SPARSE MATRICES IN DATA SCIENCE: EFFICIENT ALGORITHMS AND APPLICATIONS WITH CASE STUDY

A dissertation
Submitted in Partial Fulfilment of the Requirements for the Degree of

MASTER OF SCIENCE

in

Applied mathematics

By

Mallika Bisht
(2K22/MSCMAT/61)

Niharika Srivastava
(2K22/MSCMAT/27)

Under the supervision of
Mr. Jamkhongam Touthang
Assistant Professor, Applied Mathematics
Delhi Technological University

Department of applied mathematics

DELHI TECHNOLOGICAL UNIVERSITY
(Formerly Delhi College of Engineering)
Shahbad Daulatpur, Main Bawana Road, Delhi-110042, India

June, 2024

DELHI TECHNOLOGICAL UNIVERSITY

(Formerly Delhi College of Engineering)

Shahbad Daulatpur, Main Bawana Road, Delhi-42

CANDIDATE'S DECLARATION

We, Mallika Bisht and Niharika Srivastava of MSc. Mathematics hereby certify that the work which is being presented in the Dissertation entitled Sparse matrices in data science: efficient algorithms and applications with case study in partial fulfilment, of the requirement for the award of the degree Masters of Mathematics, submitted in the Department of Applied Mathematics, Delhi Technological University is an authentic record of my own work carried out during the period from to under the supervision of Mr. Jamkhongam Touthang.

The matter presented in the thesis has not been submitted by us for the award of any other degree of this or any other Institute.

Candidate's Signature

This is to certify that the student has incorporated all the corrections suggested by the examiners in the dissertation and the statement made by the candidate is correct to the best of our knowledge.

Signature of Supervisor **Signature of External Examiner**

DELHI TECHNOLOGICAL UNIVERSITY

(Formerly Delhi College of Engineering)

Shahbad Daulatpur, Main Bawana Road, Delhi-42

CERTIFICATE BY THE SUPERVISOR

Certified that Mallika Bisht and Niharika Srivastava has carried out their search work presented in this dissertation entitled "Sparse matrices in data science: efficient algorithms and applications with case study" for the award of Master of science in Applied Mathematics from Department of Applied Mathematics, Delhi Technological University, Delhi, under my supervision. The dissertation embodies results of original work, and studies are carried out by the students themselves and the contents of the dissertation do not form the basis for the award of any other degree to the candidate or to anybody else from this or any other University/Institution.

Signature

Mr. JamkhongamTouthang
Assistant Professor
Delhi Technological University
Shahbad Daulatpur, Main Bawana Road, Delhi-110042

Date:

SPARSE MATRICES IN DATA SCIENCE: EFFICIENT ALGORITHMS AND APPLICATIONS WITH CASE STUDY

Mallika Bisht and Niharika Srivastava

ABSTRACT

Sparse matrices in data structure are an important concept in data structure and algorithms. They provide a good way to store and manipulate large matrices; They are widely used in various fields for large matrices, like scientific computing, machine learning, and image processing. Create multiple fields. Effective algorithms for managing different matrices are important because they have the ability to reduce the budget and increase performance. This article examines a variety of algorithms and similar operations, including stored procedures (such as concatenated and concatenated rows), matrix-vector multiplication, and solutions to return-to-system problems. In addition, this article also examines the use of sparse matrices in optimization and parallel computing. This research shows a significant improvement in detail and insight using the technology matrix. The findings highlight the importance of visual differentiation of matrix algorithms in big data processing, highlighting their important role in the use of data science today. Effective algorithms for processing and incorporating data research demonstrate their implementation and quality.

ACKNOWLEDGEMENTS

We want to express our appreciation to Mr. Jamkhongam Touthang, Department of Applied Mathematics, Delhi Technological University (Formerly Delhi College of Engineering), New Delhi, for his careful and knowledgeable guidance, constructive criticism, patient hearing, and kind demean us throughout our ordeal of the present report. We will always be appreciative of his kind, helpful demean our and his insightful advice, which served as a catalyst for the effective completion of our dissertation report. We are grateful to our Mathematics department for their continuous motivation and involvement in this project work. We are also thankful to all those who, in any way, have helped us in this journey. Finally, we are thankful to the efforts of our parents and family members for supporting us with this project.

MALLIKA BISHT

NIHARIKA SRIVASTAVA

LIST OF TABLES

Table 3.1: Velocity vs. time data……………………………………………………21

LIST OF FIGURES

Fig. 1 Undirected graph……………………………………………………………30

Fig. 2 Directed graph…………………………………………………………...30

Fig. 3 An undirected graph of BFS, with the labels that shows the order in which we chose the vertices. Vertices 1, 2 and 3 are placed on the same level 1 as their distance is 1 from k. Similarly, vertices 4,5,6 and 7 are placed on second level…………….32

LIST OF SYMBOLS AND ABBREVIATIONS

Symbol	Symbol name	Symbols meaning
+	Plus	Addition
−	Minus	Subtraction
×	Multiplication or Cross or by	Multiplication or To show the order of a matrix
=	Equal sign	Equality
*	Asterisk	Multiplication
/	Division slash	Division
≤	Inequality	Less than equals to
≥	Inequality	Greater than equals to
√	Square Root	Square Root
∈	Epsilon	Belongs to
π	Pi	An irrational number
∑	Sigma	Summation
→	Right arrow	Directed graph
↔	Left-Right arrow	Undirected graph

Abbreviations	Full form
COO	Coordinate list
CSR	Compressed Sparse Row
CSC	Compressed Sparse Column

CONTENTS

Title	**Page No.**
Certificate	**ii**
Abstract	**iii**
Acknowledgements	**iv**
List of Tables	**v**
List of Figures	**v**
List of Symbols and Abbreviations	**v**
References	**viii**
Plagiarism Verification	**ix-x**

CHAPTER 1: INTRODUCTION 1-5

1 Introduction
1.1 Definition
1.2 Use of Sparse matrix
1.2.1 Sparse Matrices in Linear Algebra
1.2.2 Sparse Matrices in Data Learning
1.3 History of Sparse Matrices
1.3.1 Contributions of Mathematicians
1.4 Motivation
1.4.1. Advantages of Sparse Matrix
1.5. Summary

CHAPTER 2: MATHEMATICAL FOUNDATIONS AND REPRESENTATIONS 6-14

2.1 Mathematical Foundations
2.1.1. Basic Concepts
2.1.2. Types of sparse matrix
2.1.3. Theoretical Properties of Sparse Matrices
2.2. Representation of sparse matrix
2.2.1. Coordinate list representation
2.2.2. Linked list representation
2.2.3. Compressed Sparse Row (CSR)
2.2.4. Compressed Sparse Column (CSC)
2.3 Comparison
2.4 Summary

CHAPTER 3: ALGORITHMS FOR SPARSE MATRIX 15-27

3.1 Basic Operations
3.1.1 Addition
3.1.2 Subtraction
3.1.3 Multiplication
3.1.4 Transpose

3.2 Permutation and reordering
3.3 Solving sparse linear system
3.3.1 Direct Method
3.3.2 Iterative Method
3.4 Matrix Factorization
3.4.1 Cholesky Factorization
3.4.2 QR Factorization

CHAPTER 4: APPLICATIONS OF SPARSE MATRICES 28-32
4.1. Application of Sparse matrices in scientific computing
4.2. Application of Sparse matrices in PDEs
4.2.1. Finite Element Analysis (FEA)
4.2.2. Computational Fluid Dynamics (CFD)
4.3. Applications of sparse matrices in Data Science and Machine learning
4.3.1. Natural Language Processing
4.3.2. Recommendation Systems
4.3.3. Market Basket Analysis
4.4 Basics of Graphs
4.4.1 Definition
4.4.2 Basic terms
4.4.3 Adjacency Graphs
4.4.4 Graph searches
4.4.5 Breadth-First search
4.4.6 Depth-First search

CHAPTER 5: OPTIMIZATIONS, PARALLEL COMPUTING, CASE STUDY AND FUTURE DIRECTIONS 33-40
5.1. Large-Scale Optimization
5.1.1. Sparse Matrix Factorization
5.1.2. Conjugate Gradient and Krylov Subspace Method
5.1.3. Application Example
5.2. Parallel Computing Using Sparse Matrices
5.2.1. Parallel Sparse Matrix Operation
5.2.2. Application Example
5.3. Case study
5.3.1. Social Network Graph
5.3.2. Results
5.4. Challenges and Future Directions
5.4.1. Challenges
5.4.2. Future Directions

CHAPTER 6: CONCLUSION 41-42

REFERENCES

CHAPTER 1

INTRODUCTION

Sparse matrices are an important concept in linear algebra and data science and play an important role in managing and processing large data sets. “A sparse matrix is a matrix in which most of its elements are zero.” This is in contrast to the velocity method, where most elements are non-zero. Many of the zero elements in sparse matrices allow the use of special storage and computation techniques to save memory and processing time, making them especially important when working with large data sets.

1.1. Definition

“Sparse matrix is a matrix with the majority of its elements equal to zero. However, there is no fixed ratio of zeros to non-zero elements.” In a sparse matrix, the presence of zero values compared to the presence of non-zero elements give some room for information to be represented and stored, with less recent memory used.

Example 1.1: Let us take a matrix $A = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 6 & 0 & 0 & 0 \\ 0 & 5 & 0 & 0 & 8 \\ 0 & 0 & 4 & 0 & 0 \end{bmatrix}$. Let's look at the content of this matrix.

Solution: Here we can see that there are only 6 elements which are non-zero, the rest are zero. It is an example of **Sparse Matrix.**

1.2. Use of Sparse matrix

Sparse matrices are used to reduce matrix filling.

Fill-ins: The entries of the matrix are the elements that change from zero to non-zero when we apply the algorithm.
For reducing memory requirements and the arithmetic operations used during processing, it’s necessary to reduce padding by changing the rows and rows in the matrix.

1.2.1. Sparse Matrices in Linear Algebra

In linear algebra, matrices are used to show and solve systems of linear equations, perform transformations, and model many physical and computational phenomena. Sparse matrices appear in many applications:

- **Graph Theory:** An adjacency matrix represents a large graph where most nodes are not directly connected.
- **Numerical Solutions of Partial Differential Equations (PDEs):** A separate part of the equation often results in different systems due to local interactions.
- **Optimization Problems:** Many large-scale optimization problems give sparse constraint matrices.

Manipulating matrices, including using their formulas, is less useful. Representations such as Compressed Sparse Row (CSR) or Compressed Sparse Column (CSC) models, as well as specific techniques for matrix operations such as sparse matrix-vector multiplication, have been developed using sparsity. This technique reduces computational complexity and memory usage. We will introduce this notation in the following section.

1.2.2. Sparse Matrices in Data Learning

Sparse matrices are also important in data learning, especially machine learning and statistics. There are several situations that normally cause data to be underrepresented:

- **Text Mining and Natural Language Processing (NLP):** Term document matrices (in which each document is represented by a term frequency vector) are generally rare because one of the documents contains only a small fraction of the total content.
- **Recommender Systems:** The user interaction matrix is sparse because the user interacts with only a few functions on a large laptop.
- **Feature Selection in Machine Learning:** Many features in high-dimensional data are irrelevant or repeated, often resulting in little difference after applying dimensionality reduction techniques.

Using sparse matrices in data learning allows algorithms to scale well. Regular methods like lasso (minimum shrinkage and operator selection) use variables to improve model interpretation and performance. Libraries like SciPy in Python provide strong support for small matrix operations, improving the performance of complex processes on large datasets.

1.3. History of Sparse Matrices

Sparse matrices have been an important concept since their inception, but their importance increased in the mid-20th century with the development of science and engineering. Thanks to the continuous efforts of scientists and advances in computer science and technology, significant advances have been made in recent years. This success has led to many new benefits in the field. Some of them are as follows:

- 1950s-1960s: The first developments in matrix technology can rarely be traced back to this period and the advent of digital computers. Scientists began looking

for ways to effectively represent and control matrices containing many zeros, often found in scientific and engineering calculations.

- 1970s-1980s: During this period, significant growth was made in the development of algorithms, representations and data structures for sparse matrices due to the increasing need for effective numerical methods in many fields such as finite element analysis, optimization, and scientific computing. Researchers such as Alphonse Buja and Iain Duff carried out important work in this field during this period.
- 1990s to present: Advances in parallel and distributed computing models further stimulate research on differential matrix algorithms and applications. Efforts have been made to develop similar methods for sparse matrix operations, and the emergence of libraries such as PETSc (Portable and Extensible Scientific Computing Toolkit) and Trilinos have provided solutions to serious problems in solving inequality problems and eigenvalue problems.

1.3.1. Contributions of Mathematicians

Some important contributions to the development and use of sparse matrices include:

- Alphonse Buja: Published one of the first works on sparse matrices in 1959; this introduced a method for representing and managing sparse matrices in computing.
- Hans Schneider and Arnold Neumaier: In the late 1970s and early 1980s, they made important contributions to rarefied algebra, especially eigenvalue calculations.
- Yousef Saad: His work on iterative methods for linear nonlinearities (such as the gradient method) had an impact on the field of linear algebra.
- James W. Demmel and Jack Dongarra: For their significant contributions to the development of algorithms of sparse matrix and their use in parallel and distributed computing.
- Gilbert Strang: His work on sparse matrix techniques, especially in the context of finite element methods, was very influential.

1.4. Motivation

Sparse matrix has become an important area of research due to its unique properties and implications for many computational operations. The main motivation for studying sparse matrices is as follows.

1. **Memory Efficiency**: Sparse matrices require more memory storage space due to zero element majority. This is crucial for solving large problems that would be bad for thick matrices.
2. **Computational Speed**: The operation of sparse matrices can be optimized to skip zero elements, thus reducing computation time. This performance is important for applications that require immediate or short-term processing.
3. **Scalability**: The reduced memory and computational requirements of sparse matrices make it possible to solve very large data and complex problems that are required for big data today.

4. **Algorithm Optimization**: Special algorithms for sparse matrices, such as sparse matrix-vector multiplication and sparse solvers, can improve performance, making these algorithms suitable for commercial success.
5. **Relevance to Real-World Data**: Most forms of data in the world are always different matrices. For example, in networks, consensus, and computational sciences, the outcome varies due to the nature of the data.
6. **Energy Efficiency**: Reduced computing and memory requirements directly translate into lower power consumption; This is important for extending battery life in business computing and mobile devices and equipment.
7. **Enabling New Technologies**: Sparse matrices form the basis for the development of new technologies in areas such as machine learning, optimization and signal processing. Their research has led to new innovations that can efficiently process large and complex data.
8. **Improving Accuracy and Precision**: Technique's of Sparse matrix can help to reduce the number of errors in calculations by focusing resources on important points, thus increasing the accuracy and precision of results.

1.4.1. Advantages of Sparse Matrix

1. Memory operation:

Sparse matrices store only the elements which are non-zero and their values. It should be less remembered than the density matrix, especially for large matrices with low density. Since files are in both large and small formats, storing them in a smaller format will save a lot of memory and allow larger files to be processed within available limits. It becomes most important in the finite process, finite difference method or notation.

2. Faster calculations:

Faster calculations are possible. This is especially useful for operations such as matrix-vector multiplication and the solving systems. Methods for solving problems, such as the gradient network method or GMRES, generally converge faster when applied to matrices sparser than 1, thus saving all effort.

Now the question arises: What is the property of this sparse matrix? Why do we need them? How do they differ from dense matrices? What if we never discovered them? There are many. Before summarizing the use of sparse matrices in various fields. Let us see one example which will tell us the basic difference between the sparse matrix and dense matrix.

Example 1.2: Take the same matrix A = $\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 6 & 0 & 0 & 0 \\ 0 & 5 & 0 & 0 & 8 \\ 0 & 0 & 4 & 0 & 0 \end{bmatrix}$. Let's see the storage of this matrix in each case of dense matrix and sparse matrix.

Solution: The first case where we used a Dense matrix:

In this case, all details will be kept open. Here we see that the entries of the matrix are integers that store 4 bytes.

Total memory for dense matrix $= Number\ of\ elements \times Size\ of\ each\ element$

$= 25 \times 4\ bytes$

$= 100\ bytes$

Taking the first case of given Sparse matrix:

Here, the non-zero elements and their indices are:

- (0,0): 1
- (1,2): 1
- (2,1): 6
- (3,1): 5
- (3,4): 8
- (4,2): 4

Total memory for sparse matrix $= Number\ of\ non-zero\ elements \times (\text{Size of row index} + \text{Size of column index} + \text{Size of element value})$

$= 6 \times (4 + 4 + 4)\ bytes$

$= 72\ bytes$

This illustrates an important aspect of using a sparse matrix; difference matrix representation requires less memory than dense representation.

Note: If you use a small matrix, the sparse matrix will use more memory compared to the dense matrix; therefore, sparse matrices are useful for large matrices.

1.5. Summary

In Chapter 1, we introduced the definition of sparse matrix and highlighted its importance in various fields such as mathematics, engineering, and information science. We also discussed the motivations behind examining sparse matrices for their performance and applicability in solving real-world problems. Additionally, we provided background information on the development of sparse matrix theory and algorithms, acknowledging the seminal work that underpins current research and applications. Strong mathematical foundation. In the next section, we will examine the mathematical concepts and properties of different matrices, including various notations for storage and efficiency. This understanding will provide a solid foundation for the algorithms and applications discussed in the next section.

CHAPTER 2

MATHEMATICAL FOUNDATIONS AND REPRESENTATIONS

2.1. Mathematical Foundations

This section discusses the basic mathematical concepts and properties that support the study of sparse matrices. By establishing a solid foundation, we can better understand the algorithms and applications discussed in the next section.

2.1.1. Basic Concepts

- **Sparsity and Density**: Sparsity is the proportion of zero elements to all elements in the matrix. Instead, density is proportion of elements that are non-zero. The concept of sparsity is important because it determines the specific storage and computation methods required to handle sparse matrices efficiently.

- **Sparsity Patterns**: It describes the arrangement of non-zero elements in the matrix. This model can influence the choice of storage methods and algorithms. Examples include diagonal matrices, banded matrices, and block sparse matrices.

2.1.2. Types of sparse matrix

Matrices do not necessarily have to have the same form. They can be mainly categorized into two groups:

a) Regular Sparse Matrices
b) Irregular Sparse Matrices

Let us understand it one by one.

a) Regular Sparse Matrices

A sparse matrix is said to be regular if there is a pattern or pattern among its elements or if there is some degree of regularity throughout the matrix.

We can further divide regular matrix as follows:

i. Diagonal Matrix:

The non-zero elements are on the main diagonal.

Example: A = $\begin{bmatrix} d_1 & 0 & 0 \\ 0 & d_2 & 0 \\ 0 & 0 & d_3 \end{bmatrix}$

ii. Banded Matrix:

"Non-zero elements are restricted to the diagonal line containing the main diagonal and the possibility of additional diagonals on either side."

Example: A = $\begin{bmatrix} a & b & c & 0 & 0 \\ d & e & f & g & 0 \\ h & i & j & k & l \\ 0 & m & n & o & p \\ 0 & 0 & q & r & s \end{bmatrix}$

Here the name suggests, the elements form a band around the diagonal. Below is a special case of banded matrix.

a) Tridiagonal Matrix:

"Here the non-zero elements are constricted to the main diagonal, the diagonal above and below it."

Example: B = $\begin{bmatrix} a & b & 0 & 0 & 0 \\ c & d & e & 0 & 0 \\ 0 & f & g & h & 0 \\ 0 & 0 & i & j & k \\ 0 & 0 & 0 & l & m \end{bmatrix}$

iii. Block Sparse Matrix:

When a matrix is divided into smaller submatrices, some of which are sparse or zero matrices.

Example: A = $\begin{bmatrix} A & 0 & 0 \\ 0 & B & 0 \\ 0 & 0 & C \end{bmatrix}$

Here A, B and C are sparse matrices or sometimes dense.

iv. Sparse Symmetric Matrix:

A Sparse Matrix is known as symmetric if $A_{ij} = A_{ji}$. The non-zero elements are symmetrical across the main diagonal.

Example: A = $\begin{bmatrix} a & d & 0 \\ d & b & e \\ 0 & e & c \end{bmatrix}$

v. Lower and Upper Triangular Matrix:

"All the non-zero elements are on the main and below the main diagonal then that matrix is known as Lower Triangular Matrix."

Example: C = $\begin{bmatrix} a & 0 & 0 \\ d & b & 0 \\ f & e & c \end{bmatrix}$

"All the non-zero elements are on the main and above the main diagonal then that matrix is known as Upper Triangular Matrix."

Example: B = $\begin{bmatrix} a & d & f \\ 0 & b & e \\ 0 & 0 & c \end{bmatrix}$

b) Irregular Sparse Matrices

A matrix with no particular structure or pattern among its elements is known as Irregular matrix. There are no such types of this matrix, as classifying any matrix requires a pattern, which is not found in this case.

2.1.3. Theoretical Properties of Sparse Matrices

Sparse matrices have many theoretical properties that are important for understanding their behaviour and designing effective algorithms to manipulate them. Here we discuss their rank, determinant and eigenvalues.

a) Rank

"The rank of a matrix is the maximum number of linearly independent rows or columns."

Properties:

- **Linearly Independent Rows/Columns**: The row of a matrix represents the size of the vector space spanned by its rows or columns. The distribution of nonzero elements in a sparse matrix determines the linear independence of rows and columns.
- **Efficient Computation**: Sparse matrices allow decision algorithms to be more efficient, thus avoiding unnecessary operations on zero elements. Techniques such as sparse LU decomposition and iterative problem solving make use of different models.
- **Impact on Solutions**: Level is important in determining the solution of the system Ax =b. If the system is uniform, the fully ranked matrix (if rank is equal to the smallest of the matrix) means that it's a single solution.

b) Determinant

Properties:

- **Invertibility**: A non-zero determinant means that the matrix is invertible. For sparse matrices, it is often impractical to calculate the determinant directly due to computational complexity.
- **Determinant Calculation**: Use special methods such as matrix factorization (LU factorization) to calculate the rank of the difference matrix. This strategy aims to preserve sparsity and avoid padding (generating non-zeroes in the first place).
- **Sparse-Specific Methods**: In some cases, combining techniques or the use of non-uniform matrix models (e.g. block matrices, banded matrices) can be used to simplify the decision. For example, "the determinant of a diagonal or triangular matrix is the product of the diagonals it contains."

c) Eigenvalues

Properties:

- **Spectral Properties**: Eigenvalues provide information about the stability and function of the system represented by the matrix. In a sparse matrix, pattern sparsity affects the distribution of eigenvalues.
- **Computation**: Good algorithms for calculating the eigenvalues of sparse matrices include the Lanczos algorithm and Arnoldi iteration. This technique focuses on a few eigenvalues and eigenvectors, which is usually sufficient for many applications.

2.2. Representation of sparse matrix

In most real-world problems, data always forms sparse matrices. For example, in graph theory, the adjacency matrix has numerous zeros for a large sparse graph, indicating that there are no edges between nodes. Likewise, matrices representing objects in simulations such as finite element analysis often have a structure that causes sparsity. Efficient representation and control of these sparse matrices are important for the efficiency and development of computational algorithms.

Sparse matrices can be represented efficiently using special formulas to save memory and computational resources. Below are three representations of sparse matrices:

2.2.1. Coordinate list representation (COO)

To represent the sparse matrix, a 2-dimensional array is used with three rows written as follows:

- Row: The row index of the non-zero element.
- Column: The column index of the non-zero element.
- Value: The value of the non-zero element located at the corresponding index.

Example 2.1: Taking the same matrix A = $\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 6 & 0 & 0 & 0 \\ 0 & 5 & 0 & 0 & 8 \\ 0 & 0 & 4 & 0 & 0 \end{bmatrix}$.
Represent this matrix using a Coordinate list representation.

Solution: For the given matrix, the non-zero elements are 1, 1, 6, 5, 8, and 4. We will store only these elements, organizing them into 3 rows: Row, Column, and Value. Below is the tabular representation as an array.

ROW	0	1	2	3	3	4
COLUMN	0	2	1	1	4	2
VALUE	1	1	6	5	8	4

2.2.2. Linked list representation

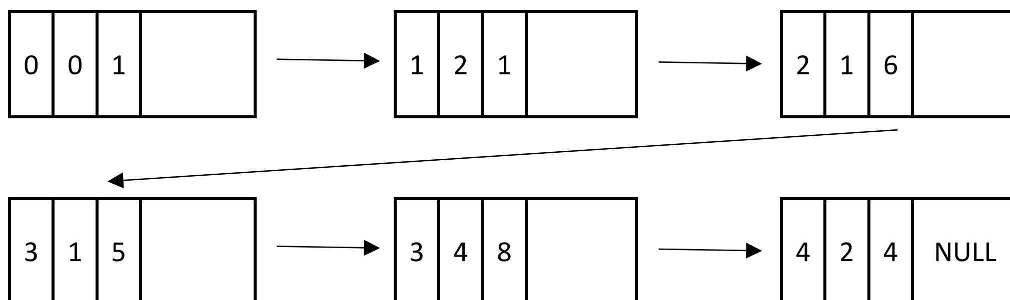
To represent a sparse matrix using a linked list, you need to define nodes that will store only elements (non-zero) and their row and column indices. Using linked names we can represent a matrix mathematically as follows:

- Row: The row index of the non-zero element.
- Column: The column index of the non-zero element.
- Value: The value of the non-zero element located at the corresponding index.
- Next node: Told the location of the next node.

NODE STRUCTURE	ROW	COLUMN	VALUE	ADDRESS OF THE NEXT NODE

Example 2.2: Taking the same matrix A = $\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 6 & 0 & 0 & 0 \\ 0 & 5 & 0 & 0 & 8 \\ 0 & 0 & 4 & 0 & 0 \end{bmatrix}$.
Represent this matrix using linked list representation.

Solution: As mentioned before, non-zero elements are 1, 1, 6, 5, 8, and 4. We start by creating the first node using the first non-zero element, then move on to the next node, and so on.



Here, we can see that in row 3 there are two elements in the same row which means we are storing one more element in the row in the above two representations. Do we have such representation through which we can save the space of this extra entry? Indeed, the answer is yes. Another representation is used to avoid duplication of row indices.

2.2.3. Compressed Sparse Row (CSR)

As the name suggests, this representation stores a sparse matrix by compressing the row data. We can represent it in the below 3 rows named as follows:

- Value: The value of the non-zero element located at the corresponding index.
- Column: The column index of the non-zero element.
- Row Pointers: It shows from where each row begins in the column and value indices arrays.
  Note: If we have an $n \times m$ matrix, the size of the row pointers or the length is $n + 1$.

Example 2.3: Consider the same matrix $A = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 6 & 0 & 0 & 0 \\ 0 & 5 & 0 & 0 & 8 \\ 0 & 0 & 4 & 0 & 0 \end{bmatrix}$.
Represent this matrix using Compressed Sparse Row representation.

Solution: We start with the 3 rows as mentioned above. The value and column indices will remain the same so move to the row pointer row.
The first-row pointer is shown as:

- Row Pointers: [0]

First row begins at the index ‘0’ so the first element in row pointer is ‘0’.

- Row Pointers: [0, 1]

Second row begins at the index ‘1’ so the second element in row pointer is ‘1’.

- Row Pointers: [0, 1, 2]

Third row begins at the index ‘2’ so the third element in row pointer is ‘2’.

- Row Pointers: [0, 1, 2, 3]

The fourth row starts at the index ‘3’ so the fourth element in the row pointer is ‘3’.

- Row Pointers: [0, 1, 2, 3, 5]

The fifth row starts at the index ‘5’ so the fifth element in the row pointer is ‘5’. In other words, the element ‘8’ is in the same row with element ‘5’ so we will skip index ‘4’ and go to index ‘5’.

- Row Pointers: [0, 1, 2, 3, 5, 6]

The last element ‘6’ indicates the end of the last row also it shows the total number of non-zero elements.

The final representation is as follows:

- Value: [1, 1, 6, 5, 8, 4]
- Column: [0, 2, 1, 1, 4, 2]
- Row Pointers: [0, 1, 2, 3, 5, 6]

This example illustrates a single row containing two non-zero elements, highlighting the usefulness of CSR representation, particularly in scenarios where such multiple rows are there.

Similarly, as CSR we have Compressed Sparse Column (CSC) representation which compresses the column data.

2.2.4. Compressed Sparse Column (CSC)

We show the matrix through 3 rows named as follows:

- Value: The value of the non-zero element located at the corresponding index.
- Row: The row index of the non-zero element.
- Column Pointers: It shows from where each column begins in the row and value indices arrays.
  Note: If we have an n × m matrix, the size of the row pointers or the length is n + 1.
  Example 2.4: Let us take the same example 5.

Solution: The value and row indices will remain the same so move to the row pointer row.

- The first column, there is a non-zero element at index 0 in values array.
- Second column, there are non-zero elements at indices 2 and 3 in the values array.
- The third column, there are non-zero elements at indices 1 and 4 in the values array.
- The fourth column, there is no non-zero element. Hence, we will use the index ‘5’.
- The fifth column, there is a non-zero element at index 4 in values array.

The final representation is as follows:

- Value: [1, 6, 5, 1, 4, 8]
- Row: [0, 2, 3, 1, 4, 3]
- Row Pointers: [0, 1, 3, 5, 5, 6]

There are various other representations such as Diagonal, Block compressed sparse Row, Ellpack – Itpack, Jagged Diagonal storage. However, we have discussed the one which is used most commonly. In practice, the choice of variable matrix representation depends on the particular attributes of the problem and the operations to be performed. So, chose wisely as per the scenario.

2.3. Comparison

The below table provides a quick comparison of four common sparse matrix representations, highlighting their structural properties, memory usage, construction complexity, efficiency in row and column access, ease of insertion and deletion, and ease of conversion to other formats.

Feature	**Coordinate list (COO)**	**Linked list**	**Compressed Sparse Row (CSR)**	**Compressed Sparse Column (CSC)**
Structure	List of (row, col, value) tuples	Nodes with pointers	values, col indices, row pointer	values, row indices, col pointer
Memory Usage	Three arrays	High due to pointers	Efficient	Efficient
Construction	Simple	Flexible	Complex	Complex
Row Access	Moderate	Efficient	Fast	Moderate
Column Access	Moderate	Efficient	Moderate	Fast
Insertion/Deletion	Easy	Easy	Complex	Complex
Conversion	Easy to convert	Harder to convert	Specific to row operations	Specific to column operations

2.4. Summary

A solid foundation is laid by introducing the basic concepts and theoretical properties of sparse matrices. We explored different types of sparse matrices and examined their special properties, such as rank, determinant, and eigenvalues. Additionally, we examined different representations such as Coordinate list (COO), Linked List,

Compressed Sparse Row (CSR), and Compressed Sparse Column (CSC). Also compared their performance and suitability for different tasks. With a better understanding of these important concepts, we are now ready to move on to Chapter 3, where we will explore algorithms for optimizing the performance of different matrices, starting with the simple operations which shape the foundation of more complex computational methods.

Chapter 3

ALGORITHMS FOR SPARSE MATRIX

3.1 Basic Operations

Simple operations are ubiquitous when working with matrices and will be used in the next section of this paper. Sparse matrices are easier to work with than other matrices because most of the elements are zero. Here we will discuss adding, subtracting, multiplying matrices and finding the transformation of a matrix.

Example 3.1: Let us take A = $\begin{bmatrix} 1 & 0 & 0 & 2 \\ 0 & 3 & 0 & 0 \\ 0 & 0 & 4 & 0 \\ 5 & 0 & 0 & 6 \end{bmatrix}$ and B = $\begin{bmatrix} 0 & 7 & 0 & 0 \\ 8 & 0 & 0 & 0 \\ 0 & 0 & 0 & 9 \\ 0 & 0 & 10 & 0 \end{bmatrix}$. Apply operations like addition, subtraction, multiplication of matrices and also find Transpose of a matrix.

Solution: Representing the given matrices in Coordinate representation (COO). I chose this representation, but the reader is free to use his or her preference.

COO representation of matrix A is as follows.

ROW	0	0	1	2	3	3
COLUMN	0	3	1	2	0	3
VALUE	1	2	3	4	5	6

3.1.1 Addition

COO representation of matrix B is as follows.

ROW	0	1	2	3
COLUMN	1	0	3	2
VALUE	7	8	9	10

Addition of A and B matrices: We will add only those entries whose corresponding row and value indices are same. For rest of the row and column indices, we will just note it down as it is. COO representation of (A + B) is as follows.

ROW	0	0	0	1	1	2	2	3	3	3
COLUMN	0	1	3	0	1	2	3	0	2	3
VALUE	1	7	2	8	3	4	9	5	10	6

3.1.2 Subtraction

Similarly, COO representation of (A - B) is as follows.

ROW	0	0	0	1	1	2	2	3	3	3
COLUMN	0	1	3	0	1	2	3	0	2	3
VALUE	1	-7	2	-8	3	4	-9	5	-10	6

3.1.3 Multiplication

COO representation of (A * B) is as follows.

Here, for each non-zero element of A (i,k) get an element of B (k,j) such that column index k of A matches the row index k of B. Calculate the product:

A (i,k) * B (k,j) and then add it result matrix as C (i,j).Representing the multiplication of (A * B).

ROW	0	0	1	2	3
COLUMN	1	2	0	3	1
VALUE	7	20	24	36	35

3.1.4 Transpose

For transposition of matrix, we will simply swap the row indices with column indices. COO representation of (A^T) is as follows.

ROW	0	3	1	2	0	3
COLUMN	0	0	1	2	3	3
VALUE	1	5	3	4	2	6

It's similar to what we do with simple matrices, but it's very important when we deal with large matrices as here, we are only dealing with non-zero matrices.

3.2 Permutations and Reordering: Permuting the rows or columns (or rows and columns) of a sparse matrix is a common task. In fact, reordering rows and columns is one of the most important components used in parallel implementation of direct and iterative solution methods.

Let A be a matrix and $\pi = \{i_1, i_2, \ldots, i_n\}$ a permutation of the set $\{1, 2, \ldots, n\}$. Then the matrices:

$A_{\pi,*} = \{a_{\pi(i),j}\}$ i = 1,...,n; j = 1,...,m,

$A_{*,\pi} = \{a_{i,\pi(j)}\}$ i = 1,...,n; j = 1,...,m

are called row π-permutation and column π-permutation of A respectively.

The set {1, 2, . . ., n} is obtained as a result of n or fewer permutations, that is, the fundamental permutations in which only two entries are exchanged. The exchanged matrix is the identity matrix in which two rows are exchanged. Let us denote these matrices as Xij. where i and j are the number of rearranged rows.

b) Permutation basically rearranges the rows and/or columns of a matrix.
c) **Purpose**: To change the sparsity pattern, often to improve computational efficiency.
d) **Mathematical Representation**: If PP and QQ are permutation matrices, the permuted matrix A' is given by A′=PAQ.

P permutes the rows of A.

Q permutes the columns of A

Sparse matrix reordering is an optimization technique used to improve the efficiency of operations on sparse matrices by rearranging their rows and columns. Matrix reordering has a variety of uses.

Changing the order of the rows and columns of a sparse matrix can affect the speed and memory requirements of matrix operations.

By rearranging the rows and columns of a matrix, we can reduce the number of fill-ins produced by factorization, thus reducing the time and storage cost of subsequent calculations.

3.3 Solving sparse linear system: Solving sparse matrix is basically about solving a system of equations given as:

Ax=b

Where x and b belong to Rn and A is our nxn sparse matrix.

There are two methods to solve the sparse linear equations first is direct method and the second is iterative method.

For iterative algorithm, there are some famous methods as Gauss-seidel and Jacobi method and etc. For direct algorithm the famous methods are Gaussian elimination, LU decomposition, etc.

3.3.1 Direct Method: Direct methods solve a system by performing a finite series of operations to transform a matrix into a simpler form from which the solution can be derived directly.

Gaussian Elimination Method: Gaussian elimination is a direct method for solving systems of linear equations. This involves converting the system to upper triangular matrix form, where the solution can be easily found by back substitution. The Gaussian elimination method is simple and effective for dense matrices.

Steps involve in above method,

- **Forward Elimination**: Convert matrix A to an upper triangular matrix U using a series of row operations.
- **Back Substitution**: Solve the triangular system $Ux=b$ to find the solution vector x.
- **Sparse Gaussian Elimination:** This approach extends Gaussian elimination to sparse matrices, carefully managing fill-in during the elimination process. Pivoting strategies and data structures optimized for sparse matrices are used to minimize fill-in.

Example 3.2: Solve the given system by Gaussian elimination.

$$2x+3y=6$$

$$x-y=1/2$$

Solution: First, we write this as an augmented matrix.

$$\left[\begin{array}{cc|c} 2 & 3 & 6 \\ 1 & -1 & 1/2 \end{array}\right]$$

$$R1 \leftrightarrow R2 \rightarrow \left[\begin{array}{cc|c} 1 & -1 & 1/2 \\ 2 & 3 & 6 \end{array}\right]$$

$$-2R1+R2 = R2 \rightarrow \left[\begin{array}{cc|c} 1 & -1 & 1/2 \\ 0 & 5 & 5 \end{array}\right]$$

multiply row 2 by 1/5.

$$1/5\ R2 = R2 \rightarrow \left[\begin{array}{cc|c} 1 & -1 & 1/2 \\ 0 & 1 & 1 \end{array}\right]$$

Using back-substitution, the second row of the matrix represents y=1.

Back-substitute y=1 into the first equation.

$$x-1 = 1/2$$

$$x=3/2$$

The solution is the point (3/2,1).

LU Decomposition Method: For sparse matrices, LU decomposition aims to preserve sparsity while factorizing the matrix. Several methods have been developed to achieve this, including:

Sparse LU Factorization Algorithms: Specialized algorithms, such as the multifrontal method and nested dissection, exploit the sparsity pattern of the matrix to perform LU decomposition efficiently. These algorithms typically divide the matrix into smaller blocks and factorize them independently, reducing the computational complexity.

For a non-singular matrix [A] one can always write it as

$$[A]=[L][U]$$

Where

[L]=Lower triangular matrix

[U]=Upper triangular matrix

Then if one is solving a set of equations

$$[A][X]=[C]$$

then

$$[L][U][X]=[C] \text{ as } ([A]=[L][U])$$

Multiplying both sides by $[L]^{-1}$,

$$[L]^{-1}[L][U][X]=[L]^{-1}[C]$$

$$[I][U][X] = [L]^{-1}[C] \text{ as } ([L]^{-1}[L] = [I])$$

$$[U][X]=[L]^{-1}[C] \text{ as } ([I]\,[U]=[U])$$

Let

$$[L]^{-1}[C]=[Z]$$

then

$$[L][Z]=[C] \quad (1)$$

and

$$[U][X]=[Z] \quad (2)$$

So we can solve Equation (1) first for [Z] by using forward substitution and then use Equation (2) to calculate the solution vector [X] by back substitution.

Example 3.3: Find the LU decomposition of the matrix

$$[A] = \begin{bmatrix} 25 & 5 & 1 \\ 64 & 8 & 1 \\ 144 & 12 & 1 \end{bmatrix}$$

Solution: [A]=[L][U]

$$= \begin{bmatrix} 1 & 0 & 0 \\ l21 & 1 & 0 \\ l31 & l32 & 1 \end{bmatrix} \begin{bmatrix} u11 & u12 & u13 \\ 0 & u22 & u23 \\ 0 & 0 & u33 \end{bmatrix}$$

The [U] matrix is the same as found at the end of the forward elimination of Naïve Gauss elimination method, that is

$$[U] = \begin{bmatrix} 25 & 5 & 1 \\ 0 & -4.8 & -1.56 \\ 0 & 0 & 0.7 \end{bmatrix}$$

To find l_{21} and l_{31}, find the multiplier that was used to make the a_{21} and a_{31} elements zero in the first step of forward elimination of the Naïve Gauss elimination method. It was

$$l_{21} = 64/25$$

$$= 2.56$$

$$l_{31} = 144/25$$

$$= 5.76$$

To find l_{32},

$$\begin{bmatrix} 25 & 5 & 1 \\ 0 & -4.8 & -1.56 \\ 0 & -16.8 & -4.76 \end{bmatrix}$$

So

$$l_{32} = -16.8/-4.8$$

$$= 3.5$$

Hence,

$$[L] = \begin{bmatrix} 1 & 0 & 0 \\ 2.56 & 1 & 0 \\ 5.76 & 3.5 & 1 \end{bmatrix}$$

Confirm $[L][U]=[A]$.

$$[L][U] = \begin{bmatrix} 1 & 0 & 0 \\ 2.56 & 1 & 0 \\ 5.76 & 3.5 & 1 \end{bmatrix} \begin{bmatrix} 25 & 5 & 1 \\ 0 & -4.8 & -1.56 \\ 0 & 0 & 0.7 \end{bmatrix}$$

$$= \begin{bmatrix} 25 & 5 & 1 \\ 64 & 8 & 1 \\ 144 & 12 & 1 \end{bmatrix}$$

3.3.2 Iterative Methods: Iterative methods solve a system by starting with an initial guess and iteratively improving it to get closer to the solution.

Gauss-Siedel Method: The Gauss-Seidel method is an iterative method used to solve systems of linear equations and is particularly useful for sparse matrices. This method is an improvement over the Jacobi method and can be more efficient in many cases. Here is a detailed explanation of the Gauss-Seidel method, its application to sparse matrices, and its advantages and disadvantages.

Gauss-Seidel Method Overview:

The Gauss-Seidel method iteratively updates the solution of the system of linear equations $Ax=b$ by using the latest available values for each variable. The algorithm can be summarized as follows:

Initialization: Starting with an initial guess for the solution vector x(0).

Iterative Update: For each iteration k:

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij} x_j^{(k)} \right), \quad \text{for } i = 1, 2, \ldots, n$$

Here, a_{ij} are the elements of the matrix A, $x_{i(k+1)}$ is the updated value of the i-th component, and $x_{j(k)}$ are the values from the previous iteration.

Example 3.4: The upward velocity of a rocket is given at three different times in the following table

Table 3.1: Velocity vs. time data.

Time,t(s)	Velocity, v(m/s)
5	106.8
8	177.2
12	279.2

The velocity data is approximated by a polynomial as

$v(t) = a_1t^2+a_2t+a_3,\ 5 \le t \le 12$

Find the values of a1, a2, and a3 using the Gauss-Seidel method. Assume an initial guess of the solution as

[a1, a2, a3] = [1, 2, 5]

and conduct two iterations.

Solution: The polynomial is going through three data points (t1,v1), (t2,v2) and where from the above table

$$t1 = 5,\ v1 = 106.8$$

$$t2 = 8,\ v2 = 177.2$$

$$t3 = 12,\ v3 = 279.2$$

Requiring that $v(t)=a_1t^2+a_2t+a_3$ passes through the three data points gives

$$v(t1) = v1 = a_1t^2_1+a_2t_1+a_3$$

$$v(t2) = v2 = a_1t_2^2+a_2t_2+a_3$$

$$v(t3) = v3 = a_1t^2_3+a_2t_3+a_3$$

Substituting the data (t1,v1),(t2,v2),and(t3,v3) gives

$$a_1(5^2)+a_2(5)+a_3 = 106.8$$

$$a_1(8^2)+a2(8)+a_3 = 177.2$$

$$a_1(12^2)+a_2(12)+a_3 = 279.2$$

or

$$25a_1+5a_2+a_3=106.8$$

$$64a_1+8a_2+a_3=177.26$$

$$144a_1+12a_2+a_3=279.2$$

The coefficients a1, a2, and a3 for the above expression are given by

$$\begin{bmatrix} 25 & 5 & 1 \\ 64 & 8 & 1 \\ 144 & 12 & 1 \end{bmatrix} \begin{bmatrix} a1 \\ a2 \\ a3 \end{bmatrix} = \begin{bmatrix} 106.8 \\ 177.2 \\ 279.2 \end{bmatrix}$$

Rewriting the equations gives

$$a_1 = \frac{106.8 - 5a2 - a3}{25}$$

$$a_2 = \frac{177.2 - 64a1 - a3}{8}$$

$$a_3 = \frac{279.2 - 144a1 - 12a2}{1}$$

Convergence Check: Repeat the iterative update until the solution converges, i.e., the change in the solution vector x between iterations is smaller than a predefined tolerance.

Jacobi Method: The Jacobi method is another iterative method used for solving systems of linear equations, particularly suitable for sparse matrices. It is simpler than the Gauss-Seidel method and can be parallelized more easily, although it may converge more slowly. Here's a detailed explanation of the Jacobi method, its application to sparse matrices, and its advantages and disadvantages.

Jacobi Method Overview

The Jacobi method iteratively updates the solution of the system of linear equations $Ax=b$ by using the values from the previous iteration for all variables. The algorithm can be summarized as follows:

Initialization: Starting with an initial guess for the solution vector $x(0)$.

Iterative Update: For each iteration k:

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j \neq i} a_{ij} x_j^{(k)} \right), \quad \text{for } i = 1, 2, \ldots, n$$

Here, aij are the elements of the matrix A, $x_{i(k+1)}$ is the updated value of the i-th component, and $x_{j(k)}$ are the values from the previous iteration.

Example 3.5: Express the following linear system in the Jacobi matrix notation.

$-2x_1 + x_2 + 1/2\, x_3 = 4$

$x_1 - 2x_2 - 1/2\, x_3 = -4$

$x_2 + 2x_3 = 0$

Solution: let $A = \begin{bmatrix} -2 & 1 & 1/2 \\ 1 & -2 & -1/2 \\ 0 & 1 & 2 \end{bmatrix}$ and $b = \begin{bmatrix} 4 \\ -4 \\ 0 \end{bmatrix}$

$$D = \begin{bmatrix} -2 & 0 & 0 \\ 0 & -2 & 0 \\ 0 & 0 & 2 \end{bmatrix}$$

$$L + D = \begin{bmatrix} 0 & -1 & -1/2 \\ -1 & 0 & 1/2 \\ 0 & -1 & 0 \end{bmatrix}$$

$$T_j = D^{-1}(L+U) = \begin{bmatrix} 0 & 1/2 & 1/4 \\ 1/2 & 0 & -1/4 \\ 0 & -1/2 & 0 \end{bmatrix}$$

$$C_j = D^{-1}b = \begin{bmatrix} -2 \\ 2 \end{bmatrix} \; 0$$

$$D = \begin{bmatrix} -2 & 0 & 0 \\ 0 & -2 & 0 \\ 0 & 0 & 2 \end{bmatrix}$$

$$L + U = \begin{bmatrix} 0 & -1 & -1/2 \\ -1 & 0 & 1/2 \\ 0 & -1 & 0 \end{bmatrix}$$

$$T_j = D^{-1}(L+U) = \begin{bmatrix} 0 & 1/2 & 1/4 \\ 1/2 & 0 & -1/4 \\ 0 & -1/2 & 0 \end{bmatrix}$$

$$C_j = D^{-1}b = \begin{bmatrix} -2 \\ 2 \\ 0 \end{bmatrix}$$

$$\begin{bmatrix} X1(k) \\ x2(k) \\ x3(k) \end{bmatrix} = \begin{bmatrix} 0 & 1/2 & 1/4 \\ 1/2 & 0 & -1/4 \\ 0 & -1/2 & 0 \end{bmatrix} \begin{bmatrix} x1(k-1) \\ x2(k-1) \\ x3(k-1) \end{bmatrix} + \begin{bmatrix} -2 \\ 2 \\ 0 \end{bmatrix}$$

Convergence Check: Repeat the iterative update until the solution converges, i.e., the change in the solution vector x between iterations is smaller than a predefined tolerance.

3.4 Matrix Factorisation: Matrix factorizations are essential tools in numerical linear algebra, providing ways to decompose matrices into products of simpler matrices. This is especially useful in solving linear systems, optimization problems, and understanding matrix properties. For sparse matrices, which contain many zero elements, specialized algorithms take advantage of the sparsity to improve computational efficiency and reduce storage requirements.

3.4.1 Cholesky factorization: Cholesky decomposition is especially effective for sparse symmetric matrices and positive definite matrices. The goal is to factorize matrix A into the product of the lower triangular matrix L and its transpose L^T.

$A=LL^T$

For sparse matrices, it is crucial to maintain the sparsity pattern to save computational resources and memory. Let's delve into the details of Cholesky factorization for sparse matrices, including some techniques and considerations.

Steps and Techniques

1. Reordering for Reduced Fill-In:
- Fill-ins means introducing non-zero elements into the matrix at positions that were originally zero during the factorization process. Various reordering techniques are used to minimize fill-ins.
- Minimum Degree Ordering: This heuristic reduces the amount of fill-in by reordering the matrix so that nodes with the smallest degree (number of edges) are processed first.
- Nested Dissection: This method recursively divides the graph representation of the matrix into smaller subgraphs, which can reduce fill-in by minimizing edge cuts.
2. Symbolic Analysis:
   Before performing numerical factorization, we perform symbolic analysis to determine the sparsity pattern of the factor L. This step does not involve any actual numerical computation, but it establishes the structure of L.
3. Numerical Factorization:
- Using the sparsity pattern from the symbolic analysis, the numerical values of L are computed. Efficient data structures, such as compressed sparse row or compressed sparse column, are often used to store the sparse matrix and its factors.
4. Multifrontal and Supernodal Methods:
- Multifrontal Method: This technique constructs a series of smaller dense subproblems (frontal matrices) that are solved independently. These solutions are then combined to build the final factor.
- Supernodal Method: This method groups columns of the matrix into supernodes, allowing for more efficient use of dense matrix operations within each supernode.

Example 3.6: Find the Cholesky decomposition for a matrix X whose lower triangular matrix is given by L = $\begin{bmatrix} 2 & 0 \\ 2-5i & 1 \end{bmatrix}$

Solution: the lower triangular matrix is given as

$$L = \begin{bmatrix} 2 & 0 \\ 2-5i & 1 \end{bmatrix}$$

The conjugate transpose of the above lower triangular matrix is:

$$L^* = \begin{bmatrix} 2 & 2+5i \\ 0 & 1 \end{bmatrix}$$

From the Cholesky decomposition X can be written as:

$X = LL^*$

$$X = \begin{bmatrix} 2 & 0 \\ 2-5i & 1 \end{bmatrix} \begin{bmatrix} 2 & 2+5i \\ 0 & 1 \end{bmatrix}$$

$$X = \begin{bmatrix} 4 & 4+10i \\ 4-10i & 29 \end{bmatrix}$$

3.4.2 QR factorization: QR factorization is a matrix factorization method in which a given matrix A is factorized into the product of two matrices Q and R. Matrix Q is an orthogonal (or unitary for complex numbers) matrix and R is an upper triangular matrix. This extension is particularly useful for solving linear systems, least squares problems, and eigenvalue calculations.

There are several methods for actually computing the QR decomposition. One of such method is the Gram-Schmidt process.

Consider the Gram-Schmidt procedure, with the vectors to be considered in the process as columns of the matrix A. That is,

$A = [a_1|a_2|a_3|\ldots\ldots\ |a_n]$

Then, $u_1 = a_1$, $e_1 = u_1 /||u_1||$,

$u_2 = a_2 - (a_2 \cdot e_1)\, e_1$,

$e_2 = u_2 /||u_2||$

$u_{k+1} = a_{k+1} - (a_{k+1} \cdot e_1)\, e_1 - \cdot\cdot\cdot - (a_{k+1} \cdot e_k)\, e_k$,

$e_{k+1} = u_{k+1}/\ ||u_{k+1}||$

Note that $||\cdot||$ is the L_2 norm.

Example 3.7: Consider the matrix

$$A = \begin{bmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \end{bmatrix}$$

with the vectors $a_1 = (1, 1, 0)^T$, $a_2 = (1, 0, 1)^T$, $a_3 = (0, 1, 1)^T$.

Performing the Gram-Schmidt procedure,

$u_1 = a_1 = (1, 1, 0)$,

$e_1 = u_1 /||u_1|| = 1/\sqrt{2}\ (1, 1, 0) = (1/\sqrt{2}, 1/\sqrt{2}, 0)$,

$u_2 = a_2 - (a_2 \cdot e_1)\, e_1 = (1, 0, 1) - 1/\sqrt{2}\ (1/\sqrt{2}, 1/\sqrt{2}, 0) = (1/\ 2, -\ 1/\ 2, 1)$,

$e_2 = u_2/\ ||u_2|| = 1/\sqrt{(3/)2}\ (1/\ 2, -\ 1/\ 2, 1) = (1/\sqrt{6}, -\ 1\ /\sqrt{6}, 2\ /\sqrt{6})$,

$u_3 = a_3 - (a_3 \cdot e_1)\, e_1 - (a_3 \cdot e_2)\, e_2$

$= (0, 1, 1) - 1/\sqrt{2}\,(1/\sqrt{2}, 1/\sqrt{2}, 0) - 1/\sqrt{6}(1/\sqrt{6}, -1/\sqrt{6}, 2/\sqrt{6}) = (-1/\sqrt{3}, 1/\sqrt{3}, 1/\sqrt{3})$

$e_3 = u_3 / \|u_3\| = (-1/\sqrt{3}, 1/\sqrt{3}, 1/\sqrt{3})$.

Thus,

$$Q = [e_1 \,|e_2\, |\ldots\ldots |e_n\,] = \begin{bmatrix} 1/\sqrt{2} & 1/\sqrt{6} & -1/\sqrt{3} \\ 1/\sqrt{2} & -1/\sqrt{6} & 1/\sqrt{3} \\ 0 & 2/\sqrt{6} & 1/\sqrt{3} \end{bmatrix}$$

$$R = \begin{bmatrix} a1.e1 & a2.e1 & a3.e1 \\ 0 & a2.e2 & a3.e2 \\ 0 & 0 & a3.e3 \end{bmatrix} = \begin{bmatrix} 2/\sqrt{2} & 1/\sqrt{2} & 1/\sqrt{2} \\ 0 & 3/\sqrt{6} & 1/\sqrt{6} \\ 0 & 0 & 2/\sqrt{3} \end{bmatrix}$$

All these algorithms are very useful. In the next chapter, we are going to understand the applications in deep.

CHAPTER 4

APPLICATIONS OF SPARSE MATRICES

Sparse matrices are particularly vital in scientific computing, where large-scale problems often involve matrices with a prominent number of zero elements. Applications in Scientific computing and its application in solving partial differential equations, finite element analysis, and computational fluid dynamics and in data science and machine learning and graph theory are discussed in detail below.

4.1. Application of Sparse Matrices in Scientific Computing: Computational science, also known as scientific computing or scientific computation, is a rapidly growing multidisciplinary field that uses advanced computing capabilities to understand and solve complex problems. It is a field of research that spans many disciplines, but at its core it is concerned with the development of models and simulations to understand natural processes.
sparse matrix computation as an important parallel pattern. There are many real-world applications of sparse matrix that involve modelling complicated phenomenon. In addition, sparse matrix computation is a simple example of data-dependent performance behaviour of many large real-world applications. Since the number of zero elements is large, compaction techniques are used to reduce the amount of accessing memory, storing, and calculating zero points.

4.2. Application of Sparse Matrices in PDEs

1. **Discretization:**
   Discretization is the simpler way to solve PDEs. Discretization of PDEs approximates them by equations that involve a finite number of unknowns. Generally, we get after large and sparse matrices after discretization; i.e., they have very few nonzero entries.
1. **Efficiency:**
   Utilizing sparse matrix techniques reduces the memory footprint and computational cost, enabling the solution of very large systems that would otherwise be infeasible with dense matrix techniques.
2. **Example:**

- Consider the 2D Poisson equation $\Delta u = f$ on a rectangular domain, discretized using a finite difference method. This results in a large sparse matrix representing the Laplacian operator. Efficient sparse solvers like Conjugate Gradient or Multigrid methods are then used to solve the system.

4.2.1. Finite Element Analysis

Finite Element Analysis is a numerical method for finding approximate solutions to boundary value problems in PDEs. Discretization approximates the PDEs with large sparse systems or numerical model equations, which can be solved using numerical

methods. The solution to the numerical model equations is an approximation of the real solution to the PDEs. The finite element method is used in the calculation of these approximations.

4.2.2 Computational Fluid Dynamics

Computational Fluid Dynamics involves the numerical simulation of fluid flows governed by the Navier-Stokes equations. These simulations are essential in fields like aerospace, automotive engineering, and weather forecasting.

Conclusion:

Sparse matrices are indispensable in scientific computing for solving PDEs, performing finite element analysis, and conducting computational fluid dynamics simulations. By leveraging the sparsity of matrices, computational efficiency and scalability are greatly enhanced, allowing for the solution of large-scale and complex problems in numerous scientific and engineering fields.

4.3 Applications of sparse matrix in Machine learning and Data Science:

Sparse matrices, which are matrices predominantly composed of zero elements, are widely used in data science and machine learning for a variety of applications. Their efficient storage and computational benefits are leveraged to handle large-scale data and complex models. Here are some key applications:

4.3.1. Natural Language Processing: The occurrence of words in a document can be represented as a sparse matrix, where the words in the document are only a small fraction of the words in the language. If we have a row for every document and a column for every word, each column stores the number of words that appear in the document with high percentage of zero.

4.3.2. Recommendation Systems: A sparse matrix can be used to represent which user watched the video.

4.3.3. Market Basket Analysis: Since the number of purchased items is tiny compared to the number of non-purchased items, a sparse matrix is used to represent all products and customers.

4.4. Basics of Graphs

Graphs and sparse matrices are used in computer science, especially networks. It is closely related to the representation of information networks such as here we explore the fundamentals of graphs and their representation using sparse matrices.

4.4.1. Definition

"A **Graph** is a finite set represented as $G = (V, E)$ where V is the set of vertices (nodes) and set E of edges defined as pairs of distinct vertices."

"If there is no distinction between the pair of vertices (u, v) and (v, u), the edges are represented by unordered pairs then the graph is **undirected graph**. If the pairs are ordered the graph is set to be **directed graph (Digraph)**." [7]

Example 4.1: The following graph shows the undirected and directed graph.

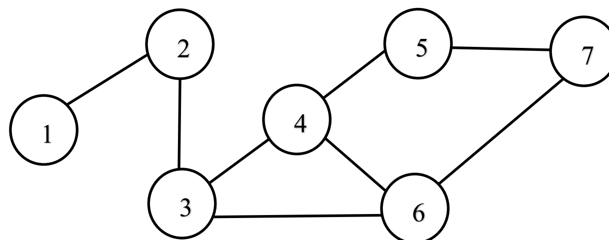


Fig. 1 Undirected

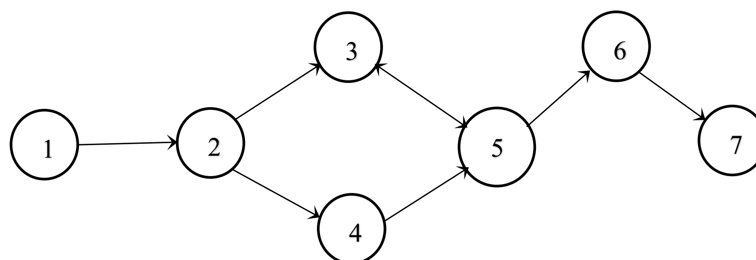


Fig. 2 Directed graph

In directed graph, there is an edge (3 → 5) and another edge (5 → 3).

In the **Fig. 1**, vertices 1 and 2, 2 and 3, 3 and 4 and so on are **adjacent** as there is an edge $e = (u, v)$ connecting the vertices u and v. We show the edge as $(u \leftrightarrow v)$ or $(u \overset{G}{\leftrightarrow} v)$. The **adjacency set** $adj_G\{u\}$ is the set of all the adjacent vertices, and the number of vertices belonging to V that are adjacent to $u \in V$ is said to be **degree** of u and can be written as $deg_G(u)$. [7]

Similarly, In the **Fig. 2**, here the notation is different to show the edge. We show the edge as $(u \rightarrow v)$ or $(u \overset{G}{\rightarrow} v)$ for a direct edge as there can be an edge $(u \rightarrow v)$ but not $(v \rightarrow u)$. As in this case, two directions can exist so the adjacent set will split into two parts as follows.

$$adj_G^+\{u\} = \{ v \mid (u \rightarrow v) \in \text{E}\} \quad \text{and} \quad adj_G^-\{u\} = \{ v \mid (v \rightarrow u) \in \text{E}\} \qquad [7]$$

For the vertex 2 in Fig. 2, $adj_G^+\{2\} = \{3,4\}$ and $adj_G^-\{2\}$ =1.

4.4.2. Basic terms

Before moving to the Graph Search Algorithms, let us understand some basic terminologies.

a) Walk: "When there is an undirected graph G a sequence of k edges is called the walk of length k."

$$u_0 \leftrightarrow u_1 \leftrightarrow \ldots. \leftrightarrow u_{k-1} \leftrightarrow u_k$$

When the G is a diagraph then the sequence is known as **Direct walk**.

$$u_0 \rightarrow u_1 \rightarrow \ldots. \rightarrow u_{k-1} \rightarrow u_k$$

b) Reachable: “If the vertices u_0 and u_k are connected by the walk for k > 0, u_k is said to be reachable from u_0. The set of the vertices that reachable from u_0 is denoted by $\boldsymbol{Reach}(\boldsymbol{u_0})$. “
c) Cyclic and Acyclic: “A walk is said to be **cyclic** if it is **closed** $(u_0 = u_k)$. If a graph does not have cycles, then it is **acyclic**.” [7]
d) Trail and Path: “A walk in which all the edges are distinct then it is a trail and if a trail has all the vertices are distinct.” It can be represented as $i \Leftrightarrow j$ (Undirect graph) and $i \Longrightarrow j$ (Diagraph) [7]
e) Length of the path: “The **length** is the no. of edges in the shortest path connecting the two vertices.” [7]
f) DAG: “A directed acyclic graph is called DAG. In case of DAG, if there is a path $u \Longrightarrow v$, then u is called an **ancestor** of v and v is called a **descendant** of u.”
g) Connected: “An undirect graph is **connected** if every pair of vertices is connected by a path.”
h) Tree: “A **tree** is an undirected graph in which any two vertices are connected by exactly one path.” [7] It can be represented as T.
i) Leaf: “In a tree if we have at least two vertices of degree 1, then such vertices are known as **leaf**.”
j) Forest: It is a graph which consists of disjoint union of trees.

Let us take an undirected tree $T = (V, E)$ can be changed to **Direct rooted tree** let say $T' = (V, E')$ by taking a vertex r as **root** vertex. “An edge $(u, v) \epsilon E$ becomes direct edge $(u \to v) \epsilon E'$ if there is a path from u to r such that the first edge of this path is from u to v. Consider, a directed edge $(u \to v) \epsilon E'$, u is said to be a **child** of v and v is said to be **parent** of u. Also, if two vertices in case of a rooted tree have same parent then they are said to be **siblings**. [7]

4.4.3. Adjacency Graphs

These graphs are highly important in order to get a clarity on the upcoming topics. Let a sparse matrix A of order n, “then the **adjacency graphs** $G(A) = (V(A), E(A))$ with *n* vertices can be associated with it.” [7]
In case of structurally symmetric matrix A, then the edge set can be represented as

$$E(A) = \{(i, j) \mid a_{ij} \neq 0, i \neq j\}$$

In case of a Diagraph when we are taking nonsymmetric A by using

$$E(A) = \{(i \to j) \mid a_{ij} \neq 0, i \neq j\}$$

4.4.4. Graph searches

“A **graph search** is basically used to perform step by step exploration of vertices and edges of G(A), and generates sets of visited vertices and explored edges. Let V_v be the set of vertices which we visited and V_n be the set of vertices which we have not visited yet.” In the search we explore the edges and whose one vertex should belong to V_v and if another vertex belongs to V_n, then this vertex is moved into V_v, and the edge is marked as explored so that we would not visit that edge again.
We will below discuss some of the search methods here.

4.4.5. Breadth-First search

For understanding, let us choose a start vertex k. In the Breadth-first search (BFS), we explore all the vertices adjacent to k. Then we will explore all those vertices whose distance from k is 2 and then we will choose vertices with distance 3 from k and so on until there are no unexplored edges (u, v) where $u \in V_v$ and $v \in V_n$ that are reachable from k. All the vertices which are at the same distance from the vertex k are placed at the same level say level 1,2,3…. and so on. For visiting a vertex there is no fixed order. [7]

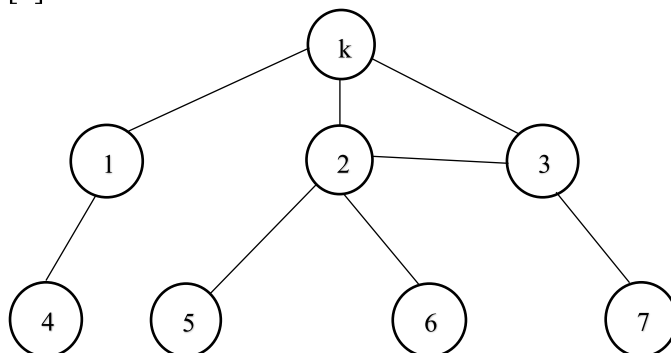


Fig. 3 An undirected graph of BFS, with the labels that shows the order in which we chose the vertices. Vertices 1, 2 and 3 are placed on the same level 1 as their distance is 1 from *k*. Similarly, vertices 4,5,6 and 7 are placed on second level.

4.4.6. Depth-First search

In depth search (DFS) of graph G, we first visit the children and then reach the sibling vertices. We have two access rows to vertices:

1. Pre-order DFS:

   In pre-order DFS, the node is processed before its children. We process the current node first and then the adjacent nodes. Let us choose a start vertex *k*, then we choose the first adjacent vertex to *k* until we reach to an unvisited node. Once we reach to an unvisited node, we move back to the previous node and check of there is any unvisited adjacent node if not then move back to *k.* This process will continue until there is no unvisited vertex left.

2. Post-order DFS:

   In post-order DFS, the node is processed after its children. We process the adjacent node first and then the current nodes. Here, the process remains the same, but in order we will mention those vertices where there are no further children.

CHAPTER 5

OPTIMIZATIONS, PARALLEL COMPUTING, CASE STUDY AND FUTURE DIRECTIONS

Optimization and parallel computing are two critical areas where sparse matrices play a significant role. The efficient handling of sparse matrices allows for solving large-scale optimization problems and leveraging parallel computing architectures to speed up computations. Here's a detailed look at how sparse matrices are utilized in these contexts.

Optimization problems often involve large-scale systems where many variables and constraints lead to sparse matrices. Efficiently managing and solving these sparse systems is key to practical optimization.

5.1 Large-Scale Optimization

5.1.1 Sparse Matrix Factorization:

Sparse Cholesky and LU factorizations are used to solve the linear systems arising in optimization problems. These factorizations take advantage of sparsity to reduce computational complexity and memory usage.

5.1.2 Conjugate Gradient and Krylov Subspace Methods:

A Krylov subspace, denoted as Km(A,b), is a subspace of a vector space generated by applying the sparse matrix A to a starting vector b and iteratively building the subspace. The Krylov subspace is defined as

$Km(A,b) = span\{b, A_1 b, A_2 b, \ldots, A_{m-1b}\}$

The Krylov subspace is often used in iterative methods for solving linear systems and eigenvalue problems. Common iterative methods like the Conjugate Gradient (CG) method and GMRES (Generalized Minimal Residual) method utilize Krylov subspaces.

Conjugate Gradients for Sx=b Compute the matrix-vector product Spk. Compute the step size αk using the formula $\alpha_k = p_k T S p_k r_k T r_k$. Update the solution: $x_{k+1} = x_k + \alpha_k p_k$. Update the residual: $r_{k+1} = r_k - \alpha_k S p_k$. Check for convergence. If the solution is accurate enough, stop the iterations. Compute the beta value: $\beta_{k+1} = r_k T r_k r_{k+1} T r_{k+1}$. Update the search direction: $p_{k+1} = r_{k+1} + \beta_{k+1} p_k$.

5.1.3 Application Example:

Network Flow Optimization: In telecommunications and transportation, network flow problems involve optimizing the flow through a network. The incidence matrix representing the network is sparse, and solving these optimization problems requires efficient sparse matrix techniques.

5.2 Parallel Computing Using Sparse Matrices

sparse matrix computation is a simple example of data-dependent performance behaviour of many large real-world applications. Because of the large amount of zero elements, compaction techniques are used to reduce the amount of storage, memory accesses, and computation performed on these zero elements and Parallel computing is a computing technique that breaks a problem into smaller tasks and runs them simultaneously. Its ability to handle multiple tasks simultaneously makes it faster than a sequential computer. Parallel computing helps solve large and complex problems in less time.

5.2.1 Parallel Sparse Matrix Operations

1. **Decomposition and Distribution:**

Sparse matrices are decomposed into submatrices that can be distributed across multiple processors. Techniques like domain decomposition are used in finite element methods to partition the problem domain.

2. **Parallel Solvers:**

Multigrid Methods: These methods solve large sparse linear systems efficiently by operating on multiple levels of grid resolution and are highly parallelizable.

Iterative Solvers: Parallel implementations of iterative methods like Conjugate Gradient, GMRES, and BiCGSTAB exploit sparse matrix-vector multiplication (SpMV), which can be parallelized.

3. **Libraries and Frameworks:**
   - PETSc: The Portable, Extensible Toolkit for Scientific Computation supports parallel sparse matrix operations and solvers, providing scalability and efficiency for large-scale scientific computations.
   - Trilinos: Offers a suite of parallel algorithms for sparse linear algebra and optimization, enabling scalable computations on distributed memory systems.
   - Intel MKL: The Math Kernel Library includes optimized routines for sparse matrix operations that are parallelized to take advantage of multi-core and many-core processors.

5.2.2 Application Example

1. **Computational Fluid Dynamics (CFD):**
   - Simulating fluid flow involves solving large sparse linear systems derived from discretizing the Navier-Stokes equations. Parallel sparse matrix solvers enable efficient handling of these large systems, allowing for detailed simulations in aerospace and automotive industries.
2. **Finite Element Analysis (FEA):**
   - Structural analysis using FEA generates large sparse stiffness matrices. Parallel computing techniques distribute the computation of element matrices and the assembly of the global stiffness matrix across multiple processors, significantly speeding up the analysis process.

Conclusion

In optimization, they enable the handling of vast systems of equations with numerous variables and constraints. In parallel computing, their structure allows for effective decomposition and distribution of computational tasks, harnessing the power of modern multi-core and distributed computing environments. Leveraging specialized algorithms and libraries, sparse matrices facilitate scalable and efficient solutions across various scientific and engineering applications.

5.3. Case study

The algorithms and representations that we studied in earlier sections are important when case study is considered.

5.3.1. Social Network Graph

In this section, we are considering the applications: Friend recommendation and Community detection.

Foundation:

a) User and Nodes: 1000 users represented as nodes.
b) Friendship and Edges: Represented as edges between nodes.
c) Adjacency Matrix: To represent the social network as a graph, we can use the adjacency matrix.

Elements: Each element of the matrix A_{ij} is:

a) 0 if there is no relationship between user i and j, i.e. there is no edge.
b) 1 if there is a friendship between user i and j, i.e. there is an edge between nodes i and j.

For simplicity, let us create a small graph:
Let's illustrate the creation of the adjacency matrix for the given social network with 1000 users, and then discuss how breadth-first search (BFS) can be applied to explore the network structure.

Step 1: Creating the Adjacency Matrix
Given that there are 1000 users in the social network, we'll have a 1000x1000 adjacency matrix to represent the connections between users. Initially, all entries in the matrix will be zero, indicating no friendships.

To populate the matrix, we would need additional information about the friendships among users. For simplicity, let's assume we have the following information:

- Each user is randomly connected to an average of 10 other users (friends).
- Friendships are bidirectional (If User A is a friend of User B, then User B is also a friend of User A).

We can then randomly generate the friendships and fill in the adjacency matrix accordingly.

For simplicity, let's assume we start the BFS from User 1.

Step 1: Initialization

- Start with User 1 as the initial node.
- Enqueue User 1 into the BFS queue.
- Mark User 1 as visited.

Step 2: BFS Iterations

Iteration 1: Explore User 1's Friends

- Dequeue User 1 from the queue.
- Look at User 1's row in the adjacency matrix to find its friends (nodes connected to User 1).
- Enqueue all unvisited friends of User 1 into the queue.

- From User 1's row, we see that User 1 is friends with Users 2 and 3.
- Enqueue Users 2 and 3 into the queue.
- Mark Users 2 and 3 as visited.

Iteration 2: Explore User 2 and User 3's Friends

- Dequeue User 2 from the queue.
- Look at User 2's row in the adjacency matrix to find its unvisited friends.
- Enqueue User 4 and User 5 into the queue (User 1 has already been visited).
- Dequeue User 3 from the queue.
- Look at User 3's row in the adjacency matrix to find its unvisited friends.
- Enqueue User 5 into the queue (User 1 has already been visited).

Iteration 3: Explore User 4 and User 5's Friends

- Dequeue User 4 from the queue.
- Look at User 4's row in the adjacency matrix to find its unvisited friends.
- Enqueue User 2 and User 5 into the queue (User 1 and User 3 have already been visited).
- Dequeue User 5 from the queue.
- Look at User 5's row in the adjacency matrix to find its unvisited friends.
- Enqueue User 2, User 3, and User 4 into the queue (User 1 has already been visited).

Iteration 4: Explore User 2, User 3, and User 4's Friends

- Dequeue User 2 from the queue.
- User 2's friends have already been visited, so no new nodes are enqueued.
- Dequeue User 3 from the queue.
- User 3's friend User 5 has already been visited, so no new nodes are enqueued.
- Dequeue User 4 from the queue.

- User 4's friend User 2 has already been visited, so no new nodes are enqueued.

Iteration 5: Explore User 5's Friends

- Dequeue User 5 from the queue.
- Look at User 5's row in the adjacency matrix to find its unvisited friends.
- Enqueue User 3 and User 4 into the queue (User 1 and User 2 have already been visited).

Iteration 6: Explore User 3 and User 4's Friends (Again)

- Dequeue User 3 from the queue.
- User 3's friend User 1 has already been visited, so no new nodes are enqueued.
- Dequeue User 4 from the queue.
- User 4's friend User 2 has already been visited, so no new nodes are enqueued.

Iteration 7: Explore User 3's Friend

- Dequeue User 3 from the queue.
- User 3's friend User 5 has already been visited, so no new nodes are enqueued.

Iteration 8: Explore User 4's Friend

- Dequeue User 4 from the queue.
- User 4's friend User 5 has already been visited, so no new nodes are enqueued.

Step 3: Termination

- The BFS algorithm terminates because all reachable nodes have been visited, and the queue is empty.

At this point, all nodes in the social network graph have been visited, and the BFS algorithm has explored the network starting from User 1, systematically traversing through the graph to discover its structure and relationships. This process helps us understand the connectivity patterns and identify communities or clusters within the social network.

5.3.2. Results

The result of applying BFS to the social network graph starting from User 1 is a traversal of the graph that systematically explores the network structure, identifying users who are directly or indirectly connected to User 1.
In this specific example, the BFS algorithm visited the following users:

a) User 1
b) User 2
c) User 3
d) User 4
e) User 5

This traversal indicates that User 1 is directly connected to Users 2 and 3, who in turn are connected to Users 4 and 5. Through this exploration, we have identified the immediate friends of User 1 and indirectly discovered the friends of User 2 and User 3.
Furthermore, we have also observed that Users 4 and 5 are indirectly connected to each other through mutual friendships with Users 2 and 3. This connectivity information can be valuable for various purposes, such as friend recommendation systems, community detection, or understanding the general composition of the social network.
Overall, result of the BFS traversal provides insights into the relationships and connectivity patterns within the social network, allowing us to understand how users are interconnected and how information or influence might propagate through the network.

5.4. Challenges and Future Directions

As we dive into the complex world of sparse matrices and their applications, it is imp ortant to recognize the challenges and consider the future directions of the field. Belo w is an introduction to the challenges and prospects for further research.
Sparse matrices have improved the technology by providing good solutions to proble ms related to big data and complex processes. However, many challenges remain that prevent their full use and require continued research.

5.4.1. Challenges

Scalability means the ability of a system or algorithm to manage increasing amounts of data or computing resources without sacrificing performance. In the context of sparse matrices, as datasets grow and computational requirements continue to increase, it is important to ensure that algorithms and systems can scale appropriately to these growing needs. and data processing are crucial to achieve high performance in low-matrix operations, especially distributed computing and high-performance computing (HPC). Parallel algorithms reduce computational time by allowing multiple computers to perform different operations on a problem simultaneously. However, it may be difficult to achieve similar results because of the irregular composition of sparse matrices and the memory access structure. These algorithms should effectively exploit the parallelism in matrix sparse operation while reducing communication overhead and underload on computers. Additionally, efficient use of

distributed computing resources, such as clusters of connected systems in HPC systems, is crucial to providing scalability for large matrices.

Efficient storage formats for sparse matrices are essential for minimizing memory usage and optimizing computational performance. Sparse matrices typically hold a large number of zero elements, makes it inefficient to store them explicitly. Various storage formats address this issue by storing only the non-zero elements with their corresponding row and column indices. Common formats include the Coordinate List (COO), Compressed Sparse Row (CSR), and Compressed Sparse Column (CSC) formats, each offering different trade-offs in terms of storage space and access efficiency. COO stores all non-zero element with their row and column indices, making it simple but potentially inefficient for certain operations. CSR and CSC formats compress the row or column indices, respectively, reducing storage overhead and enabling faster access to rows or columns. Choosing the most suitable storage format hangs on factors like the matrix's sparsity pattern, the types of operations performed, and memory constraints. Developing adaptive storage schemes that dynamically select the optimal format based on runtime conditions is an ongoing research area to further enhance storage efficiency for sparse matrices.

Algorithmic performance in the context of sparse matrices refers to the efficiency and effectiveness of algorithms designed to operate on sparse matrix data structures. Unlike dense matrices, which contain mostly non-zero elements, sparse matrices have a significant number of zero elements, leading to specific challenges in algorithm design and implementation. Improving algorithmic performance involves developing specialized algorithms tailored to exploit the sparsity of matrices efficiently. This includes designing data structures that minimize storage requirements, optimizing computational complexity to reduce time and memory overhead, and ensuring numerical stability and accuracy in computations. Algorithmic performance also encompasses considerations such as parallelization for efficient execution on multi-core CPUs or distributed computing platforms, as well as adaptability to different matrix structures and problem domains. Enhancing algorithmic performance in sparse matrix computations is essential for accelerating scientific simulations, machine learning algorithms, and various other computational tasks reliant on sparse data representations.

5.4.2. Future Directions

The field of sparse matrices continues to evolve with advancements in various domains such as computational mathematics, computer science, machine learning, and data science. Here are some potential future directions in this field:

Development of Sparse Neural Networks: As deep learning models become increasingly larger and more complex, there is growing interest in leveraging sparse matrices to decrease the computational and memory needs of neural networks. Research in this area aims to develop techniques for training and deploying sparse neural networks efficiently while maintaining high predictive performance.

Sparse Deep Learning Architectures: In addition to sparse neural networks, there is also interest in developing sparse architectures for other deep learning models such as convolutional neural networks (CNNs) and recurrent neural networks (RNNs). Sparse architectures can help reduce the memory footprint and computing cost of these models, makes them more practical for resource-constrained environments such as mobile devices and edge devices.
Sparse Optimization Techniques: Sparse matrices are often encountered in optimization problems arising in various fields such as machine learning, signal processing, and operations research. Future research in this area may focus on developing efficient optimization techniques specifically tailored for sparse matrices, including algorithms for sparse convex optimization, sparse nonconvex optimization, and distributed optimization on sparse data.
Sparse Graph Algorithms: Graphs are often represented using sparse matrices in algorithms for tasks such as network analysis, social network analysis, and recommendation systems. Future research in this area may focus on developing efficient algorithms for graph-related tasks that take advantage of the sparsity structure of the underlying matrices, including algorithms for graph traversal, clustering, and community detection.

The future of sparse matrices holds great promise across a multitude of domains, from machine learning and optimization to quantum computing and beyond. As researchers continue to innovate and develop new algorithms, data structures, and applications, we can expect to see further advancements that leverage the inherent efficiency and scalability of sparse representations. By addressing the current challenges and exploring new directions, the field of sparse matrices is poised to make significant contributions to the advancement of computational science and technology in the years to come.

CHAPTER 6
CONCLUSION

Sparse matrices are a fundamental concept in linear algebra and data science. They play a crucial role in managing and processing large datasets efficiently. Unlike dense matrices, where most elements have values, sparse matrices contain mostly zeros. This sparsity allows for specialized storage techniques and computational algorithms that significantly reduce memory usage and processing time.

Here's a breakdown of the key points covered in this chapter:

- **Importance of Sparse Matrices:**
  - Efficient storage and manipulation of large datasets.
  - Reduced memory requirements compared to dense matrices.
  - Faster computations due to skipping zero elements.
- **Properties of Sparse Matrices:**
  - Sparsity patterns: Arrangement of non-zero elements, influencing storage and algorithms.
  - Types of sparse matrices: Regular (diagonal, banded, block sparse) and irregular.
  - Theoretical properties: Rank, determinant, eigenvalues (impact solution methods).
- **Representations of Sparse Matrices:**
  - Coordinate list (COO): Stores row, column, and value for each non-zero element.
  - Linked list: Uses nodes to store element values, row/column indices, and pointers.
  - Compressed Sparse Row (CSR): Efficient for row-wise operations.
  - Compressed Sparse Column (CSC): Efficient for column-wise operations.
- **Basic Operations on Sparse Matrices:**
  - Addition, subtraction, multiplication, and transpose are performed efficiently by considering only non-zero elements.
- **Permutations and Reordering:**
  - Rearranging rows or columns to improve sparsity patterns and algorithm performance.

Sparse matrices are a powerful tool for handling large-scale data problems in various fields, including:

- **Machine Learning and Statistics:** Feature selection, recommender systems, text mining.
- **Scientific Computing:** Finite element analysis, graph theory, partial differential equations.

- **Signal Processing and Image Analysis:** Image compression, filtering, data reconstruction.

By understanding the concepts and techniques discussed in this chapter, you can leverage the benefits of sparse matrices to solve complex computational problems efficiently.

References

[1] (Davis, 2006; O'Connor, 2021; Rose, 1982; S. et al., 1991; Saad, 2003; Scott & Tůma, 2023; Strang, 2013, 2019)

[2] Davis, T. A. (2006). Direct Methods for Sparse Linear Systems. In *Direct Methods for Sparse Linear Systems*. https://doi.org/10.1137/1.9780898718881

[3] O'Connor, D. (2021). An introduction to Sparse Matrices. *Irish Mathematical Society Bulletin*, *0015*. https://doi.org/10.33232/bims.0015.6.30

[4] Rose, N. J. (1982). Linear Algebra and Its Applications (Gilbert Strang). *SIAM Review*, *24*(4). https://doi.org/10.1137/1024124

[5] S., G. W., Golub, G. H., & Loan, C. F. Van. (1991). Matrix Computations. *Mathematics of Computation*, *56*(193). https://doi.org/10.2307/2008552

[6] Saad, Y. (2003). Iterative Methods for Sparse Linear Systems, Second Edition. In *Methods*.

[7] Scott, J., & Tůma, M. (2023). Algorithms for Sparse Linear Systems. In *Necas Center Series: Vol. Part F1834*.

[8] Strang, G. (2013). Linear Algebra and its applications fourth edition. *Pressure Vessel Design Manual*.

[9] Strang, G. (2019). Linear Algebra and Learning from Data. In *Wellesley-Cambridge*.

PLAGIARISM VERIFICATION

Title of the Thesis – Sparse matrices in data science: efficient algorithms and applications with case study

Total Pages 40

Name of Scholars – Mallika Bisht (2K22/MSCMAT/61) and Niharika Srivastava (2K22/MSCMAT/27)

Supervisor- Mr. Jamkhongam Touthang

Department -Applied Mathematics

This is to report that the above thesis was scanned for similarity detection. Process and outcome is given below:

Software used: Turnitin

Similarity Index: 14%

Total Word Count: 12867

Date:

Candidate's Signature **Signature of Supervisor**

x

Similarity Report

PAPER NAME

SPARSE MATRICES.pdf

WORD COUNT	CHARACTER COUNT
12867 Words	**63898 Characters**
PAGE COUNT	FILE SIZE
48 Pages	**1.7MB**
SUBMISSION DATE	REPORT DATE
Jun 4, 2024 6:14 PM GMT+5:30	**Jun 4, 2024 6:15 PM GMT+5:30**

14% Overall Similarity

The combined total of all matches, including overlapping sources, for each database.

- 12% Internet database
- 6% Publications database
- Crossref database
- Crossref Posted Content database
- 8% Submitted Works database

Excluded from Similarity Report

- Bibliographic material
- Quoted material
- Cited material
- Small Matches (Less then 9 words)
- Manually excluded text blocks

DELHI TECHNOLOGICAL UNIVERSITY
(Formerly Delhi College of Engineering)
Shahbad Daulatpur, Main Bawana Road, Delhi-42

CERTIFICATE OF THESIS SUBMISSION FOR EVALUATION

1. Name: Mallika Bisht and Niharika Srivastava
2. Roll No.: 2K22/MSCMAT/61 and 2K22/MSCMAT/27
3. Thesis title: "Sparse Matrices in Data Science: Efficient Algorithms and Applications with case study".
4. Degree for which the thesis is submitted: M.Sc. Mathematics
5. Faculty of the University to which the thesis is submitted: Mr. Jamkhongam Touthang.

-

6. Thesis Preparation Guide was referred to for preparing the thesis. YES ☐ NO ☐
7. Specifications regarding thesis format have been closely followed. YES ☐ NO ☐
8. The contents of the thesis have been organized based on the guidelines. YES ☐ NO ☐
9. The thesis has been prepared without resorting to plagiarism. YES ☐ NO ☐
10. All sources used have been cited appropriately. YES ☐ NO ☐
11. The thesis has not been submitted elsewhere for a degree. YES ☐ NO ☐
12. Submitted 2 hard bound copies plus one CD. YES ☐ NO ☐

(Signature of Candidate)
Name(s): Mallika Bisht and Niharika Srivastava
Roll No.: 2K22/MSCMAT/61 and 2K22/MSCMAT/27

DELHI TECHNOLOGICAL UNIVERSITY

(Formerly Delhi College of Engineering)
Shahbad Daulatpur, Main Bawana Road, Delhi-42

CERTIFICATE OF FINAL THESIS SUBMISSION

1. Name: Mallika Bisht and Niharika Srivastava
2. Roll No.: 2K22/MSCMAT/61 and 2K22/MSCMAT/27
3. Thesis title: "Sparse Matrices in Data Science: Efficient Algorithms and Applications with case study".
4. Degree for which the thesis is submitted: M.Sc. Mathematics
5. Faculty of the University to which the thesis is submitted: Mr. Jamkhongam Touthang.

-

6. Thesis Preparation Guide was referred to for preparing the thesis. YES ☐ NO ☐
7. Specifications regarding thesis format have been closely followed. YES ☐ NO ☐
8. The contents of the thesis have been organized based on the guidelines. YES ☐ NO ☐
9. The thesis has been prepared without resorting to plagiarism. YES ☐ NO ☐
10. All sources used have been cited appropriately. YES ☐ NO ☐
11. The thesis has not been submitted elsewhere for a degree. YES ☐ NO ☐
12. All the correction has been incorporated. YES ☐ NO ☐
13. Submitted 2 hard bound copies plus one CD. YES ☐ NO ☐

(Signature of Candidate)
Name(s): Mallika Bisht and Niharika Srivastava
Roll No.: 2K22/MSCMAT/61 and 2K22/MSCMAT/27