**Leandro Salias (leandrosalias42@gmail.com) – Gonzalo Clementi (gonzacl26@gmail.com)**

**Grupo 6**  **Ayudante: Lucas Boccanfuso**

Proyecto Cursada 2017 - Etapa 2 – AyD Algoritmos I

Análisis y diseño de algoritmos I

**Introducción al problema**

En esta segunda etapa del proyecto de cursada se nos presentó un problema en el cual se debe implemente una nueva forma de multiplicar matrices alternativa a la forma estándar. Para ello, la manera más eficiente de resolver este conflicto, es utilizando el algoritmo de Strassen, el cual mejora su complejidad temporal para matrices mayores de 4x4.

**Tipos de datos utilizados**

CLASS Matriz[Elem]

IMPORTS Natural

BASIC CONSTRUCTORS inicMatriz, Insertar

EFFECTIVE

TYPE Matriz

OPERATIONS

inicMatriz: Nat \* Nat -> Matriz;

Insertar: Matriz \* Nat \* Nat -> Matriz;

Obtener: Matriz \* Nat \* Nat -> Elem;

Suma: Matriz \* Matriz -> Matriz;

Resta: Matriz \* Matriz -> Matriz;

Multiplicación: Matriz \* Matriz -> Matriz;

obtenerCuadrante: Matriz\* Nat -> Matriz;

unirCuadrante: Matriz \* Matriz \* Matriz \* Matriz -> Matriz;

obtenerFilas: Matriz -> Nat;

obtenerColumnas: Matriz -> Nat;

AXIOMS

….

END-CLASS.

**Matriz** // Constructora

**Insertar** // Constructora

**Obtener** // Observadora

**Suma** // Modificadora

**Resta** // Modificadora

**Multiplicación** // Modificadora

**obtenerCuadrante** // Modificadora

**unirCuadrante** // Modificadora

**obtenerFilas** // Observadora

**obtenerColumnas** // Observadora

**Estructura de datos utilizada**

**TDA Matriz:** Para la implementación de esta nueva parte, creamos una nueva clase y no modificamos las estructuras anteriormente implementadas. Esta vez definimos el TDA Matriz en el cual definimos un puntero que almacena memoria para las filas y por cada columna se recorre y se van generando los bloques a lo ancho. Las matrices que construye esta clase son dinámicas, ya que se le da su tamaño durante la ejecución del programa.

La ventaja de utilizar esta estructura es que se hace más fácil los recorridos, ya que son más legibles y se puede visualizar fácilmente si se quiere realizar un seguimiento en la hoja de lo implementado. La desventaja es que estos recorridos poseen complejidad temporal de orden cuadrático para las matrices cuadradas y complejidad filas\*columnas para las no cuadradas, ya que se recorre tanto filas como columnas.

**Descripción de los métodos utilizados**

**Matriz** // Constructora básica del TDA Matriz

Complejidad Temporal: O(filas), ya que para cada celda del arreglo debe generar sus respectivos arreglos hacia abajo.

**Matriz** // Constructor por copia

Complejidad Temporal: O(filas\*columnas), ya que copia toda la matriz recorriendo filas y columnas.

**~Matriz** // Destructor de la clase.

Complejidad Temporal: O(filas), ya que es necesario recorrer toda las filas para eliminarla.

**Insertar** // Inserta un elemento en una posición dada (fila,columna).

Complejidad Temporal: O(1).

**get** // Retorna el elemento en la posición dada.

Complejidad Temporal: O(1).

**Suma** // Realiza la suma de dos matrices y la guarda en una tercera.

Complejidad Temporal: O(dimension2)

**Resta** // Realiza la resta de dos matrices y la guarda en una tercera.

Complejidad Temporal: O(dimension2)

**Multiplicación** // Realiza la multiplicación estándar de dos matrices y la guarda en una tercera.

Complejidad Temporal: O(filas\*columnas2).

**obtenerCuadrante** // Obtiene los cuatro cuadrantes en los que el algoritmo de Strassen divide a la matriz.

Complejidad Temporal: O(dimension2) siendo dimensión la dimensión del cuadrante.

**unirCuadrante** // Une los cuadrantes solución y devuelve una matriz con el resultado de haber hecho el producto mediante el algoritmo Strassen.

Complejidad Temporal: O(dimension2), siendo dimensión la dimensión del cuadrante.

**obtenerFilas** // Devuelve la cantidad de filas de la matriz.

Complejidad Temporal: O(1).

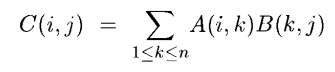
**obtenerColumnas** // Devuelve la cantidad de columnas de la matriz.

Complejidad Temporal: O(1).

**Algoritmo desarrollado y análisis de complejidad**

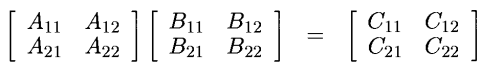
**Algoritmo de Strassen**

Sea A y B dos matrices nxn, la matriz resultante C = AB es también una matriz de nxn, la cual sus Cij se generan a partir de la siguiente operación:

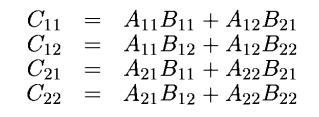


Para calcular C(i,j) usando esta fórmula, necesitamos n multiplicaciones. Como la matriz C posee n2 elementos, el tiempo del algoritmo para esta multiplicación (método convencional) es O(n3).

La estrategia divide y conquista sugiere otra manera para computar el producto de dos matrices nxn. Por simplicidad asumimos que n es potencia de 2, esto significa que existe un natural K tal que n = 2k. Imaginemos que A y B están particionadas en 4 submatrices cuadradas, cada submatriz teniendo dimensiones n/2 \* n/2. Entonces el producto AB puede ser computado usando la formula anterior para el producto de matrices de 2\*2. Si AB es:

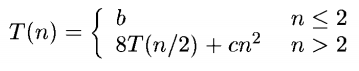
 **(1)**

Entonces:

**(2)**

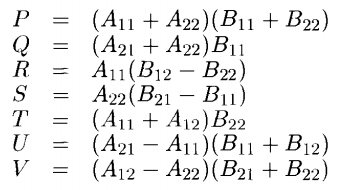
Si n = 2, entonces las formulas (1) y (2) son computadas usando un método operador de multiplicación para los elementos de A y B. Para n > 2, los elementos de C pueden ser calculados usando multiplicación de matrices y operaciones de adición aplicadas a matrices de tamaño n/2 \* n/2. Como n es potencia de 2, los productos de estas matrices pueden ser computados de forma recursiva con el mismo algoritmo que usamos para el caso n \* n. Este algoritmo se continuará llamando a si mismo trabajando sobre submatrices de menor tamaño, hasta que se vuelva adecuadamente pequeña (n = 2 o n = 4) para que el producto se calcule mediante el algoritmo estándar o convencional.

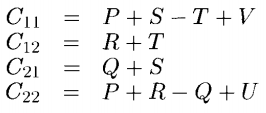
Para calcular AB usando (2), debemos realizar ocho multiplicaciones de matrices n/2 \* n/2 y cuatro adiciones de matrices del mismo tamaño. Como dos matrices de n/2 \* n/2 pueden ser sumadas en tiempo c \* n2 para alguna constante c, el tiempo de computación general T(n) del algoritmo divide y conquista está dado por la siguiente recurrencia:



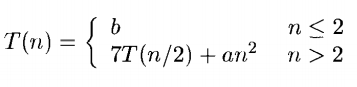
Donde c y b son constantes.

Esta recurrencia puede ser resuelta de la misma manera que las recurrencias anteriores para obtener T(n) = O(n3). Aun así, no se realizó ninguna mejora con respecto al método convencional. Como las multiplicaciones de matrices son más costosas que las adiciones (O(n3) vs O(n2)), Volker Strassen reformuló las ecuaciones para Cij y así reducir las multiplicaciones y aumentar las adiciones. Su método primero computa 7 matrices n/2 \* n/2 P, Q, R, S, T, U y V, como en (3). Así las Cij son calculadas usando las formulas en (4). Como se puede apreciar P, Q, R, S, T, U y V pueden ser computadas usando 7 multiplicaciones y 10 adiciones o sustracciones de matrices. Los Cij requieren de unas 8 adiciones o sustracciones adicionales.

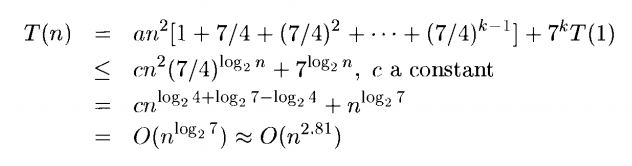
**(3)**

**(4)**

La recurrencia resultante para T(n) es:



Donde a y b son constantes. Trabajando un poco con esta fórmula, obtenemos:



Donde se puede apreciar que la complejidad mejoro de O(n3) a O(n2,81) aproximadamente.

**Impacto de la estructura sobre la solución**

Como mencionamos anteriormente la clase Matriz que implementamos posee en sus recorridos una complejidad O(filas\*columnas). Teniendo en cuenta que para realizar el algoritmo de Strassen se necesitan hacer varias adiciones, sustracciones y la multiplicación, sumados los métodos para obtener y unir cuadrantes, son muchas llamadas a métodos de orden cuadrático, lo cual es perjudicial.

En nuestro caso, por cómo se definió la estructura no podemos mejorar el costo del algoritmo. Sin embargo, usando una clase matriz quadtree podríamos identificar las submatrices que contengan mayor cantidad de ceros, para así descartarlas y no realizar operaciones innecesarias.

**Etapa de análisis**

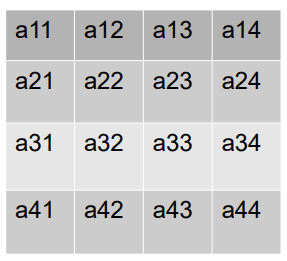
En este problema se nos planteó aplicar el algoritmo de Strassen a matrices no ralas, es decir, a matrices con minoría de ceros, pero si ahora suponemos que las matrices sean ralas (mayoría de ceros), deberían realizarse ciertas modificaciones tanto a la estructura como a al algoritmo para poder hacerlo más eficiente.

Para nuestro caso decidimos implementar una nueva clase, que es el TDA Matriz, para poder realizar un mejor seguimiento y tener una mayor facilidad a la hora de resolver los cálculos.

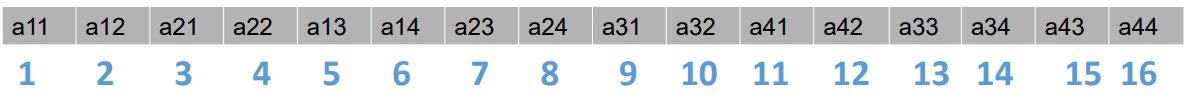
Una posible mejora a la clase, una vez implementado el algoritmo de Strassen es en el método de la multiplicación. Lo que se podría hacer en este caso es en lugar de tener un método para la multiplicación estándar y otro para la multiplicación por Strassen, preguntar dentro de la función que multiplica si las matrices a multiplicar son cuadradas, si esto se cumple luego verificar si el log2(dimStrassen) = log2(dimensiónOtraMatriz) entonces ejecuto el algoritmo, de lo contrario siempre y cuando se cumpla que las filas de una matriz A = columnas de B, hacemos la multiplicación estándar. Además, podríamos mejorar que la matriz de resultado se vaya generando en este mismo método.

Así mismo se podría haber reutilizado la clase arreglo realizada en la etapa 1 de tal forma, que la matriz se almacene de la siguiente forma:

**Matriz**



**Matriz representada en el arreglo**



Esta forma almacenaría los índices de la siguiente manera:

Siendo n = 2k y n = 8, sea Anxn

índice (A11) = índice(A)

índice (A12) = índice (A) + (n/2)2 **(5)**

Índice (A21) = índice (A) + 2 (n/2)2

Índice (A22) = índice (A) + 3 (n/2)2

Así A11 va desde el índice 1 al 16, A12 va desde el índice 17 al 32, A21 va desde 33 hasta 48 y A22 va desde 49 hasta 64.

Como desventaja, los índices del arreglo no son de fácil acceso para determinar en qué cuadrante nos encontramos. Lo que se podría implementar serían métodos que calculen los índices como se hizo en (5) para facilitar la lectura y comprensión de la matriz en forma de arreglo.

En nuestro caso, al TDA Matriz le realizamos funcionalidades básicas como devolver el número de filas y columnas, sumar y restar dos matrices iguales y multiplicar dos matrices (siempre y cuando para AB = C, que las columnas de A sean iguales a las filas de B). Pero para poder implementar el algoritmo debimos agregar dos nuevos métodos: el de obtener cuadrante y el de unir los cuadrantes. Éstos son claves a la hora de implementar el algoritmo de Strassen ya que se basa justamente en dividir la matriz de tamaño 2k en cuatro cuadrantes iguales, y luego se unen las soluciones para brindar la matriz del producto. Para el caso de las matrices ralas se podría implementar un método que verifique que una submatriz que ya no pueda dividirse y obtener sus cuadrantes, tenga mayoría o sea completamente de ceros, para así poder descartarla y no realizar operaciones sobre ella.

Por ejemplo:

bool esCero(Matriz mat, int dimension)

{

int elementos = dimension\*dimension;

int ceros = 0;

for (int i = 0; i < dimension; i++)

for (int j = 0; j < dimension; j++)

{

If (mat[i][j] == 0)

ceros += 1;

}

if (elementos == ceros)

return true;

else

return false;

}

Para mejorar la búsqueda de estas submatrices ralas proponemos una posible mejora o alternativa a la estructura implementada por nosotros.

Ésta es la implementación de la clase Matriz Quadtree, la cual almacena en un arreglo las direcciones en donde se encuentran las submatrices y en otro arreglo se guardan todos los elementos de la matriz. En el arreglo de direcciones un valor positivo indica una matriz que vuelvo a partir y representa el índice dentro del mismo arreglo donde se encuentran las particiones, un valor negativo representa la posición en el vector de elementos en la cuál está almacenada la información, y un 0 representa una región de ceros que conocemos.

Implementando de esta forma sería mucho más fácil identificar la submatrices de ceros para que luego cuando se aplique el algoritmo de Strassen no se realicen cálculos innecesarios sobre matrices ralas.

En el caso de que las matrices no sean de tamaño potencia de 2, lo que podría hacerse es rellenar con filas o columnas de ceros, según sea necesario, hasta lograr que las matrices sean de tamaño potencia de 2 más cercana al número de filas y columnas que se le pasen por parámetro. Luego usando la estructura de Matriz Quadtree el algoritmo de Strassen evitaría trabajar con esas filas y columnas de ceros agregadas.

**Conclusión**

Para finalizar el trabajo, podemos concluir que existen varias formas de implementar el algoritmo algunas más eficientes que otras, pero todas con sus respectivas ventajas y desventajas. En un principio pensamos en modificar la clase Arreglo implementada en la Etapa 1 del trabajo, pero luego nos dimos cuenta de que el manejo de índices iba a ser muy complicado. Por ello, decidimos crear una nueva clase (Matriz) que nos facilitara tanto el seguimiento como la comprensión de los métodos implementados y del algoritmo planteado.

**Código Fuente**

**Matriz.h**

#ifndef MATRIZ\_H\_INCLUDED

#define MATRIZ\_H\_INCLUDED

template <class T>

class Matriz {

public:

Matriz(int fil,int col); //Constructor

Matriz(const Matriz& m); //Constructor por copia

~Matriz(); //Destructor

void insertar(const T& elem, int fil, int col);

T get(int i,int j)const;

void suma(const Matriz& otraM,Matriz& res,int dim);

void resta(const Matriz& otraM,Matriz& res,int dim);

void mult(const Matriz& otraM,Matriz& res);

void getCuadrante(int dim, Matriz& M11,Matriz& M12,Matriz& M21,Matriz& M22);

void unirCuadrantes(int dim,const Matriz& M11,const Matriz& M12,const Matriz& M21,const Matriz& M22);

int getFilas()const;

int getColumnas()const;

void cargar();

private:

T \*\* elementos;

int filas;

int columnas;

};

#endif // MATRIZ\_H\_INCLUDED

**Matriz.cpp**

#include "Matriz.h"

#include <assert.h>

#include <iostream>

#include <stdlib.h>

template <class T>

Matriz<T>::Matriz(int fil, int col)

{

elementos = new T\*[fil];

for (int i = 0; i < fil; i++)

elementos[i] = new T[col];

filas = fil;

columnas = col;

}

template <class T>

Matriz<T>::Matriz(const Matriz& m)

{

elementos = new T\*[m.filas];

filas = m.filas;

columnas = m.columnas;

for (int i = 0; i < filas; i++)

elementos[i] = new T[columnas];

for (int i = 0; i < filas; i++)

for (int j = 0; j < columnas; j++)

elementos[i][j] = m.elementos[i][j];

}

template <class T>

Matriz<T>::~Matriz()

{

for (int f = 0; f < filas ; f++)

delete [] elementos[f];

delete [] elementos;

elementos = 0;

}

template <class T>

void Matriz<T>::insertar(const T& elem, int fil, int col)

{

assert((fil < filas) && (col < columnas));

elementos[fil][col] = elem;

}

template <class T>

T Matriz<T>::get(int i, int j)const

{

assert((i < filas) && (j < columnas));

return elementos[i][j];

}

template <class T>

void Matriz<T>::suma(const Matriz& otraM, Matriz& res,int dim)

{

assert((filas == otraM.filas) && (columnas == otraM.columnas));

for (int i = 0; i < dim; i++)

for (int j = 0; j < dim; j++)

res.elementos[i][j] = elementos[i][j] + otraM.elementos[i][j];

}

template <class T>

void Matriz<T>::resta(const Matriz& otraM, Matriz& res,int dim)

{

assert((filas == otraM.filas) && (columnas == otraM.columnas));

for (int i = 0; i < dim; i++)

for (int j = 0; j < dim; j++)

res.elementos[i][j] = elementos[i][j] - otraM.elementos[i][j];

}

template <class T>

void Matriz<T>::mult(const Matriz& otraM, Matriz& res)

{

assert((columnas == otraM.filas) && (res.filas == filas) && (res.columnas == otraM.columnas));

for (int i = 0; i < filas; i++)

for (int j = 0; j < columnas; j++)

{

res.elementos[i][j] = 0;

for (int k = 0; k < columnas; k++)

res.elementos[i][j] = res.elementos[i][j] + elementos[i][k] \* otraM.elementos[k][j];

}

}

template <class T>

void Matriz<T>::getCuadrante(int dim, Matriz& M11,Matriz& M12,Matriz& M21,Matriz& M22)

{

for(int i = 0; i < dim; i++)

for(int j = 0; j < dim; j++)

{

M11.insertar(elementos[i][j],i,j);

M12.insertar(elementos[i][j+dim],i,j);

M21.insertar(elementos[i+dim][j],i,j);

M22.insertar(elementos[i+dim][j+dim],i,j);

}

}

template <class T>

void Matriz<T>::unirCuadrantes(int dim,const Matriz& M11,const Matriz& M12,const Matriz& M21,const Matriz& M22)

{

for(int i = 0; i < dim; i++)

for(int j = 0; j < dim; j++)

{

elementos[i][j] = M11.get(i,j);

elementos[i][j+dim] = M12.get(i,j);

elementos[i+dim][j] = M21.get(i,j);

elementos[i+dim][j+dim] = M22.get(i,j);

}

}

template <class T>

int Matriz<T>::getFilas()const

{

return filas;

}

template <class T>

int Matriz<T>::getColumnas()const

{

return columnas;

}

template <class T>

void Matriz<T>::cargar() // Aclaracion: este metodo se utilizo para cargar la matriz con numeros randomizados. Para

{ // la prueba del algoritmo se puede modificar

for (int i = 0; i < filas; i++)

{

for (int j = 0; j < columnas; j++){

int aux=rand() %10+1;

elementos[i][j]=j+1;

}

}

}

template class Matriz<float>;

template class Matriz<double>;

template class Matriz<unsigned int>;

template class Matriz<int>;

**Strassen.h**

#ifndef STRASSEN\_H\_INCLUDED

#define STRASSEN\_H\_INCLUDED

#include "Matriz.h"

void Strassen(int dim, Matriz<int>& A, Matriz<int>& B, Matriz<int>& C)

{

if (dim == 4)

A.mult(B,C);

else

{

int newdim = dim/2;

Matriz<int> A11(newdim,newdim);

Matriz<int> A12(newdim,newdim);

Matriz<int> A21(newdim,newdim);

Matriz<int> A22(newdim,newdim);

Matriz<int> B11(newdim,newdim);

Matriz<int> B12(newdim,newdim);

Matriz<int> B21(newdim,newdim);

Matriz<int> B22(newdim,newdim);

Matriz<int> C11(newdim,newdim);

Matriz<int> C12(newdim,newdim);

Matriz<int> C21(newdim,newdim);

Matriz<int> C22(newdim,newdim);

Matriz<int> M1(newdim,newdim);

Matriz<int> M2(newdim,newdim);

Matriz<int> M3(newdim,newdim);

Matriz<int> M4(newdim,newdim);

Matriz<int> M5(newdim,newdim);

Matriz<int> M6(newdim,newdim);

Matriz<int> M7(newdim,newdim);

Matriz<int> AA(newdim,newdim);

Matriz<int> BB(newdim,newdim);

A.getCuadrante(newdim,A11,A12,A21,A22);

B.getCuadrante(newdim,B11,B12,B21,B22);

//Calcular M1

A11.suma(A22,AA,newdim);

B11.suma(B22,BB,newdim);

Strassen(newdim,AA,BB,M1);

//Calcular M2

A21.suma(A22,AA,newdim);

Strassen(newdim,AA,B11,M2);

//Calcular M3

B12.resta(B22,BB,newdim);

Strassen(newdim,A11,BB,M3);

//Calcular M4

B21.resta(B11,BB,newdim);

Strassen(newdim,A22,BB,M4);

//Calcular M5

A11.suma(A12,AA,newdim);

Strassen(newdim,AA,B22,M5);

//Calcular M6

A21.resta(A11,AA,newdim);

B11.suma(B12,BB,newdim);

Strassen(newdim,AA,BB,M6);

//Calcular M7

A12.resta(A22,AA,newdim);

B21.suma(B22,BB,newdim);

Strassen(newdim,AA,BB,M7);

//Calcular C11

M1.suma(M4,AA,newdim);

M5.suma(M7,BB,newdim);

AA.resta(BB,C11,newdim);

//Calcular C12

M3.suma(M5,C12,newdim);

//Calcular C21

M2.suma(M4,C21,newdim);

//Calcular C22

M1.suma(M3,AA,newdim);

M2.suma(M6,BB,newdim);

AA.resta(BB,C22,newdim);

//Unimos todas las soluciones (C11,C12,C21,C22)

C.unirCuadrantes(newdim,C11,C12,C21,C22);

}

}

#endif // STRASSEN\_H\_INCLUDED

**Bibliografía y referencias**

* Cátedra Análisis y Diseño de Algoritmos (Cursada 2017), Filminas Divide y Conquista
* Horowitz, Ellis (1998). *Fundamentals of Computer Algorithms (2nd Edition).* Nueva York: W. H. Freeman and Company