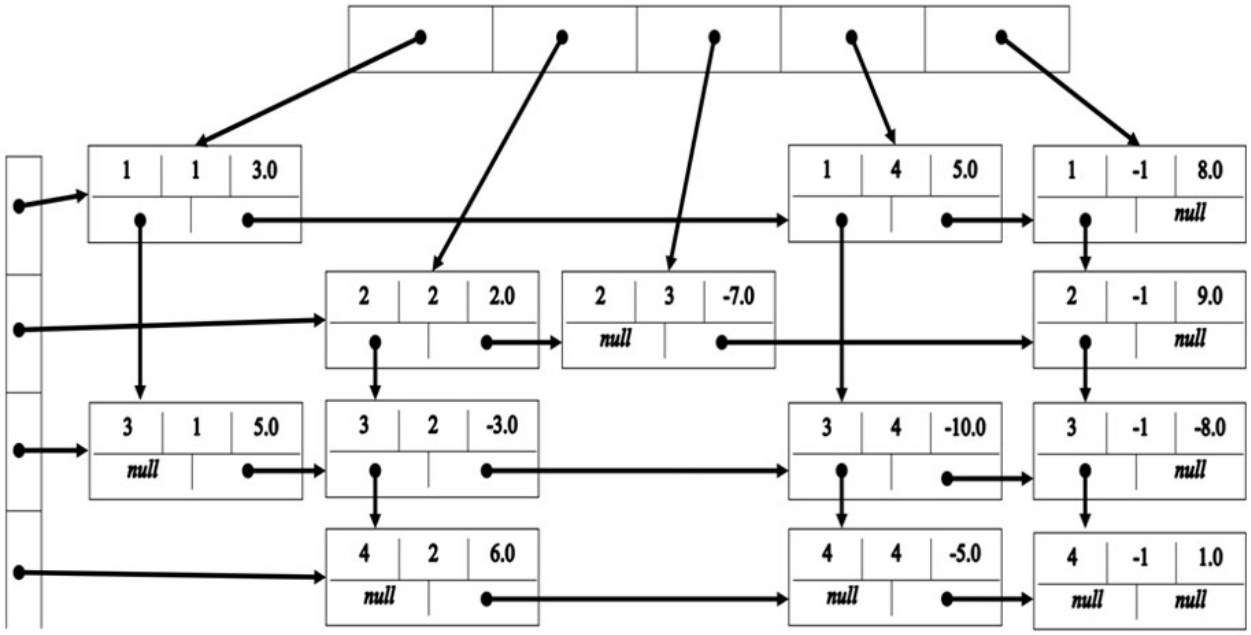
19AIE111 - DATA STRUCTURES AND ALGORITHMS - 1

SPARSE MATRIX AND SPARSE VECTOR USING LINKED LIST



DEEPTHI SUDHARSAN (CB.EN.U4AIE19022)

JAYASHREE O (CB.EN.U4AIE19031)

KALIGATLA SREE SAMHITHA (CB.EN.U4AIE19034)

PODURU RAMA SAILAJA (CB.EN.U4AIE19045)

CONTENTS

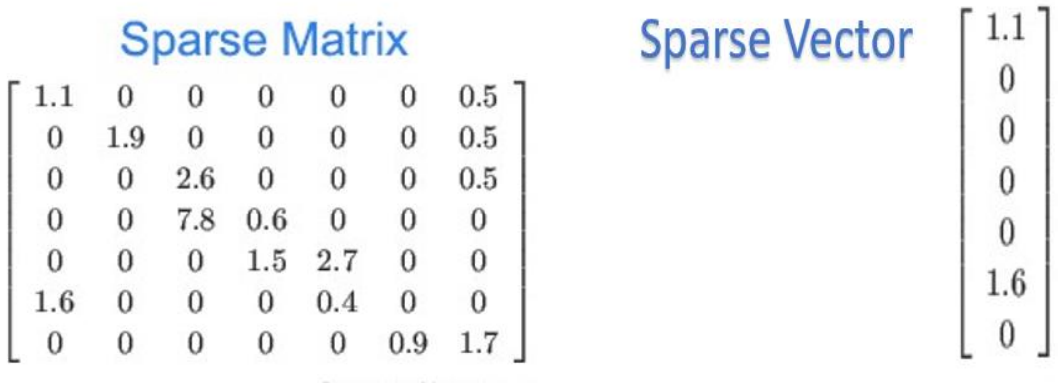
ACKNOWLEDGEMENT

For the completion of this project, we have been guided by our Data Structures - 1 faculty, Mr. Vijay Krishna Menon, who has provided us with sufficient knowledge and guidelines.

We would also like to thank our other professors for supporting this project and Amrita University for this opportunity to hone and display our skills.

ABSTRACT

Matrices with most of their values as zeros are called Sparse Matrices and similarly, vectors with most of their values as zeros are called Sparse Vectors.



$$\begin{bmatrix} 1.1 & 0 & 0 & 0 & 0 & 0 & 0.5 \\ 0 & 1.9 & 0 & 0 & 0 & 0 & 0.5 \\ 0 & 0 & 2.6 & 0 & 0 & 0 & 0.5 \\ 0 & 0 & 7.8 & 0.6 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1.5 & 2.7 & 0 & 0 \\ 1.6 & 0 & 0 & 0 & 0.4 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0.9 & 1.7 \end{bmatrix} \qquad \begin{bmatrix} 1.1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1.6 \\ 0 \end{bmatrix}$$

Designing code that performs operations on Sparse Matrices and Sparse Vectors such that the efficient and smarter usage of memory and minimization of memory wastage are taken into factor, is made easier using Linked List.

A linked list is a linear data structure where each element is a separate object. Linked list elements are not stored at contiguous locations; the elements are linked using pointers. It is a data structure consisting of a collection of nodes which together represent a sequence.

PROBLEM

Create Sparse matrix and Sparse vectors using linked lists. A matrix stores data with 2 indices row and column (r,c). Suppose the matrix is big, say 5000 x 5000 but only 100 values in it are non zeros, then such matrices with most of their values as zeros, are called sparse matrices. We need a list that finds the row and another that identifies a column, where the data is

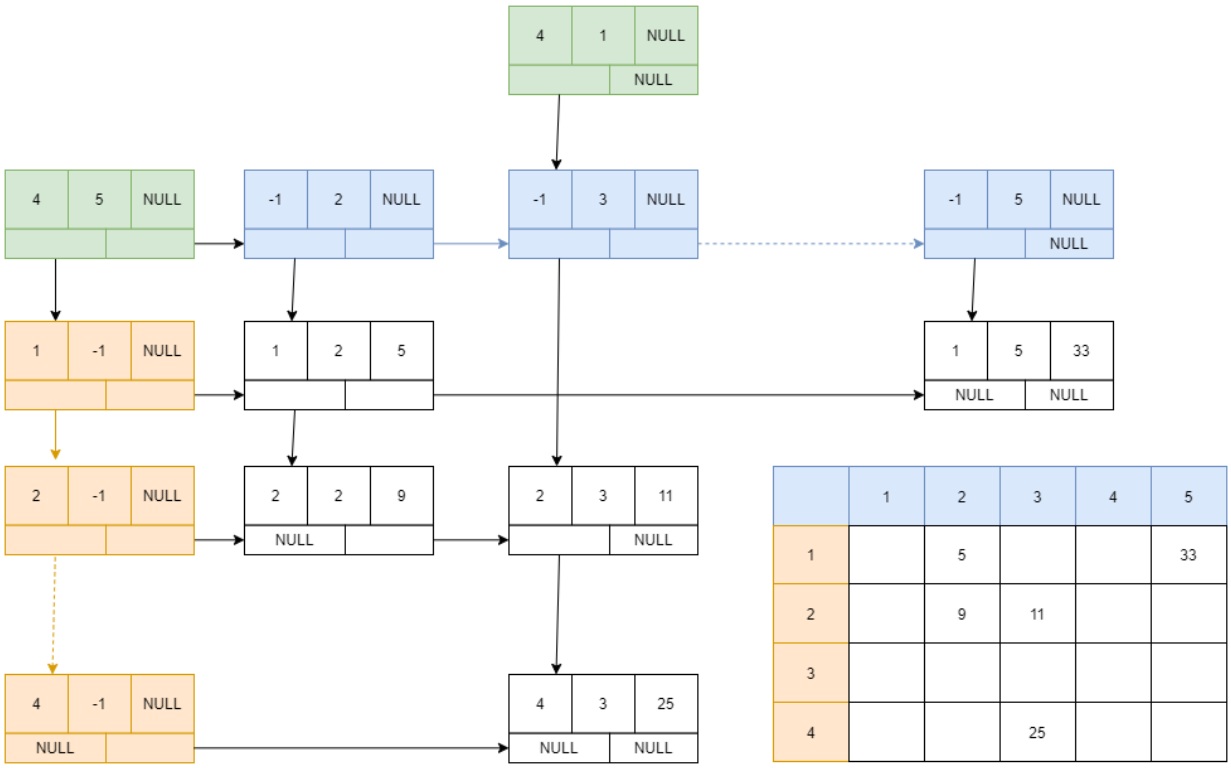
stored if non zero. The list will not store row, column indices that have zero value, hence saving space, which is the purpose of this data structure.

The final sparse matrix and vector class will need to implement, transpose, multiplyVector and multiplyMatrix for the matrix class and dotProduct for the vector class, as operations between them. The matrix class should also have a function to slice rows or columns from the matrix and return as vectors.

APPROACH

As soon as we are beginning to create our empty sparse matrix, the first step we take care to do is to create the main header node for the entire sparse matrix. As and when we add non-zero elements to the matrix, we make sure that the row and column headers for the element node have been created, if not, we create the headers and then create and link it to the new element node. When we want to slice a vector from the matrix, we create a vector row slice/ column slice header node to point to the vector in the matrix.

Once, the matrix has been created and the vectors have been sliced. Operations are performed on the matrix or vectors based on the user's choice.



The ORANGE HEADERS are the row header nodes, consisting of the row index and column index (is set to -1 by default as the “row headers” column isn’t considered as a part of the total number of columns of the matrix). And even the value for the header is set to null as headers don’t have element values stored in them.

The BLUE HEADERS are the column header nodes, consisting of the row index (it is set to -1 by default as the “column headers” row isn’t considered as a part of the total number of rows of the matrix) and column index. And even the value for the header is set to null as headers don’t have element values stored in them.

The GREEN HEADER at the top left is the header node for the entire matrix from which the column and row header nodes are begin and linked. This header node contains the total number of rows and columns of the matrix and default value like other header nodes (NULL).

The Green header node pointing at a single column is the column slice vector header node, pointing to the column slice vector. It contains the total number of rows of the matrix and number of columns as one along with some default value if it is a column slice and if it is a row slice vector, it contains the number of rows as one and total number of columns of the matrix along with some default value.

The WHITE BOXES represent the element nodes and store the current row index, column index and non-zero element value.

Apart from these, the headers also have two common nodes, the right node and the down/bottom node to link to the next column and row respectively.

WORKING OF THE CODE AND ITS FUNCTIONS

Node Class

We initialize the variables and create two nodes (down and right) and set them to NULL (initially it is an empty matrix so there are no rows or columns to point to).

DEFAULT_VALUE is set to zero; this variable is used to print zeros while printing the matrix to positions that haven't been assigned values or linked to (other than non-zero locations).

IGNORED_HEADER_INDEX is used to set the row index and column index of the column and row header to -1 so that it can be differentiated as the header nodes from the element nodes.

Since initially the total number of rows and columns are 0 (empty matrix) we initialize them to 0 and set the value to some random value (Integer.MIN_VALUE, a constant holding the minimum value an int can have, -2^{31}).

Two classes, Vector and Matrix Class, extend the Node class.

Vector Class

- **public Vector(int rows, int cols)**

This parameterized constructor of Vector class initializes the total number of rows and columns of the Vector.

- **public int size()**

If down is not equal to null and the total number of columns is one (column slice), then the function returns the total number of rows as the size of the vector. Else, it is a row slice, so it returns the total number of columns.

- **public int get(int index)**

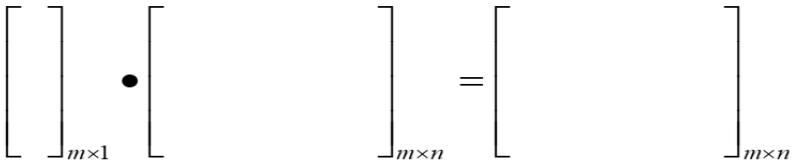
This function takes the index of the Vector as a parameter whose value is to be retrieved.

If it is a row slice vector (if the down pointer isn't null and col is equal to 1), we check if the index is out of bounds. If so, we throw a Runtime Exception. Else, curr is set to vector.down and while we keep traversing all the rows of the row vector simultaneously checking if we have not crossed the desired

row index. When the row index is equal to the index we are looking for we return the value at that index.

Similarly, if it is a column slice vector, we do the same process. But unlike the procedure for row slice, we point to the column until the desired column index.

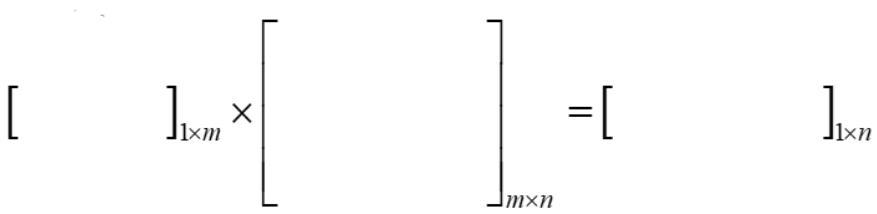
- **public Matrix dotProduct(Matrix rhs)**



In this function, we are checking if the row size of the Matrix (to which the Vector should be multiplied to) is equal to the Vector.size(),else ,we create a new Matrix object and call it's parameterized construction using the input matrix row and column size as arguments.

We run our nested for loop (one for looping through all the rows and one for looping through all the columns of the matrix). We get the value of the matrix one by one using the getValue function and multiply it with the respective row index (same row index of the Matrix whose value we just got) of the Vector. Then we set the row and column of the newly created resultant matrix with the computed value and the loop goes on. Once the looping is done, we return the resultant matrix.

- **public Matrix multiply(Matrix rhs)**



In this function, we are checking if the row size of the Matrix (to which the Vector should be multiplied to) is equal to the Vector.size(),else ,we create a new Matrix object and call it's parameterized construction with the row size as one and total columns of the input matrix as arguments (as resultant would be a 1 x rhs.col size matrix or row slice vector).

Then, using nested for loops (one for looping through all the rows and one for looping through all the columns of the matrix) we perform normal matrix multiplication and we set the resultant in the newly created resultant matrix using setValue function and the loop goes on. In the setValue function, we have given the row index as 0, as it is row slice and so there is only one row and we just have to set all the column values for that row.

We return the resultant Matrix.

- **public String toString()**

This function is for printing the row number,column number and the respective value of the vector in a proper format. The function first checks if down is not equal to null (that is, if we are not in the last row of the vector, here, if it is a column vector). If not, then we print the row index, column index and respective element value separated by comma and equal to sign for proper format. And then we set curr to curr.down (point to next row) and repeat the steps until the end of the vector.

Similarly, if it is a row slice, we print in proper format and traverse till the end of the vector by setting curr to curr.right (point the next column).

Matrix Class

- **public Matrix(int rows, int cols)**

This parameterized constructor initializes the number of columns and rows and sets the down and right pointer initially to null and also the matrix value is initially set to zero (empty matrix).

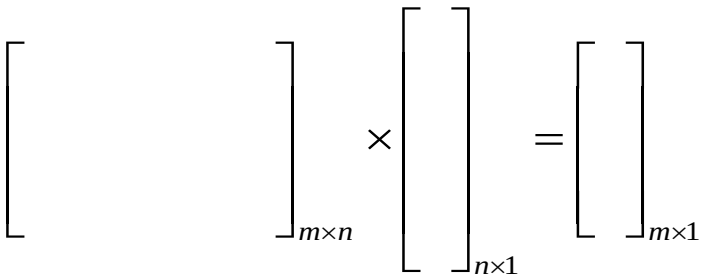
- **public int colSize()**

This returns the total number of columns of the matrix

- **public int rowSize()**

This returns the total number of rows of the matrix

- **public Matrix multiply(Vector vector)**



This function is used to multiply a matrix with a vector.

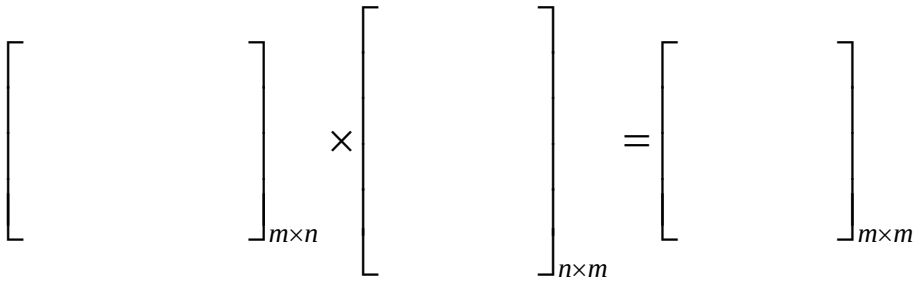
If the total number of columns of the matrix is not equal to the total number of rows of the vector (column slice vector), it will give an error as it can't be multiplied. Else we can multiply. We create a new Matrix object and call its parameterized construction with the row size equal to the total number of rows matrix and the total number columns in matrix equal to total number of columns in vector as arguments (equal to 1 because it is a column slice vector).

Then, using nested for loops (one for looping through all the rows and one for looping through all the columns of the matrix) we perform normal matrix multiplication and we set the resultant in the newly created resultant matrix using setValue function and the loop goes on. In the setValue function, we

have given the column index as 0, as it is column slice and so there is only one column and we just have to set all the row values for that column.

We return the resultant Matrix.

- **public Matrix multiply(Matrix rhs)**



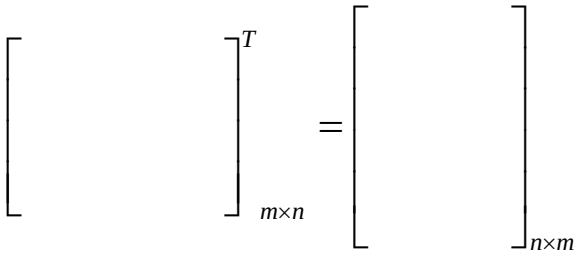
This function is used to multiply a matrix with a matrix

If the total number of columns of the matrix is not equal to the total number of rows of the input matrix(rhs), it will give an error as it can't be multiplied. Else we can multiply. We create a new Matrix object and call its parameterized construction with the row size equal to the total number of rows of the matrix and the column size as the total number of columns in the input matrix .

Then, using nested for loops (one for looping through all the rows and one for looping through all the columns of the matrix) we perform normal matrix multiplication and we set the resultant in the newly created the resultant matrix using setValue function and the loop goes on.

We return the resultant Matrix.

- **public Matrix transpose()**



We create a new Matrix object and call its parameterized construction with the row size equal to the total number of columns of the matrix and the column size as the total number of rows in the matrix . (we are giving the opposite row and column size as this will be the resultant matrix that will store the transpose matrix.)

While the right pointer is not equal to null (i.e., the right pointer is pointing to any column or we are not in the last column) then we are setting curr to curr.right or the column it is pointing to and we create a new node called column and we set it to curr. Now that we are pointing to a particular column we check using another while loop if we are not in the last row or the column (i.e., the down pointer is not equal to null).

We keep traversing down the column until we reach the last row, and we simultaneously keep setting the column values as the row values of the newly created matrix using the setValue function.

We return the resultant Matrix.

- **public int getValue(int row, int col)**

Given the row and column this function returns the value stored in it. The while loop used here is to check whether the matrix is not empty or not pointing to the last column. This loop continues to point to the next column until it finds the required column index. Now, once the required column is found the next while loop in the function checks for the row index by making sure the down pointer is not pointing to the last row or null. The element stored in the required row and column is then returned. If the row

and column size is within the range of the matrix and there is no node pointing to the address then it returns default value, that is zero.

- **public void setValue(int row, int col, int value)**

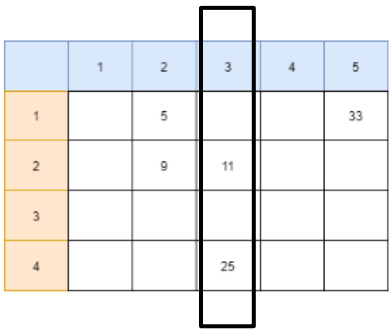
	1	2	3	4	5
1		5			33
2		9	11		
3					
4			25		

This function stores the given value in the given row and column. If the given row and column index is out of the range of the matrix then out of bound exception is thrown. If the range is satisfied but there is no element node then the corresponding header are created and the value is stored.

- Given public void setValue(1,1,4),then the value 4 is set to (1,1) position.
- Given public void setValue(6,6,2),then out of bound exception is thrown.
- Given public void setValue(3,4,5),then column and row header are created and 5 is stored in (3,4).
- Given public void setValue(2,4,7),row header already exists and column header is created and 7 is stored in (2,4)

- **public Vector colSlice(int col)**

this function returns the values in the given column. If the column is not in the range of the matrix then out of bound exception is thrown. For example, if column 3 is the selected column then [0,11,0,25] are returned.



	1	2	3	4	5
1		5			33
2		9	11		
3					
4			25		

- **public Vector rowSlice(int row)**

This function returns the values in the given row. If the row is not in the range of the matrix then out of bound exception is thrown.
For example, if row 2 is selected then [0,9,11,0] are returned.

- **public String reshapeM()**

Shaping the output in proper matrix format . This function gets the values of the matrix from the getValue() and formats the output in a properly formatted matrix form.

- **public String toString()**

The while loop used here is to check whether the matrix is not empty or not pointing to the last column. This loop continues to point to the next column and prints the Matrix size(giving row and column index) in proper format.

public Matrix makeMatrix(int rows, int cols)

Creates an object of Matrix class, calling the Matrix parameterized constructor with the rows and columns given by the user as arguments and creates an empty matrix.

public static void main(String[] args)

We [here, in main] are creating an object sp of class SparseMatrix. We are calling the respective functions to create the matrix, set values, retrieve value and also restructure and display the matrix properly. We are also slicing a row vector and column vector from the matrix, getting values from the vector, and we are also performing a sample transpose, dot product and vector-matrix multiplication.

APPLICATIONS

1. By search engines (google) to rank web-sites to improve searches
2. Solving partial differential equations using the finite element method
3. Dimensionality Reduction (DR) techniques
4. Computer-aided control systems design

REFERENCES

1. https://web.njit.edu/~avp38/projects/mae/sparse_matrix.html
2. https://www.youtube.com/watch?v=ya-R_75BrLA

WORK BREAKDOWN STRUCTURE (WBS)

The code and all of its functions were designed by all of us equally. We divided the functions, classes and approach equally amongst ourselves while also helping each other out whenever required. The report and powerpoint was also jointly worked on by all of us, but we made sure we exchanged each other's parts (that we worked on during the coding and compilation process) so that each one of us could get to work on the entire project and code.