University of Rhode Island

CSC 212

Department of Computer Science

Sparse Matrices with Linked Lists

April 25, 2022

Authors:

Demetrios Petrou, Justin Watkins,
Lisandro Nunez, Frederick Herchuk

Contents

1 Introduction

This final project group was tasked with creating the representation of a sparse matrix utilizing a linked list structure with customized nodes for storing related data essential to maintaining optimal operability in the sense of matrix multiplication and addition. There were a myriad of secondary goals for this project which included: proper data structure functionality, algorithm analysis for the group's implementation, a deeper dive into next steps for optimization of sparse matrix calculations, and exploring real world applications revolving around sparse matrices and being able to more effectively deal with them.

The goal of proper data structure functionality is centered around securing a more formalized approach to implementing code utilizing C++ resources to allow for a matrix container class suited for sparse matrices and some of their correlated functions. The algorithm analysis for the code implementation includes comparisons to utilization of already established container classes and their comparative runtimes on sparse matrices. This portion of the project included overall asymptomatic analysis to secure essential knowledge like big O functionality of the code developed over the course of this project. The steps taken past the point of analysis were all related to the further use of sparse matrix operations utilized in a wide range of applications. This paper will specifically address the application and optimization side of the linked list implementation and solving systems of linear equations in regards to machine learning algorithms.

1.1 What are matrices?

Matrices are two-dimensional arrays of scalars with one or more columns and one or more rows. These scalars are usually defined as elements or entries of the matrix. Their structure provides a "table" of values defined by its established structure of m number of rows by n number of columns.

A matrix is defined by the notation of a capital letter referencing the entirety of the matrix itself and the same letter in lowercase form with a double subscript of i and j to define the i-th row and the j-th column of the matrix position where the element the letter represents resides inside of the

matrix.[1]

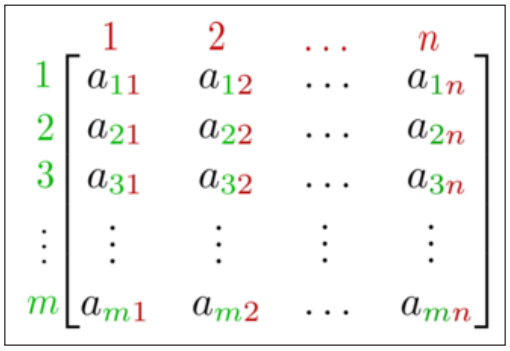

Figure 1: The depiction above replaces the i subscript with m directly and the j subscript with n directly.[1]

1.2 What are sparse matrices?

Matrices which contain mostly zero values are considered sparse matrices. These matrices are considered sparsely populated with non-zero values. A rule of thumb is usually if the matrix is composed of greater than fifty percent of zero value elements, the matrix can be considered a sparse matrix. Matrices which hold primarily non-zero values are considered dense matrices.

$$\begin{pmatrix} 1.0 & 0 & 5.0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 3.0 & 0 & 0 & 0 & 0 & 11.0 & 0 \\ 0 & 0 & 0 & 0 & 9.0 & 0 & 0 & 0 \\ 0 & 0 & 6.0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 7.0 & 0 & 0 & 0 & 0 \\ 2.0 & 0 & 0 & 0 & 0 & 10.0 & 0 & 0 \\ 0 & 0 & 0 & 8.0 & 0 & 0 & 0 & 0 \\ 0 & 4.0 & 0 & 0 & 0 & 0 & 0 & 12.0 \end{pmatrix}$$

Figure 2: The matrix depicted above has only 12 non-zero values and 52 zero valued elements out of the total number of 64 elements in this 8x8 matrix meaning 81.25% is populated by zeros, thus it is considered a sparse matrix.[2]

1.3 Sparse Matrices' Relevance to Real-World Data

The basic definition of a matrix being defined as m number of rows and n number of columns establishes a predefined dimensionality of this structure. In the case where additional data points are to be added, and they exceed the m by n capacity, a new matrix of increased capacity must be made in either an additional row, column, or both. Effectively, when a matrix is made

and exceeds the initial dimensionality described to it, the formation of new columns or rows lends itself to needing rows and columns to be filled with many other zero values to properly fill in the matrix to its full dimensions. This fact lends itself to sparse populations of data sets with non-zero values when being utilized to store real world data such as connections between components in a system (where there are a large number of components but few connections between each component) and generally for mathematical models like finite element modeling and general systems of partial differential equations. Sparse matrices find a role within a wide range of computational tasks due to their continual reemergence in real world data sets for a myriad of subjects from natural language processing to circuit simulations.

For instance, one basic use is for recommender systems. Recommender systems are algorithms used to suggest relevant content or items to users. In most cases they use customer data to predict what content a user may be likely to purchase/watch/interact with next. These systems work with sparse matrices as the number of items offered in a specific company's ecosystem is quite large whereas the number of items a user interacts with out of the overall set of items is a very small subset. If you were to imagine the data Netflix's recommender systems use, they would include the representation of each user's profile of viewed content as a row spanning a long range of columns which each represented a particular film or show. The column's range of values would span across all of the shows which Netflix currently has and each element would be represented as a 0 for unwatched or 1 for watched. In this scenario, each row representing a user would have interacted and watched only a very small fraction of the total content Netflix actually has; thus, the recommender system's initial data set is filled with sparsely populated row entries for each user.[3]

	item-1	item-2	item-3	item-4	...	item-6	item-7	item-8	item-*m*
user-1	1	0	0	0	...	0	1	0	0
user-2	1	0	0	0	...	0	0	0	1
user-3	0	1	0	0	...	1	0	0	0
...	...	...	...	...	...	...	...	...	...
user-5	0	0	1	0	...	0	0	0	1
user-6	0	0	0	1	...	0	1	0	0
user-*n*	0	0	0	1	...	0	0	1	0

Figure 3: Recommender system "sparse matrix" for multiple users.[3]

Another application of sparse matrices is in electrical circuit analysis where software tools for electrical engineers solve systems of partial differential equations using methods like the Newton-Raphson method or better yet the conjugate gradient method. The use of matrices allows you to represent circuitry components in matrix form where most of the connections can be mapped out using potentials, branch currents, and resistances throughout the circuit simulation. However, there are numerous components of a circuit which are connected to each other in very limited ways, which lead to positional matrix representation of the circuit's values to be sparsely populated.

The following examples illustrate matrix multiplication in the form of Ax = b in order to solve a system of linear equations representing values along the circuit below. Consider the circuit in Figure 4.

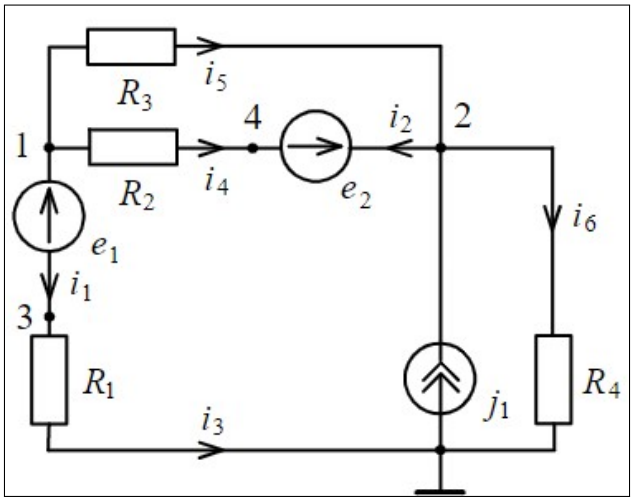

Figure 4: Example electrical circuit[4]

We would positionally represent the nodes, resistive components, and

sources and produce a solution for the x vector representation for the produced currents and electric potentials across this circuit.

Referencing back to the matrix form of a system of linear equations (Ax = b): A equates to the matrix containing the R (resistive matrix), K (incidence matrix and its transpose) and the 0 submatrix; b equates to the known sources values for electromotive forces (emf) represented by the e matrix and the currents represented by the j matrix; whereas x represents the i which is the branch currents vector and which is the electric potential vector which is trying to be solved for in this circuit.[4]

$$\begin{bmatrix} -\mathbf{R} & \mathbf{K}^{\mathbf{T}} \\ \mathbf{K} & \mathbf{0} \end{bmatrix} \begin{bmatrix} \mathbf{i} \\ \boldsymbol{\phi} \end{bmatrix} = \begin{bmatrix} \mathbf{e} \\ \mathbf{j} \end{bmatrix}$$

Figure 5: Matrix relations between circuit characteristics.[4]

The matrix representation will be similar to the sparse matrix in Figure 6 below.

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & -1 \\ 0 & 0 & -R_1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -R_2 & 0 & 0 & 1 & 0 & 0 & -1 \\ 0 & 0 & 0 & 0 & -R_3 & 0 & 1 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -R_4 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & -1 & 1 & 0 & 0 & 0 & 0 \\ -1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} i_1 \\ i_2 \\ i_3 \\ i_4 \\ i_5 \\ i_6 \\ \phi_1 \\ \phi_2 \\ \phi_3 \\ \phi_4 \end{bmatrix} = \begin{bmatrix} e_1 \\ e_2 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ j_1 \\ 0 \\ 0 \end{bmatrix}.$$

Figure 6: Sparse matrix expansion of circuit.[4]

Here it can be seen first hand how data that is represented to take into account practical situations like a simple circuit can end up being represented as a sparse matrix. Generally speaking, in real world applications data which can be put into matrix form for computational purposes like solving systems

of linear/non-linear equations or simply tracking large sets of data that have very little interaction comparative to their overall amount of data, will more than likely be represented by a sparse matrix.

1.4 Linked List Implementations

Linked lists are a basic data structure concept used in OOP which allow a sequence of nodes to be connected via links. These nodes contain both a data structure that holds data in numerous forms depending on the structure chosen (which holds an element) and a pointer/"link" to the next node in the sequence of nodes. The linked list itself contains the original head pointer/"link" which points to the first node object in the sequence of the linked list. These linked lists store nodes in non-contiguous locations in memory as the link in each node provides the method of accessing all elements in the sequence but the access to each node is only given by either the initial head pointer at the beginning of the sequence (or potentially a tail pointer at the end of the sequence). These structures can grow and shrink dynamically with fast insertions and deletions but longer accessing times.[5]

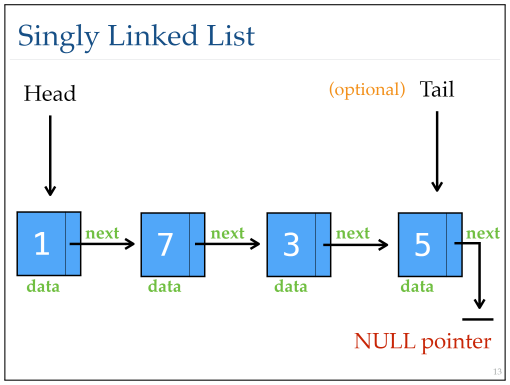

Figure 7: A singularly linked list with the optional tail pointer.[6]

The reason behind a linked list's advantageous insertion and deletion times is due to the structure of the linked list and the comparison to what occurs in an array or vector container class. Whenever an element is deleted or inserted into a linked list there is a constant number of operations of manipulating the pointers between nodes and properly deleting or creating a node object via dynamic memory allocation procedures. Plus, with the addition of a tail pointer, at the end of a linked list, insertion can become constant also. Whereas, with a vector or an array there are a few issues which

cause linear time complexity. For instance, to delete or insert within a vector or array, the class must recopy all of the data in the container to exclude the value deleted or include the value inserted (if not inserted at the end) and move all the data values in the indexes after deletion back by one or up by one after insertion. Also, in order to dynamically grow beyond the certain set allocation of contiguous memory these containers have, these structures must recreate themselves by iterating through and copying every piece of data in their structure and reinserting the data into a new container of their type with the data being added. The process of recopying the entirety of data (called reallocation) and the general processes of insertion and deletion times for dynamic arrays and vectors give a linear time complexity. Plus, due to the non-contiguous nature of linked lists, they more efficiently use memory instead of having to save large amounts of data to store contiguous memory container classes.

However, traversing a linked list and accessing each of its individual nodes is a majorly significant downside which has a linear time complexity compared to that of a vector or an array with constant time complexity. The reason for this is due to the fact that non-contiguous memory is used to store the nodes of a linked list and getting to any point in a linked list is dependent on traversing the chain of pointers from the head pointer to whatever element on the linked list in question. Thus, in order to get to the i-th element of a linked list, the process always starts from the head pointer and goes down every other node and its corresponding next pointer until the i-th element is accessed. Attachment of a tail pointer lessens the worst case to O(n/2) as the linked list can access either half of the array in n/2 traversals where the first half goes from the head pointer to the element index attempting to be accessed, and the second half goes from the tail pointer to the element attempting to be accessed.

SINGLY LINKED LIST OPERATION	REAL TIME COMPLEXITY	ASSUMED TIME COMPLEXITY
Access i-th element	O(√N * N)	O(N)
Traverse all elements	O(√N * N)	O(N)
Insert element E at current point	O(1)	O(1)
Delete current element	O(1)	O(1)
Insert element E at front	O(1)	O(1)
Insert element E at end	O(√N * N)	O(N)

Figure 8: Time complexities for linked lists.[7]

There are other implementations such as the doubly linked list where there are two pointers stored in each node with one pointing to the next node in the sequence and secondary one pointing to the previous node also. The other modification that can be made is creating a singularly circular linked list where the last node contains a pointer/"link" of the first element and a doubly circular linked list which includes the pointer in the last node which points to the first element and also has the first node contain a pointer/"link" to the last element as the previous pointer in the sequence. The main difference between these implementations is slight variations in access time and slightly larger memory requirements for each extra pointer stored in each node.

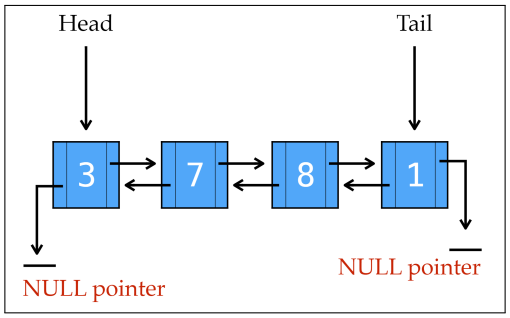

Figure 9: Doubly linked list.[6]

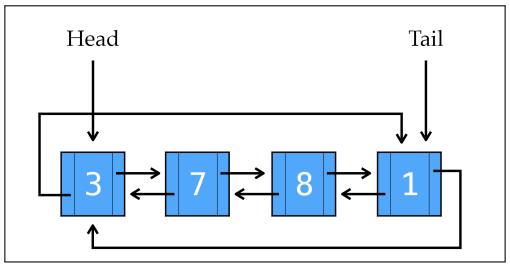

Figure 10: Circular singly linked list.[6]

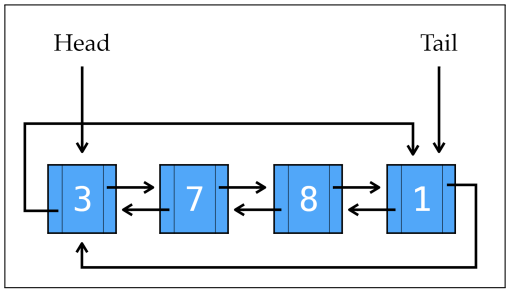

Figure 11: Circular double linked list.[6]

2 Sparse Matrices with Linked Lists Class

The intuitive way to imagine matrices for most people is a 2D array or vector that represents a grid of boxes nicely. However, are arrays or vectors always the best choice for sparse matrix representation? In general, alternative representations of certain data structures may seem more convoluted and take up more lines of code, but they may be better-suited to a certain application in terms of time and space complexity.

2.1 Why sparse matrices and linked lists?

Sparse matrices appear in numerous real world data sets and applications. In order to properly utilize them, they require specialized data structures and algorithms to deal with them more efficiently and take advantage of their sparse nature to reduce memory storage and computational power needs.

A practical way of representing them is through linked lists. There are numerous data structures for sparse matrices which use linked lists in some form or another with different advantages based on the application. The decision to use linked lists in these representations is based on trading the longer access times of a linked list traversal with the non-contiguous memory advantage and elimination of the storage of zero values in the matrices.

With a linked list implementation of matrices specifically geared for sparse ones, there are two formal types which can be utilized called List of Lists (LIL) and Coordinate Lists (COO).

The List of lists (LIL) is formatted so that the structure stores one linked list per row. These rows of elements are stored usually in another data structure like a vector which will hold the head pointers of each linked list. Within this vector of rows, an empty space in the vector can be allowed to show a row containing no non-zero values and allows the index of the vector to identify the row index in the matrix. This implementation allows for faster access time to each row to reduce row traversal to a constant time and the linked list traversal to simply O(n) where n is the number of non-zero entries in that row. There are other variations which can utilize linked lists themselves to store rows also where the node of the linked list storing rows

would simply contain the row index value and two pointers, one which is the head of the linked list containing the non-zero elements of that row index and one to the next row index value linked list node. This would be a direct application of a linked list of linked lists. There are numerous variations to this format that can be utilized based on application need and memory versus access time requirements for the programs being implemented.

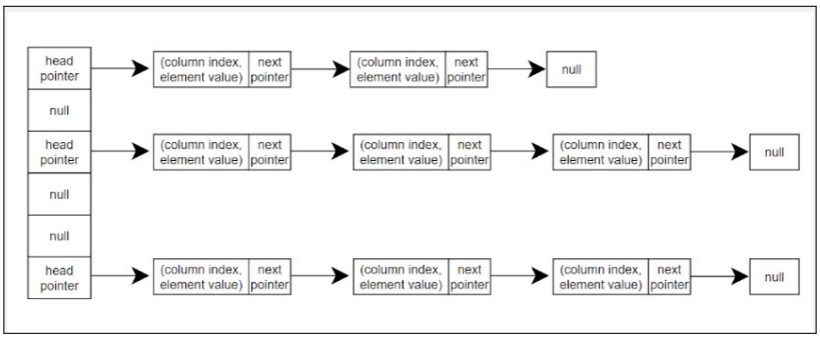

Figure 12: Vector of Linked Lists.

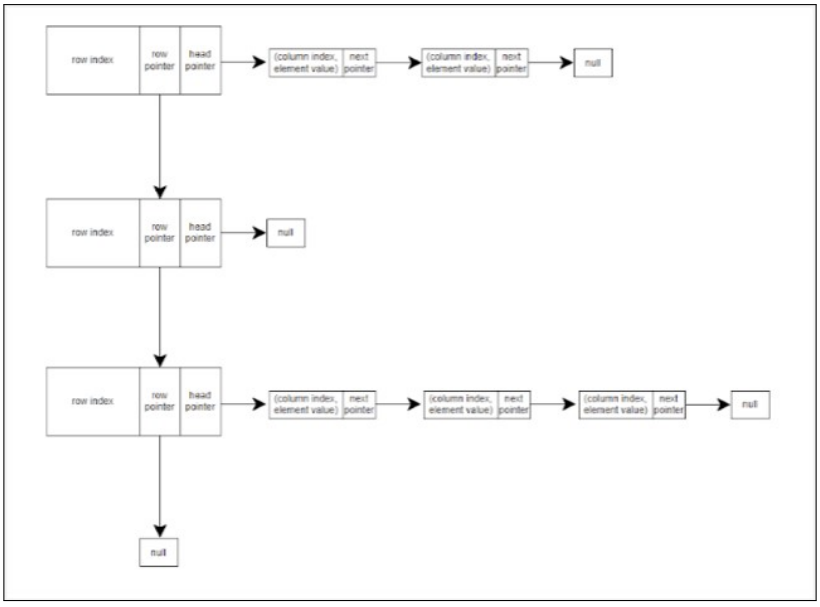

Figure 13: Linked List of Linked Lists.

The other way linked lists can be utilized is by taking advantage of the node functionality and attempting to contain all needed data within a single linked list structure. This method is the Coordinate List method (COO) where every node contains three basic data members apart from the necessary pointers: an element value, the row coordinate of that element, and the column coordinate of that element. With this simple information, everything

that is done with a normal vector matrix can be done with a linked list. The trick is that all non-zero cells are not included in the linked list, thus eliminating most of the wasted memory used in vector matrices. Yes, the implementation will be tedious and the computation for each single operation will be more convoluted. However,because of the immense difference in memory usage, it will result in a far more efficient sparse matrix data structure.

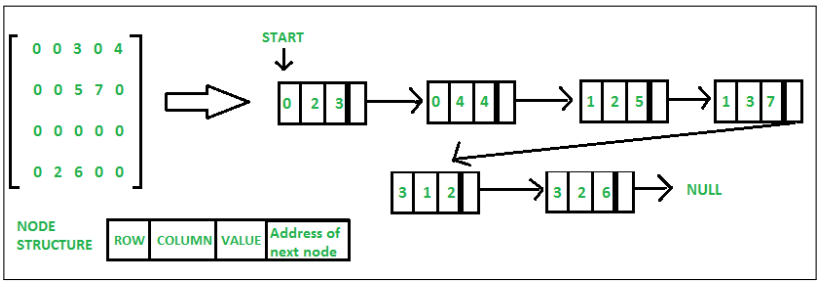

Figure 14: Linked List of Linked Lists.[8]

These implementations of sparse matrix storage can also be complimented with their array counterparts of the Yale format, compressed sparse row (CSR or CRS), and compressed sparse column (CSC or CCS). The difference between the two types of data structures used revolves around the goal to either support efficient modification of the elements or efficient matrix operations of the elements.[9] These linked list methods fall more so into the category of modification and initial storage and creation of the sparse matrix representation. The project's goal was to implement things according to the specifications of reduced memory usage and implementation of linked lists so the group's main code implementation is formatted more along the latter version of a COO structure with slight modifications to allow for somewhat easier matrix operations.

2.2 Class Structure

To visualize the prior explanation and add onto it, Figure 7 shows a visual representation of the classic vector matrix and Figure 8 the class structure for our linked list matrix.

The far smaller linked list encompasses all the information in the vector matrix in a much more concise manner. As you can see, the next pointer points to the proceeding non-zero cell from left to right and going down. The

below pointer is less intuitive in the linked list format, but it simply points to the proceeding non-zero element from up to down and to the right. This extra pointer will be important during the matrix multiplication method.

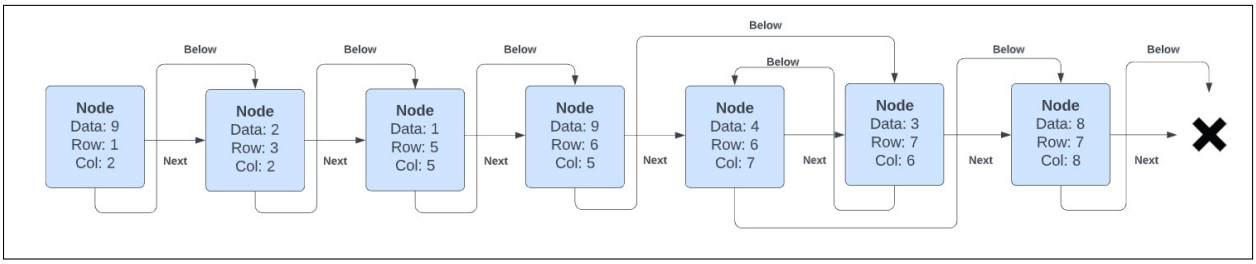

Figure 15: Vector matrix visual representation

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	9	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0
3	0	0	2	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	1	0	0	0
6	0	0	0	0	0	9	0	4	0
7	0	0	0	0	0	0	3	0	8

Figure 16: Linked list matrix visual representation with next and below pointers

Apart from the main data structure, there are other helper functions, the main addition and multiplication operations, and other I/O methods. These will be discussed in Methods and Implementation.

2.3 Class Features

There are a few important features about our class that should be mentioned. These include an additional pointer for a secondary method of matrix traversal, matrix read and write functions from and to text files, a precedence factor variable for insertion and I/O, random matrix generation, and the ability to insert elements into the matrix in any order.

First, the below pointer is an additional data member of our Node class. Instead of pointing to the next node in row-major order, it points to the first node in column-major order. This pointer will be very helpful with

matrix multiplication since the below pointers at each node can be exploited to perform the foundational row times column operations. Additionally, a vertical head pointer parallels the main head pointer as the first node in column-major order. Many times, the two head pointers point to the same node, but they are important to have for their respective kinds of matrix traversal.

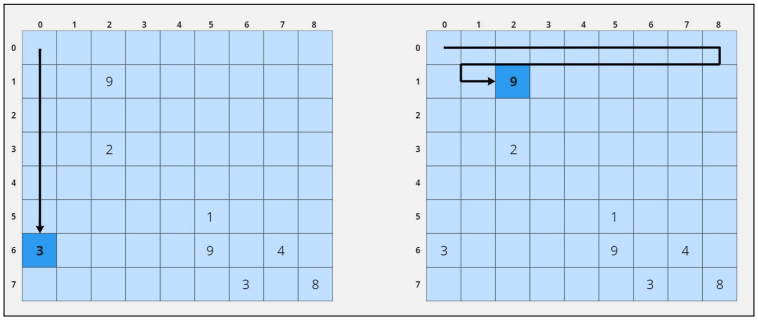

Figure 17: Row-major and column-major head pointers

Another important feature of our class is the input/output methods or reading/writing from and to text files for data management. The Matrix class is able to initialize a list matrix from a given file in the constructor, and it can also read in a file after instantiation. It is convenient to have the ability to write to text files as this provides a very general form of data storage.

An important feature that goes along with input and output along with insertion is the precedence factor variable. This is based on the idea of row-major and column-major order to understand where elements should be placed or where in the list matrix we are. The variable is based on the row or column dimension of the matrix, whichever is greater. It is used by multiplying it by the row (for row-major) or column (for column-major) and adding that product to the column or row dimension that was not used. This gives a sort of "numerical precedence" to each cell. In a vector matrix representation, this is unnecessary since any index can be accessed at any time in any order unlike a list matrix.

For ease of testing the class and its functionality, a random matrix generator is also included that uses the ¡random¿ library to generate matrices. It takes in user input for dimensions, data bounds, and number of non-zero elements as well. Though not practical or efficient for extremely large dimension inputs, it is a convenient method for testing.

A final interesting feature of our Matrix class that is not used in practice is the ability to insert nodes in any order. In other words, our insert method can squeeze a node in if its precedence factor dictates so. However, this feature is not utilized in our class since the insert method is private and is only used when creating a matrix from a text file or the random generator. Further applications could make use of our versatile insertion.

3 Methods and Implementation

3.1 Node Search and Insertion

In our current implementation, insertion only occurs when a text file is read in and when a random matrix is generated. The function's parameters are the new node's row, column and data. The precedence factor in this case is used to determine a node's position relative to row-major order in this case. The below pointer of a node is not set in the insertion method. The process for the below pointer is similar, but it uses the precedence factor relative to column-major order instead.

```
addElement (data, row, col)
        if row > max_row then
                set_max_row (row)
        end if
        if col > max_col then
                set_max_col (col)
        end if
        set precedence_factor
        declare new_node (data, row, col)
        declare curr = head
        if (list_size == 0) then
                head = new_node
                return
        end if
        while curr->next != nullptr && new_node_precedence < curr_precedence do
                curr = curr->next
        end while
        if curr_precedence == new_node_precedence then
                curr = new_node
        end if
        else if curr_precedence > new_node_precedence then
                curr->previous = new_node
        end if
        else
                curr->next = new_node
        end if
```

Figure 18: Pseudo-code for Node Search and Insertion

3.2 Matrix Addition

Matrix addition is fairly straightforward. Two matrices are added with the overloaded addition operator, and their result is assigned to a new Matrix object. The actual addition involves a nested loop that uses the i and j control variables to check if the next nodes in each matrix match the control variables. If they both match, their data is added and added to the result matrix. If one of them matches, its data is added to the result matrix. As the data at each node is used, the next node in that matrix is accessed.

```
addition (Matrix one, Matrix two)
    declare Matrix result
    if one_dimensions != two_dimensions then
        return result
    end if
    else
        curr_one = one_head
        curr_two = two_head
        for i in row_dimension do
            for j in col_dimension do
                if curr_one_row == i && curr_one_col == j && curr_two_row == i && curr_two_col == j then
                    result.add_element(curr_one_data + curr_two_data)
                    curr_one = curr_one->next
                    curr_two = curr_two->next
                end if
                else if curr_one_row == i && curr_one_col == j then
                    result.add_element(curr_one_data)
                end if
                else if curr_two_row == i && curr_two_col == j then
                    result.add_element(curr_two_data)
                end if
            end for
        end for
    end if
    return result
```

Figure 19: Pseudo-code for Matrix Addition

3.3 Matrix Multiplication

Traversal through the list matrix for multiplication is a bit more convoluted because one matrix needs to be traversed in row-major order and the other in column-major order. It is important to note that the first two layers of loops iterate along the rows of the first matrix and the columns of the second matrix to calculate each cell as these are the dimensions of the resultant matrix according to the rules of matrix multiplication. The third inner loop iterate along the columns of the first matrix and the rows of the second matrix simultaneously since they must be equal according to the rules of matrix multiplication. The actual multiplication and addition of those cell-by-cell multiplications occurs in this third loop. One other important thing to note is that there are two temporary Node pointers used to traverse through each list matrix. The first two keep track of the specific cell being calculated and the other two keep track of the row or column (first or second matrix) being traversed to calculate a specific cell.

```
multiplication (Matrix one, Matrix two)
    declare Matrix result
    if one_col != two_row then
        return result
    end if
    else
        curr_one = one_head
        curr_two = two_vertical_head
        for m in one_row do
            for n in two_col do
                curr_one_a = curr_one
                curr_two_a = curr_two
                sum = 0
                for k in (one_col or two_row) do
                    if curr_one_a_row == m && curr_one_a_col == k && curr_two_a_row == k && curr_two_col == n then
                        sum += curr_one_a_data * curr_two_a_data
                        curr_one_a = curr_one_a->next
                        curr_two_a = curr_two_a->next
                    end if
                    if curr_one_a_row == i && curr_two_a_col == j then
                        if curr_one_a_col <= k then
                            curr_one_a = curr_one_a->next
                        end if
                        if curr_two_a_row <= k then
                            curr_two_a = curr_two_a->next
                        end if
                    end if
                    else
                        break
                    end if
                end for
                while curr_two_col <= j do
                    curr_two = curr_two->below
                end while
                add_element(sum)
            end for
            while curr_one_row <= i do
                curr_one = curr_one->next
            end while
            curr_two = two_vertical_head
        end for
    end if
    return result
```

Figure 20: Pseudo-code for Matrix Multiplication

3.4 Input and Output

For our readFile() and writeFile() methods, we adapted the I/O files provided by Christian earlier in the semester to work with linked lists. One significant difference is that the below pointers for each node are set at the end of the readFile() function once all the nodes have been added.

3.5 Data Set Generation

To test the power of our linked list matrix class, we developed a random matrix generator based on user input for matrix dimensions. The user can

also input the bounds for the non-zero elements and how many non-zero elements to include. Using the C++ random library, the method generates a non-zero integer and uses the addElement() method to add it to our new Matrix. The process is very similar to how readFile() creates a Matrix and then adds the below pointers after the fact. We used this method for our comparison testing with a vector-based matrix later on.

3.6 Helper/Additional Functions

There are various helper functions in our implementation that complement matrix addition, multiplication, insertion, and the class in general. The first two are setMaxRow() and setMaxCol() which set the dimensions of the overall matrix whenever they need to be changed. For example, if a new element is read in with a higher column index than any other element, the column dimension must be resized. The two mirror functions getMaxRow() and getMaxCol() do the opposite by returning the matrix dimensions. There is also a getSize() method that returns the number of nonzero elements in the matrix.

The other main supplementary function is printMatrix(). This traverses through the linked list in parallel with a nested for loop and prints an element every time the loop control variables (i and j) match up with the next node's coordinates. If the current loop control variables do not match, a 0 is printed as this is the assumption of how our list matrix is designed. This is a very helpful method for quick visualization of the matrix.

4 Benchmark Testing with Vector-Based Matrix Class

In general, the most significant aspect of our data structure is not necessarily the matrix multiplication, addition or any other singular algorithm or computation using matrices. It is significant because of the drastically decreased memory usage and consequently decreased run-time for our methods or any other application. To showcase this main characteristic, we implemented a functionally identical matrix class using a 2D vector as the foundational data structure as a basis of comparison. The implementation was obviously much more straightforward but lacked the efficient memory usage of our list matrix class.

To formally compare the two, we used the ctime library and our random matrix generator to graph the run-time of our methods as input dimension changes. In all cases, we collected 25 readings from a dimension input of 1 to 5000. This input range was based on two factors. First, many sparse matrix data-sets in the real world are within this range; second, it was not feasible to create larger matrices within our time constraints because of the large run-time of our random matrix generator.

4.1 Vector-Based Matrix Class

As mentioned above, the vector-based matrix class contained all the same functionality as the list matrix class, but the implementation was much simpler due to a 2D vector's structure that naturally lends itself to matrix representations. This structure eliminates the need to consider row-major or column-major order when traversing or accessing a part of the matrix since simple indices take care of one cell's relation to the rest of the matrix.

4.2 Time Complexity Analysis

By analyzing our node search and insertion, matrix addition, and matrix multiplication methods, we concluded on the following average time complexities for each.

Method	**Vector Matrix**	**List Matrix**
Node Search and Insertion	O(1)	O(n)
Matrix Addition	O(mn)	O(mn)
Matrix Multiplication	O($m_1 n_2 k$)	O($m_1 n_2 k$)

The constant time for insertion with the vector matrix is due to the fact that a search is not needed as vectors have constant time access. However, the list matrix requires a search to be made to place the node in the correct order based on its coordinates and precedence with row-major order. This creates a linear time complexity. In matrix addition, m stands for the number of rows in the matrix and n stands for the number of columns. The time complexity is seemingly the same, but in a list matrix, the actual number of operations performed is far smaller resulting in a realistically more costly run-time. In matrix multiplication, m1 stands for the number of rows in matrix one, n2 stands for the number of columns in matrix two, and k stands

for the number of columns in matrix one or rows in matrix two as they should be equal according to matrix multiplication rules. Again, the actual number of computations that occur with list matrices is drastically smaller than with vector matrices and even more so than with matrix addition.

Our comparison testing used the vector-based and list matrix classes, the ctime library, our random matrix generator, and the Plotly tool introduced in Lab 2 earlier in the semester. The protocol was based on the following guidelines, and the results of our testing are displayed on the proceeding figures.

- Time both operations at an input dimension size of NxN for 0-5000 and store the times in a CSV file.
- Collect data for sizes of N spaced out by increments of 200.
- Plot the list and vector run-times over each other using Plotly.

Figure 21: Time complexity plot for matrix addition

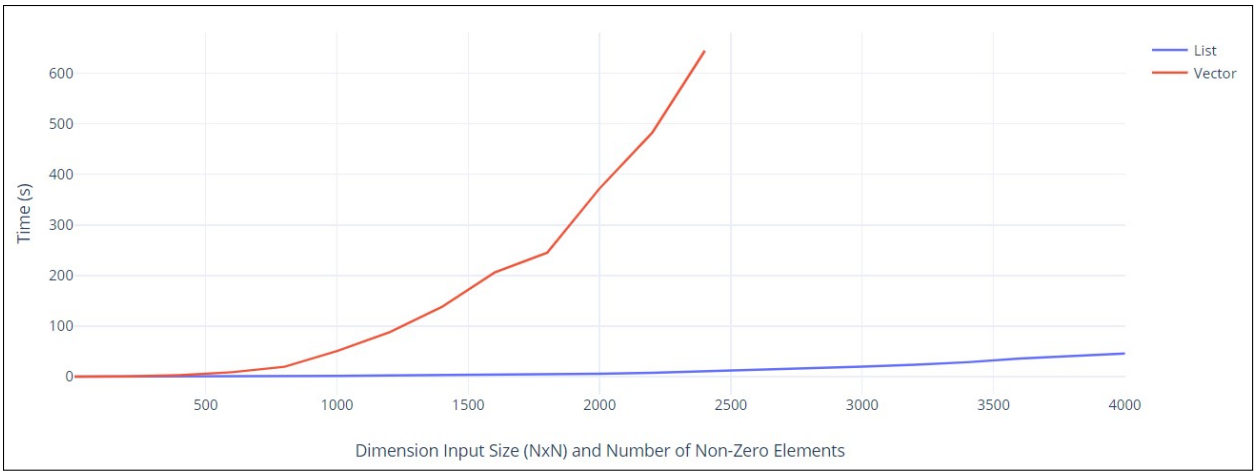

Figure 22: Time complexity plot for matrix multiplication

It is clear that the list implementation is more time-efficient, and this motivates the aim of our project to produce a data structure that can efficiently handle sparse matrices relative to the basic way with vector matrices.

The main hurdle with our testing was the run-time of the random matrix generator which slowed the process down significantly. More extensive testing was difficult because of this. However, the fact that a linked list implementation of sparse matrices can more efficiently perform operations because of more efficient memory storage is clear from our data. Possible improvements on our testing protocol include the following:

- Generate plots for method run-times as number of non-zero elements nears the number of cells
- Store randomly generated data sets to have more consistent results and make testing more efficient
- Space input sizes logarithmically

4.3 Space Complexity Analysis

The underlying reason for the superior run-time of the list matrix is its efficient memory usage specifically for sparse matrices. If our list matrix were used for dense matrices, it would negate its usefulness and be less efficient than a vector-based matrix. The reason for this is that linked lists take up more memory node-for-node than vectors do cell-for-cell. As the number of

nodes nears the number of cells in the matrix, the linked list approach is not helpful anymore.

5 Further Real-World Applications

To expand on real-world applications that sparse matrix implementations may be used for, an addendum to our report can be found on our GitHub repository.

6 Contributions and Acknowledgments

6.1 Demetrios Petrou

- List Matrix Class Structure
- Matrix Multiplication Implementation
- I/O and Random Matrix Implementation
- Vector Matrix Implementation and Comparison Testing
- Report Writing and Compilation

6.2 Justin Watkins

- Comprehensive Sparse Matrix Applications Research
- Report Writing and Compilation
- Figure Generation

6.3 Lisandro Nunez

- Presentation Slides
- MakeFile
- GitHub ReadMe

6.4 Richard Herchuk

- SFML Graphics Representations
- Matrix Addition
- Report Writing and Compilation

References

[1] *https://en.wikipedia.org/wiki/Matrix(mathematics).*
[2] *https://dziganto.github.io/Sparse-Matrices-For-Efficient-Machine-Learning/.*
[3] *https://towardsdatascience.com/why-we-use-sparse-matrices-for-recommender-systems-2ccc9ab698a4.*
[4] *https://www.sciencedirect.com/science/article/pii/S187770581503951X.*
[5] *https://www.tutorialspoint.com/data_structures_algorithms/linked_list_algorithms.htm : : text = A%20linked%20list%20is%20a, used%20data%20structure%20after%20array.*
[6] *https://github.com/cesteves/csc-212-spring22/blob/master/lectures/07-linked-lists.pdf.*
[7] *https://iq.opengenus.org/time-complexity-of-linked-list/.*
[8] *https://www.geeksforgeeks.org/sparse-matrix-representation/.*
[9] *https://en-academic.com/dic.nsf/enwiki/200849.*