��)��)��)��)��)��)��)��)) with cost 1703204: