MATRICES DISPERSAS DESCRIPCION Y APLICACIONES

Sparse Matrix description and application

José Soto Mejía, Guillermo Roberto Solarte Martínez, Luis Eduardo Muñoz Guerrero
Facultad de ingenierías, Programa Ciencias de la Computación, Universidad Tecnológica de Pereira, Semillero de Optimización GNTO, Pereira Risaralda
jomejia@utp.edu.co
gsolarte294@gmail.com
lemunozg@utp.edu.co

***Resumen*—Esta investigación tiene como objetivo realizar una descripción de las matrices dispersas, además evidencia su importancia, el funcionamiento de dichas matrices y sus diferentes formas de representación, mediante la implementación de esta teoría en un lenguaje de programación Java, utilizado un editor llamado Net beans7.0 .Como se visualiza en el artículo las matrices dispersas cada día tienden a ser más utilizadas en aplicaciones de computación científica, biomédicas y se caracterizan porque la mayoría de los elementos son cero.**

***Palabras clave*— Almacenamiento, Computación Científica Datos, Dispersión, biomédicas Estructura, Matriz, Matriz dispersa, Memoria, Netbeans, Óptimo.**

***Abstract*—This research aims to provide a description of sparse matrices, as well as evidence of its importance, the functioning of these matrices and their different forms of representation, through the implementation of this theory in a Java programming language, used an editor called Net beans7.0. As is shown in the paper every day sparse matrices tend to be used in scientific computing applications, biomedical, and are characterized most elements are zero**

***Key Word*—Best, Scientific Computing, Data, biomedical Matrix, Memory, Scatter, Sparse Matrix, Net beans, Storage, Structure.**

I. INTRODUCCIÓN

Las matrices dispersas son ampliamente usadas en la computación científica, especialmente en la optimización a gran escala, análisis estructural y de circuitos, dinámica de fluidos computacionales y en general en solución numérica de ecuaciones diferenciales parciales; otras áreas de interés en donde se pueden aplicar la representación dispersa son la teoría de grafos, teoría de redes, la combinatoria, los métodos numéricos, entre otros.

Dada su frecuente aparición existen múltiples paquetes de software que permiten su implementación y operación como son Sparspak, el Yale Sparse Matrix package, algunas rutinas de la librería de subrutinas de Harwell, además del paquete de software Matlab, entre otros.

Dado que este tipo de matrices ocurren tan naturalmente a través de los años se han desarrollado distintos métodos para representarlas en un computador, de manera que sean más eficientes en su computación y almacenamiento [5].

II. LISTAS ENLAZADAS

Es una estructura de datos dinámica que consiste en tener una lista enlazada, en la cual se almacena [1] un elemento no nulo de la matriz en cada nodo de la lista enlazada. La información que se almacenaría seria el valor numérico de cada elemento de la matriz, y la posición que ocupa ese dato dentro de la matriz, es decir, el número de fila y columna en la que se encuentra en la figura 1.

$$\begin{bmatrix} 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 3 & 0 \end{bmatrix}$$

Figura1. Matriz dispersa.

Para ilustrar el método de almacenamiento se utilizará la Figura 1 como matriz base.

En la Figura 1 se observa una matriz de 5x5 en donde solo 3 de sus elementos son diferentes de cero

Fecha de Recepción: 29 de Julio de 2012
Fecha de Aceptación: 30 de Abril de 2013

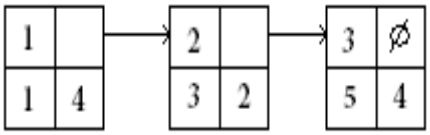

Figura 2. Lista enlazada.

En la Figura 2 se observa una representación de la matriz dispersa de la Figura 1 en una lista enlazada, en donde cada nodo almacena un elemento de la matriz, se puede observar que se tienen punteros de un nodo al siguiente, además de la información de la fila y la columna en la que se encuentra en dicha matriz.

Por ejemplo en el nodo 1 se tiene el valor numérico 1 que se encuentra en la posición 1,4 dentro de la matriz.

2 se tiene el valor numérico 2 que se encuentra en la posición 3,2 dentro de la matriz

3 se tiene el valor numérico 3 que se encuentra en la posición 5,4 dentro de la matriz

Esta forma de almacenamiento es muy ineficiente por el alto consumo de memoria aleatoria. Su única ventaja es la agregación de nuevos datos a la matriz, pero debido a que la gran mayoría de veces las matrices dispersas en la computación científica son estáticas, éste método no sirve de mucho.

III. FORMATO COORDENADO

El formato coordenado consiste en almacenar la misma información que se guardaba en el método de listas enlazadas, pero esta vez con 3 arreglos estáticos.

En el primer arreglo se almacenarán todos los datos no nulos de la matriz dispersa, por lo tanto el tamaño del vector dependerá de la cantidad de valores no nulos que tenga la matriz dispersa. En el segundo vector se almacenara la información pertinente para la fila que contiene dicho dato, y por último en el tercer vector estará almacenada la información que tiene que ver con el valor de la columna del dato.

Para la Figura 1 se tendrán entonces los siguientes vectores:

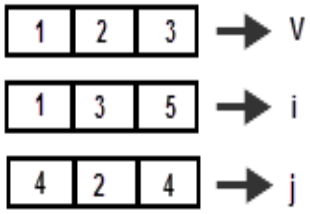

Figura 3. Formato coordenado.

Como se observa en la Figura 3 en el vector **v** se almacenan los datos que contiene la matriz dispersa. En el vector **i** se almacenan la información de la fila correspondiente para ese dato, y finalmente en el vector **j** se almacena la información que corresponde a la columna de dicho dato.

Este formato es muy eficiente cuando se quiere acceder de forma secuencial a todos los elementos, como por ejemplo cuando se quiere hacer el producto matriz por vector, en ese caso la velocidad de algoritmo usado para ese programa sería mucho más rápida y eficiente que si se usaran las listas enlazadas, sin embargo éste formato se puede mejorar [6].

IV. LISTA ENLAZADAS POR FILA

El formato comprimido tiene una eficiencia equiparable al formato coordenado, pero tiene una ventaja y es su menor uso de memoria, ya que la idea de este formato es almacenar el índice de fila implícitamente.

Para explicar este formato se debe retomar el concepto de las listas enlazadas:

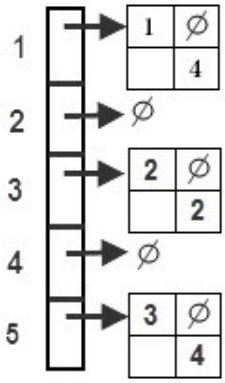

Figura 4. Lista enlazada por filas.

En la Figura 4 se muestra una estructura de datos con las listas enlazadas. En esta este formato en lugar de guardar todos los elementos de la matriz en una lista, se reagrupan por fila, por lo tanto se tendrá una lista enlazada por cada fila de la matriz.

Este formato es un poco más complejo pero tiene 2 ventajas muy importantes. La primera es el poder localizar un elemento más fácilmente, por ejemplo si se quiere conocer los datos de la fila 3, simplemente se va al puntero de la fila 4 y ahí se localizan los datos de esa fila. La segunda ventaja que se tiene es que no es necesario almacenar el índice de la fila por lo que se requerirá menos memoria. [7].

A. Formato comprimido por filas

Este formato se conoce como CSR (Compressed Sparse Row) por sus siglas en inglés, y es uno de los más utilizados en aplicaciones prácticas.

La idea de este formato es almacenar los índices de fila implícitamente, entonces se tendrán los mismos 3 vectores que en el formato coordenado, pero con una diferencia:

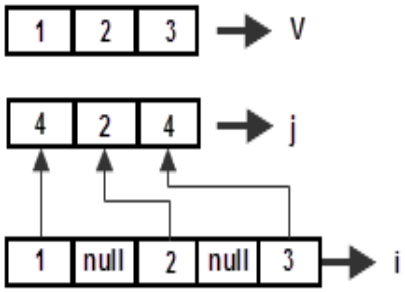

Figura 5. Formato CSR.

En la Figura 5 se puede apreciar la diferencia al formato coordenado. Los vectores **v** y **j** no cambian, estos almacenan los datos y el índice de columnas respectivamente. La diferencia se encuentra en el vector **i** ya que este almacenara la posición en donde empieza una nueva fila en el vector **j**.

B. Formato comprimido por columnas

Este formato tiene exactamente la misma forma del formato por filas, solo que esta vez el vector **j** es el que cambia, ya que este será quien almacene el índice de la fila en las cuales empiezan nuevas columnas. A este formato se le conoce como CSC (Compressed Sparse Column).

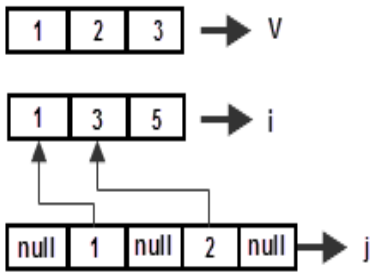

Figura 6. Formato CSC.

V. APLICACIONES

Entre las aplicaciones en las que se pueden usar este tipo de matrices se encuentran, la teoría de redes, la cuales tienen una baja densidad de datos significantes o de conexiones.

La representación de grafos es otra aplicación de las matrices dispersas, ya que los grafos en estas matrices ocuparan menos memoria debido a que solo se representaran los enlaces existentes en el grafo.

En el procesamiento vectorial se pueden aplicar matrices dispersas, debido a que existen casos especiales en operaciones vectoriales binarias como el empaquetamiento y desempaquetamiento, los cuales actúan como mascara y el vector resultado tiene un tamaño diferente al de los operandos.[8]

En la matemática combinatoria también se pueden aplicar las matrices dispersas, debido a que el fin de la combinatoria es estudiar las diferentes agrupaciones de los objetos en donde es prescindible la naturaleza de estos mas no el orden, por lo tanto se encuentran frecuentemente problemas en los que son muy útiles este tipo de matrices.

VI. EJEMPLO DE IMPLEMENTACION.

Existen múltiples factores a tener en cuenta a la hora de implementar una representación dispersa, entre ellas.

- ¿Qué nivel de dispersión se espera de las matrices con las que se desea trabajar?
- ¿Qué tipos de operaciones se van a aplicar a las matrices?
- Objetivos de diseño. ¿Se desea optimizar memoria o tiempo de procesamiento?

Como parte de este documento se presenta una implementación de matrices dispersas que permite hacer comparaciones rápidas de su rendimiento frente a la representación estándar de estas.

No se han hecho muchos compromisos a favor del ahorro de espacio en memoria ni en tiempo de procesamiento, se trató de mantener una aproximación neutral.

La implementación se realizó en el lenguaje java en su versión 1.6 corriendo sobre sobre el sistema operativo Windows 7(64 bits).

Clases:

Sparse Vector
• TreeMap<Integer, Integer> tm. • Integer N.
• SparseVector(int [])

Sparse Matrix
• Int N • Int M • SparseVector []rows
• Public SparseMatrix(int [][])
• Void Scalar(int) • SparseMatrix add(SparseMatrix, SparseMatrix) • Int[][] toNormal(SparseMatrix) • Int getNNZ(void)

Tabla1

```
import java.util.ArrayList;
import java.util.Map;
import java.lang.instrument.*;

public class SparseMatrix2 {

        int N;
        int M;
        SparseVector [] rows;


public SparseMatrix2(int [][] input)
{
                M=input.length;
                N=input[0].length;
                rows=new SparseVector[M];

for(int i=0;i<M;i++)
{
SparseVector temp=new SparseVector(input[i]);

if(temp.tm.size()!=0)
        rows[i]=temp;
else
        rows[i]=null;
  } }

public void scalar(int S)
{
for(int i=0;i<M;i++)
{if(rows[i]!=null)
{for(Map.Entry<Integer, Integer> entry :
 rows[i].tm.entrySet())
  {int key=entry.getKey();
  rows[i].tm.put(key,rows[i].tm.get(key)*S);
}}}
}

public static SparseMatrix2 add(SparseMatrix2 a,
SparseMatrix2 b)
{
for(int i=0;i<b.M;i++)
{if(b.rows[i]!=null)
{for(Map.Entry<Integer, Integer> entry :
b.rows[i].tm.entrySet())
{if(a.rows[i]==null)
{a.rows[i]=b.rows[i];
 break;}
else
{int key=entry.getKey();
if(a.rows[i].tm.get(key)==null)
{a.rows[i].tm.put(key, b.rows[i].tm.get(key));
}else
{a.rows[i].tm.put(key,a.rows[i].tm.get(key)+
b.rows[i].tm.get(key));
}}}}
}return a;
}

public static int[][] toNormal(SparseMatrix2 input)
{
int[][] retorno=new int[input.M][input.N];

for(int i=0;i<input.M;i++)
{for(int j=0;j<input.N;j++)
  {if(input.rows[i]!=null)
   {if(input.rows[i].tm.get(j)!=null)
   {retorno[i][j]=input.rows[i].tm.get(j);
    }
   else{ retorno[i][j]=0;}
   } else{  retorno[i][j]=0;} }
}return retorno;
}

public int getNNZ()
{
int contador=0;
for(int i=0;i<M;i++)
{ if(rows[i]!=null)
{  for(Map.Entry<Integer, Integer> entry :
      rows[i].tm.entrySet())
      {  contador++; }
```

```
}
}

return contador;
}

public static void main(String []args)
{
ArrayList<String> procesortime1=new
ArrayList<String>();
ArrayList<String> procesortime2=new
ArrayList<String>();
ArrayList<String> memory1=new ArrayList<String>();
ArrayList<String> memory2=new ArrayList<String>();
ArrayList<String> NNZ=new ArrayList<String>();
for(double i=0.99; i>0;i=i-0.01)
{
int filas=2000;
int columnas=2000;
System.out.println("NNZ esperados: "+i*filas*columnas);
int [][]M=new int[filas][columnas];
int [][]M2=new int[filas][columnas];
SparseMatrix.fillMatrix(M,i);
SparseMatrix.fillMatrix(M2,i);
stopWatch clock=new stopWatch();
clock.start();
int []][] add=SparseMatrix.addM(M, M2);
System.out.println("tiempo de suma no dispersa:
"+clock.stop());
procesortime1.add("tiempo de suma no dispersa:
"+clock.stop());
procesortime1.add(""+clock.stop());
SparseMatrix.PrintMatrix(add);
System.out.println("");


clock.start();
SparseMatrix2 sum1=new SparseMatrix2(M);
SparseMatrix2 sum2=new SparseMatrix2(M2);

System.out.println("Tiempo de creacion:" +clock.stop());
clock.start();
SparseMatrix2 result=SparseMatrix2.add(sum1, sum2);

System.out.println("Tiempo de suma dispersa: "+
clock.stop());

procesortime2.add("Tiempo de suma dispersa: "+
clock.stop());

procesortime2.add(""+ clock.stop());

NNZ.add(""+result.getNNZ());

clock.start();

int[][]add2=
SparseMatrix2.toNormal(result);

System.out.println("Tiempo de transformacion: "+clock.stop());

SparseMatrix.PrintMatrix(add2);


long memoria1=(12+4*M.length)*(M[0].length+1);

long memoria2=result.getNNZ()*32;

System.out.println("cantidad de memoria consumida
aproximadamente con la version 1: "+memoria1+" bytes");

System.out.println("cantidad de memoria consumida
aproximadamente con la version 2: "+memoria2+" bytes");

memory1.add("cantidad de memoria consumida
aproximadamente con la version 1: "+memoria1+" bytes");

memory2.add("cantidad de memoria consumida
aproximadamente con la version 2: "+memoria2+" bytes");

memory1.add(""+memoria1);
memory2.add(""+memoria2);

System.out.println("Numero de elementos al final:
"+result.getNNZ());

System.out.println("Numero de veces que la memoria se reduce:
"+(memoria1/memoria2));

clock.start();

SparseMatrix.normalScalar(M2, 3);

System.out.println("Normal Scalar:" +clock.stop());

SparseMatrix.PrintMatrix(M2);

System.out.println("");

clock.start();

sum2.scalar(3);

System.out.println("sparse scalar: "+clock.stop());

int [][]scalar=
SparseMatrix2.toNormal(sum2);

SparseMatrix.PrintMatrix(scalar);

}

for(int i=0;i<procesortime1.size();i++)
{
```

```
System.out.println(NNZ.get(i)+" "+memory1.get(i)+"
"+memory2.get(i)+" "+procesortime1.get(i)+"
"+procesortime2.get(i));
}}}
```

A. Sparse vector.

El vector disperso representa una fila de la matriz para su implementación se optó por un TreeMap que es una versión incluida en las colecciones de java, de un mapeo ordenado por medio de un árbol rojo negro. Esto permite acceder rápidamente a los valores guardados además de permitir recorrer las hojas en orden lo que es importante a la hora de operar matrices; la complejidad de recuperar un valor guardado *o(n)* a diferencia de un arreglo estándar en el que la complejidad computacional es *o(1)*.

Codigo fuente: SparseVector.java

```
import java.util.TreeMap;

public class SparseVector {

TreeMap <Integer, Integer>tm=new
TreeMap<Integer,Integer>();
int N;


public SparseVector(int [] input)
{int n=0;
for(int i=0;i<input.length;i++)
       {if(input[i]!=0)
       {tm.put(i, input[i]);
        n++; }}
N=n;
}}
```

B. Sparse matrix.

La matriz dispersa se representó con un vector de SparseVector.

El método add retorna la suma de dos matrices dispersas, el método toNormal devuelve la representación estándar de la matriz dispersa, el método getNNZ retorna el número de valores diferentes de 0 de la matriz dispersa.

C. Resultados.

A continuación se mostraran los resultados obtenidos al usar matrices dispersas (estos resultados se han obtenido con el programa descrito anteriormente cuyo código fuente ha sido ya puesto en este documente para efectos de reproducibilidad) y operar entre ellas, en este caso el operador es suma. Se comparan los consumos en memoria de las matrices en representación estándar y la representación dispersa propuesta en este documento. Además los tiempos de procesamiento para realizar la operación.

El tamaño original de la matriz es de 2000 filas por 2000 columnas.

NNZ son el número de elementos en la matriz resultante que son diferentes de cero.

NNZ	Memoria Estandar (bytes)	Memoria dispersa (bytes)	Tiempo Estandar (ms)	Tiempo disperso (ms)
3999591	16032012	127986912	26	14053
1749638	16032012	55988416	12	504
900997	16032012	28831904	12	155
466128	16032012	14916096	14	147
236930	16032012	7581760	12	37
157865	16032012	5051680	30	23
79927	16032012	2557884	12	11

Tabla 2.

D. Resultados retornados por el programa.

Con la implementación propuesta los beneficios de usar una representación dispersa solo se empiezan a notar a favor del uso de memoria alrededor de cuando el número de datos dispersos es igual o menor al 10% del número de elementos.
Y en tiempo de procesamiento alrededor de cuando el número de elementos diferentes a 0 rondan los 80.000 o el 2% del número de elementos originales.

VII. CONCLUSIONES

Las matrices dispersas pueden convertirse en herramientas computacionales muy útiles, pero su uso indiscriminado puede empeorar el consumo de memoria y los tiempos de procesamiento.

Existen distintas implementaciones de matrices dispersas algunas de ellas tratadas en este documento. Cada representación tiene sus ventajas y desventajas con respecto al consumo de memoria o el tiempo de procesamiento debido a esto no existe una representación universal que permita resolver todos los posibles problemas y cada implementación debe hacerse teniendo en cuenta que objetivos se quieren lograr.

Si lo comparamos con el artículo de Castellanos, J., and G. Larrazábal.[1] "Implementación out-of-core para producto matriz-vector y transpuesta de matrices dispersas", nos damos cuenta que la utilización del método out-of-core presenta resultados eficientes en el procesamiento producto matriz-vector y transpuesta de matrices dispersas, además se evidencia un ahorro importante de tiempo y memoria.

Una de las ventajas que muestra Jaramillo [2] en su investigación "Métodos directos para la solución de sistemas de ecuaciones lineales simétricos, indefinidos, dispersos y de gran dimensión", es la utilización método Cholesky, que obtiene resultados favorables en tiempo y memoria con respecto a los demás métodos.

REFERENCIAS

[1] Castellanos, J., and G. Larrazábal. "Implementación out-of-core para producto matriz-vector y transpuesta de matrices dispersas." *Conferencia Latinoamericana deComputación de Alto Rendimiento, Santa Marta,Colombia*. 2007.

[2] Jaramillo, J. D., A. V. Macıa, and F. C. Zabala. "Métodos directos para la solución de sistemas de ecuaciones lineales simétricos, indefinidos, dispersos y de gran dimensión." *Universidad Eafit* (2006).

[3] Castellanos, J., and G. Larrazábal. "Soporte out-of-core para operaciones básicas con matrices dispersas." *En Desarrollo y Avances en Metodos Numéricos Para ingeniería y Ciencias Aplicadas, Sociedad Venezolana de Métodos Numéricos en Ingeniería, Caracas, Venezuela* (2008).

[4] Moltó, Román, and José Enrique. "Estructuras de Datos para Matrices Dispersas." (2008).

[5] Sparse matrix in matlab: design and implementation [online] URL: http://www.mathworks.com/help/pdf_doc/otherdocs/simax.pdf.

[6] Direct methods for sparse linear systems, pagina 8, timothy A. Davis. "Matrices Dispersas", [online] URL: http://www.fing.edu.uy/inco/cursos/comp1/teorico/2008/matrices_dispersas.pdf

[7] "Estructuras de Datos para Matrices Dispersas", José E. Román. [online] URL: http://polimedia.upv.es/visor/?id=e7eb821d-fc60-d149-9202-946361f18d95

[8] "Rutinas para Matrices Dispersas" [online] URL: http://telematica.cicese.mx/computo/super/calafia/programacion/dispersas.php

[9] "Procesadores Vectoriales", Universidad de Valladolid [online] URL: http://www.infor.uva.es/~bastida/Arquitecturas%20Avanzadas/Vectoriales.pdf