Tarea 5: Algoritmo de Strassen

Tania Patiño 137175
Víctor Peña 119413
Javier Sagastuy 122627
Ernesto Valdés 114049
Dr. Osvaldo Gabriel Cairó Batistutti

ITAM
México, D.F.

Introducción

En principio de va a explicar el algoritmo clásico para la multiplicación de matrices para después compararlo con el algoritmo de Strassen, propuesto por Volver Strassen en 1969.

El método clásico para calcular la multiplicación de matrices de nxn consiste en :

Sean $A, B, C \in M_{2^n x 2^n}$, la multiplicación de matrices se define como: $C = AB$

Entonces $C = \begin{pmatrix} c_{11} & \dots & c_{1n} \\ \vdots & \ddots & \vdots \\ c_{n1} & \cdots & c_{nn} \end{pmatrix} = \begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nn} \end{pmatrix} \begin{pmatrix} b_{11} & \dots & b_{1n} \\ \vdots & \ddots & \vdots \\ b_{n1} & \cdots & b_{nn} \end{pmatrix}$

donde $c_{ij} = \sum_{r=1}^{n} a_{ir} b_{rj}$

esto es que:

$c_{11} = a_{11}b_{11} + \cdots + a_{1n}b_{n1}$

$c_{21} = a_{21}b_{11} + \cdots +$

Si las matrices mencionadas no son del orden definido, entonces identificamos la matriz con el orden máximo, después a ese valor le sacamos el logaritmo base dos y el resultado lo aproximamos hacia el número entero mayor al cuál designaremos como l. Finalmente, para obtener el orden con que se van a trabajar las matrices, elevaremos 2^l y por lo tanto reconstruiremos las matrices con tamaño $2^l x 2^l$. Entonces $A, B, C \in M_{2^l x 2^l}$. Los elementos que llegasen a faltar para completar el nuevo orden se llenan con ceros.

Strassen demostró que éste no es el método óptimo para realizar multiplicaciones de matrices, ya que propuso un nuevo método que es ligeramente más eficiente que el métoco clásico, pues realiza menos multiplicaciones que la forma clásica. A continuación mencionamos cómo funciona el método de Strassen:

Sean las mismas matrices A, B, C para describir paso a paso el funcionamiento del método vamos a tomar un ejemplo de matrices de tamaño 2 x 2.

Sea $A, B, C \in M_{2^n x 2^n}$ donde $n = 1$ vamos a realizar la operación:
$C = \begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix}$
Strassen propone definir una serie de matrices como se sigue:
$M_1 = (a_{11} + a_{22}) * (b_{11} + b_{22})$
$M_2 = (a_{21} + a_{22}) * (b_{11})$
$M_3 = a_{11} * (b_{12} - b_{22})$
$M_4 = a_{22} * (b_{21} - b_{11})$
$M_5 = (a_{11} + a_{12}) * b_{22}$
$M_6 = (a_{21} - a_{11}) * (b_{11} + b_{12})$
$M_7 = (a_{12} - a_{22}) * (b_{21} + b_{22})$
Por lo tanto los elementos de la matriz C se definen como:
$c_{11} = M_1 + M_4 - M_5 + M_7$
$c_{12} = M_3 + M_5$
$c_{21} = M_2 + M_4$
$c_{22} = M_1 - M_2 + M_3 + M_6$

Lo anterior constituye el caso base de la estrategia divide y vencerás, ya que cuando $n = 1$ el problema lo sabemos resolver y podemos dar un resultado. Pero cuando $n \to \infty$, tenemos que partir las matrices A, B, C en matrices de tamaño más pequeño.

Sean $A, B, C \in M_{2^n x 2^n}$ donde $n > 1$, debemos dividir las matrices en submatrices de tamaño $2^m x 2^m$ donde $2^m < 2^n$. Por lo tanto, vamos dividiendo las matrices originales en tantas matrices sea necesario hasta llegar al caso base, que es cuando $n = 1$, es decir, cuando solamente se tienen matrices de tamaño $2x2$ y existen puros elementos y no submatrices en los factores a multiplicar.

La estrategia utilizada en la solución sigue claramente "Divide y vencerás", pues partimos el problema en subproblemas más pequeños hasta el punto en que los subproblemas son tan simples que los podemos resolver y luego se va construyendo la solución a partir de la solución de esos problemas pequeños.

Para la implementación en programación, se reciben como parámetros las matrices a multiplicar, si no cumplen con ser cuadradas o potencias de 2, se realiza lo que ya se explicó de tomar el máximo y sacarle el logaritmo base 2 y aproximar hacia el número entero mayor. Elevar el 2 al valor obtenido del logaritmo. Los elementos que faltan para que las matrices sean cuadradas, se llenan con ceros. Y entonces se realiza una llamada recursiva al método "Strassen". Dentro de ese método se checa si ya se llegó al caso base, si sí, se resuelve el problema y es cuando se detiene la recursividad y se va construyendo la solución a partir de ello. En dado caso de que no se haya llegado al caso base, se hace una llamada recursiva al método de Strassen por cada matriz $M_1, M_2, M_3, M_4, M_5, M_6, M_7$ con los valores definidos para cada una de las matrices.

¿Porqué el algoritmo de Strassen es más rápido que el algoritmo clásico para multiplicación de matrices?

La multiplicación de matrices por el método clásico requiere de $2N^3$operaciones aritméticas(tomando en consideración las sumas y las multiplicaciones) y por lo tanto es de $O(N^3)$.

Para el algoritmo de Strassen, vamos a calcular el número de sumas y multiplicaciones requeridas para multiplicar matrices de $2^n x 2^n$:

Como estamos trabajando el algoritmo de Strassen recursivamente podemos saber que:

$f(n)$ que es la función recursiva del método de Strassen, se puede definir como:

$f(n) = 7f(n-1) + l4^n$, donde l depende del número de sumas realizadas en cada llamada recursiva al algoritmo. Por lo tanto se deduce que $f(n) = (7+o(1))^n$ y esto es la complejidad asintótica de multiplicar matrices de tamaño $2^n x 2^n$ usando al método de Strassen y nos queda entonces:

$O((7+o(1))^n) = O(N^{log_2 7+o(1)}) \approx \mathrm{O}(N^{2.807})$.

Como conclusión, hay una ligera reducción, ya que $O(N^{2.807}) < O(N^3)$ y por lo tanto es más eficiente el algoritmo de Strassen que el método clásico para la multiplicación de matrices.

Complejidad

El análisis de este algoritmo es bastante sencillo si se utiliza el teorema maestro. Note que como se describe anteriormente, se empieza por dividir el problema original en un problema de multiplicación de matrices de la mitad de tamaño (en cada uno de los lados). Esto implica $b = 2$ en la notación del teorema maestro. Se calculan 7 multiplicaciones de matrices de este tamaño reducido. Esto implica $a = 7$. Finalmente, para dividir el problema y para combinar las soluciones parciales solamente se utilizan sumas las cuales toman $\theta(n^2)$ pues se trata de matrices cuadradas de $n \times n$. Note que también parte importante de la división del problema está en obtener las submatrices. Esto se hace 8 veces y si la implementación de las librerías que estamos utilizando es adecuada, debería ejecutarse en $O\left(\frac{n^2}{4}\right) = O(n^2)$. Así, tenemos que el tiempo de ejecución del algoritmo de Strassen está dado por la fórmula de recurrencia:

$$T(n) = 7T\left(\frac{n}{2}\right) + \theta(n^2)$$

Así, al aplicar el teorema maestro, tenemos que

$$n^{\log_2 7} = n^r$$

con $r = \log_2 7 = 2.80735$. Puesto que claramente

$$\theta(n^2)" < "\theta(n^{r-\varepsilon}) \quad \forall\, 0 < \varepsilon < r - 2 = 0.80735$$

el teorema maestro nos permite concluir que

$$T(n) = \theta\left(n^{\log_b a}\right) = \theta\left(n^r\right) = \theta\left(n^{2.80735}\right)$$

A continuación se presenta el código del algoritmo implementado en python. Como se puede ver, en cada nivel de la recursión se resuelven 7 subproblemas

de la mitad del tamaño. También se puede ver que la división del problema y la combinación de resultados toma $\theta(n^2)$. El algoritmo se manda llamar con el método `strassen`. Éste sólo se encarga de "rellenar" con ceros las matrices a multiplicar de tal modo que queden cuadradas y de dimensión $2^m \times 2^m$ donde $m = \lceil \lg(\text{máx}\{ma, na, nb\}) \rceil$. Finalmente, este método manda llamar `strassenR` que ejecuta el algoritmo de Strassen de manera recursiva utilizando la técnica divide y vencerás.

Código:

```
def strassen(A,B):
        (ma,na) = A.shape
        (mb,nb) = B.shape
        if na != mb:
                print 'Dimensions dont agree'
        else:
                n = max(ma,na,nb)
        m = math.ceil(math.log(n,2))
        A0 = np.zeros(shape=(2**m, 2**m))
        B0 = np.zeros(shape=(2**m, 2**m))
        A0[0:ma,0:na] = A
        B0[0:mb,0:nb] = B
        C0 = strassenR(A0,B0)
        return C0[0:ma,0:nb]

def strassenR(A,B):
        l = len(A)
        if l == 1:
                return A.dot(B)
        A11 = A[0:l/2,0:l/2]
        A12 = A[0:l/2,l/2:l]
        A21 = A[l/2:l,0:l/2]
        A22 = A[l/2:l,l/2:l]
        B11 = B[0:l/2,0:l/2]
        B12 = B[0:l/2,l/2:l]
        B21 = B[l/2:l,0:l/2]
        B22 = B[l/2:l,l/2:l]

        M1 = strassenR(A11+A22, B11+B22)
        M2 = strassenR(A21+A22, B11)
        M3 = strassenR(A11, B12-B22)
        M4 = strassenR(A22, B21-B11)
        M5 = strassenR(A11+A12, B22)
        M6 = strassenR(A21-A11, B11+B12)
        M7 = strassenR(A12-A22, B21+B22)

        C11 = M1 + M4 - M5 + M7
        C12 = M3 + M5
        C21 = M2 + M4
        C22 = M1 - M2 + M3 + M6
```

```
C = np.zeros(shape=(l,l))

C[0:l/2,0:l/2] = C11
C[0:l/2,l/2:l] = C12
C[l/2:l,0:l/2] = C21
C[l/2:l,l/2:l] = C22

return C
```

Resultados

Con la finalidad de tener una comparación objetiva en cuanto al tiempo de ejecución del algoritmo de Strassen con respecto al producto de matrices tradicional, se implemento dicho algoritmo tanto en Python como en Java. Para que los resultados sean comparables, la implementación del algoritmo es la misma y se utilizaron estructuras de datos similares. Una vez implementado el algoritmo de Strassen, nos dimos a la tarea de comparar su desempeño con respecto al producto matrical tradicional.

Lo que se esperaba observar era que el algoritmo de Strassen fuera mucho mejor que un producto de matrices tradicional sin embargo al realizar las primeras ejecuciones dicho supuesto debió de ser replanteado; ya que los tiempos de ejecución para realizar el producto de matrices pequeñas mediante el algoritmo de Strassen resultaba considerablemente mayor (al rededor de 500 veces). En un primer momento se busco reducir el uso de la memoria por el tiempo toma la asignación de esta. Una vez que se redujo al mínimo el uso de la memoria, no quedo otra alternativa más que remitirse a la complejidad del problema para tratar de explciar el comportamiento que estabamos observando. El algrotimo de Strassen tiene complejidad aproximada a $\theta(n^{2.8})$ y el producto de matrices tradicional con una complejidad de $\theta(n^3)$ sabemos que habrá un momento en el cual el algoritmo de Strassen resultará más eficiente que el producto de matrices tradicional.

Una vez que estimamos la complejidad del algoritmo de Strassen y los resultado experimentales demostraron que conforme se iban incrementando las dimensiones de las matrices la diferencia entre un algoritmo y el otro se reducian, definimos las dimensiones de las matrices a comparar. Dichas dimensiones fueron $4x4$, $10x10$, $100x100$, $1000x1000$ y $4000x4000$ considerando que conforme se incrementaban las dimensiones de la matriz se requería más tiempo y memoria para resolver el problema. Con los casos de prueba determinados, se procedió a probar el desempeño del algoritmo de Strassen y se obtuvieron los siguientes resultados:

Python

Dimensión matiz	Producto tradicional	Strassen
4x4	0.000890016555786 seg	0.0022599697113 seg
10x10	0.000956058502197 seg	0.0236649513245 seg
100x100	0.441963911057 seg	7.53841590881 seg
1000x1000	352.728410006 seg	2596.56640506 seg
4000x4000		

Java

Dimensión matriz	Producto tradicional	Strassen
4x4	0 seg	0.002 seg
10x10	0 seg	0.017 seg
100x100	0.002 seg	1.324 seg
1000x1000	20.429 seg	252.179 seg
4000x4000	3180.524 seg	13802.893 seg

Análisis de Resultados

A partir de las ejecuciones para diferenctes matrices y con los tiempos de ejecución obtenidos reflejados en las tablas anteriores, se puede concluir lo siguiente:

1. El algoritmo de Strassen es extremadamente ineficiente al multiplicar matrices pequeñas respecto al producto tradicional.

2. Existe al menos una matriz de dimensiones nxm para las cuales el algoritmo de Strassen tenga un tiempo de ejecución menor que el producto tradicional.

3. El algoritmo de Strassen se encuentra limitado a la cantidad de memoria disponible para las dimensiones de la matriz en el caso de Java.

4. Conforme se aumenta el tamaño de las matrices, el algoritmo de Strassen comienza a tener un mejor desmpeño hasta que este sea mejor que el obtenido en el producto tradicional.

Consideraciones posteriores

Al observar que los resultados de la ejecución del algoritmo de Strassen tanto en Python como en Java resultaban contradictorios con respecto a lo que la teoría indicaba, se investigaron las posibles causas del problema. Se encontró en una publiación de la universidad de Duke [3] que una de las posibles mejoras a realizar en el algoritmo era truncar la recursión cuando se tuvieran matrices de 16x16. La publiación también habla de una posible manera de optimizar

la memoria. Nosotros únicamente implementamos en el algoritmo de Strassen el truncado de la recursión en matrices de 8x8 y de 16x16 para comparar su desempeño. En ambos casos se observó que el tiempo que le tomaba al algoritmo de Strassen realizar el producto de matrices resultaba mucho menor que si se realizara la recursión completa; sin embargo, el truncado óptimo fue de 16x16 par ambos casos. Los resultados de las implementaciones en Python y en Java se muestran a continuación.

Python

Truncado de la recursión en matrices de 8x8

Dimensión matriz	np.dot()	Strassen
4x4	0.00002 seg	0.00002.seg
10x10	0.00002 seg	0.00026 seg
100x100	0.0015 seg	0.0461 seg
1000x1000	13.7685 seg	16.0488 seg
4000x4000	1450.2125 seg	758.5871 seg

Truncado de la recursión en matrices de 16x16

Dimensión matriz	np.dot()	Strassen
4x4	0.00002 seg	0.000105 seg
10x10	0.00002 seg	0.00012 seg
100x100	0.00155 seg	0.008996 seg
1000x1000	13.666 seg	3.186 seg
4000x4000	1406.5 seg	156.5 seg

Java

Truncado de la recursión en matrices de 8x8

Dimensión matriz	Producto tradicional	Strassen
4x4	-	-
10x10	0 seg	0.026 seg
100x100	0.002 seg	0.082 seg
1000x1000	19.991 seg	5.143 seg
4000x4000	2753.416 seg	220.649 seg

Truncado de la recursión en matrices de 16x16

Dimensión matriz	Producto tradicional	Strassen
4x4	-	-
10x10	0.001 seg	0.024 seg
100x100	0.001 seg	0.077 seg
1000x100	18.026 seg	2.815 seg
4000x4000	2646.278 seg	107.474 seg

Como se puede observar en las tablas anteriores, la reducción en tiempo de ejecución del algoritmo de Strassen en Python y Java son consistentes. Asimismo, el haber utilizado como condición de truncado matrices de 16x16 esta asociada a características propias de la arquitectura del equipo en el que se realizó la ejecución del algoritmo de Strassen por lo cual si bien en este caso resultó conveniente utilizar matrices de 16x16 podrían utilizar matrices de cualquier otra dimensión que fuera compatible para optimizar la ejecución.

Aplicaciones

Algunas aplicaciones de la multiplicación de matrices de grandes dimensiones incluyen procesamiento de imágenes y animación digital. En la presentación se cubrieron estos temas. Otra aplicación importante es en la rama de la astronomía, que al final de cuentas se reduce a procesamiento de imágenes. En todas las aplicaciones mencionadas anteriormente las dimensiones de las matrices se encuentran actualmente en el orden de 1000 elementos por lado. Por ejemplo: el animación digital se utilizan matrices de 1920×1080 que equivale a la resolución de una película en alta definición.

Referencias

[1] Bravo,Ignacio et.al., "Algoritmos no sistólicos para la multiplicación de matrices en FPGA'S", Departamento de electrónica, Universidad de Alcalá.

[2] Algoritmo de Strassen. (2013, 8 de marzo). Wikipedia, La enciclopedia libre. Fecha de consulta: 16:42, octubre 9, 2013 desde http://es.wikipedia.org/w/index.php?title=Algoritmo_de_Strassen&oldid=64482643.

[3] Thottethodi Mithuna., Lebeck R, Alvin,Tuning Strassen's Matrix Multiplication for Memory Efficiency. Duke University, Durham, NC Siddhartha Chatterjee The University of North Carolina, Chapel Hill, NC, Duke University, Durham, NC, Pages 1-14, IEEE Computer Society Washington, DC, USA ©1998