UNIVERSITY OF OKLAHOMA

GRADUATE COLLEGE

COMPRESSION TECHNIQUES FOR EXTREME-SCALE GRAPHS AND MATRICES: SEQUENTIAL AND PARALLEL ALGORITHMS

A DISSERTATION

SUBMITTED TO THE GRADUATE FACULTY

in partial fulfillment of the requirements for the

Degree of

DOCTOR OF PHILOSOPHY

BY

SUDHINDRA GOPAL KRISHNA
Norman, Oklahoma
2023

COMPRESSION TECHNIQUES FOR EXTREME-SCALE GRAPHS AND MATRICES: SEQUENTIAL AND PARALLEL ALGORITHMS

A DISSERTATION APPROVED FOR THE
SCHOOL OF COMPUTER SCIENCE

BY THE COMMITTEE CONSISTING OF

Acknowledgements

I would like to express my sincere gratitude to the following individuals and institutions, without whose support and guidance, this dissertation would not have been possible:

First and foremost, I would like to thank my dissertation advisor, Dr. Sridhar Radhakrishnan, for their invaluable guidance, feedback, and support throughout my research journey. Their expertise, patience, and encouragement have been instrumental in shaping my ideas and refining my arguments.

I am also indebted to my committee members, Dr. Charles Nicholson, Dr. John Antonio, and Dr. Chongle Pan, for their insightful comments, suggestions, and critiques. Their collective expertise and diverse perspectives have helped me to approach my research questions from different angles and refine my arguments.

I would like to extend my appreciation to the University of Oklahoma and the School of Computer Science for providing me with the resources, facilities, and funding that enabled me to conduct my research. Would also love to especially thank Mrs. Virgine Perez-Woods, and Mr. Philip Johnson for their continued support and help throughout my degree.

I would like to acknowledge the countless individuals who have motivated me and guided me in every research stage. I would like to individually point out Dr. Amlan Chatterjee and Dr. Chandra N. Sekharan, who have constantly supported my research at stages.

Most importantly, I owe a debt of gratitude to my family and friends. Their unwavering love, support, and understanding throughout my academic journey have motivated and inspired me.

To my parents, Mrs. Mallika Gopal and Mr. Gopal Krishna, thank you for instilling in me a love of learning and a strong work ethic. Your encouragement and support, both emotional and financial, have been critical to my success.

To my partner, Mrs. Sukrutha Shivaram, thank you for being my rock and my biggest cheerleader. Your unwavering support, understanding, and patience during the many ups and downs of the dissertation process have been a source of comfort and inspiration.

To my siblings, Mr. Suraj Gopal, and Dr. Vaibhav Rao, thank you for always being there for me, whether listening to my research ideas or distracting me with your funny stories. Your love and support mean the world to me.

My heartfelt thanks go to my dear friends, Dr. Aditya Narasimhan, Mrs. Uma Kaipa, for their unwavering support and understanding throughout my dissertation journey. Your presence, advice, and encouragement have been instrumental in helping me navigate the complexities of this process. I am lucky to have such amazing friends who have always been there for me, no matter how busy or stressful my research got. Thank you for your unwavering friendship and support.

Finally, I would like to acknowledge OU Badminton, Bicycle League of Norman, and the countless individuals who offered me words of encouragement, provided a listening ear, or offered a helping hand when I needed it most. Your support and friendship have meant more to me than words can express.

Thank you all from the bottom of my heart.

Abstract

A graph $G = (V, E)$ is an ordered tuple where V is a non-empty set of elements called vertices (nodes), and E is a set of an unordered pair of elements called links (edges), and a time-evolving graph is a change in the states of the edges over time. With the growing popularity of social networks and the massive influx of users, it is becoming challenging to store the network/graph and process them as fast as possible before the property of the graph changes with the graph evolution.

Graphs or networks are a collection of entities (individuals in a social network) and their relationships (friends, followers); ways to represent a graph can help how the information could be extracted. The increase in the number of users increases the user's relationship, making the graphs massive and nearly impossible to store them in friendly structures such as a matrix or an adjacency list. Therefore, an exciting area of research is storing these massive graphs with a smaller memory footprint and processing with very little extra memory.

But there is always a trade-off with time and space; to get a small memory footprint, one must rigorously remove the redundancy, which consumes time. In the same way, when traversing these tight spaces, the time required to query also increases compared to a matrix or an adjacency list.

In this dissertation, we provide the encoding technique to store the graphs in the Compressed Sparse Row (CSR) data structure and extend the encoding to store time-evolving graphs as a CSR. We also propose combinations of two structures (CSR + CBT) to store the time-evolving graphs and to improve the time and space trade-off. Our compression technique also enables us to access any node without the need of decompressing the entire structure.

We then provide four ways to store multi-dimensional data representing intricate social network relations. Once the data are stored in compressed format, it is important to provide algorithms that support the structures. One such computation, the basis for any graph algorithm, is matrix multiplication. We now extend our work to perform value-based matrix multiplication on compressed structures. We test our algorithm on extremely large matrices in the order of 100s of millions with various sparsity levels. Using matrix-matrix multiplication and keeping the theme of storing the data in small spaces, we propose another way of compression through dimensionality reduction called Matrix Factorization.

Performing these operations on a compressed structure without decompressing would be time-consuming. Therefore, in this dissertation, we introduce a parallel technique to construct the graph and run a list of queries using the querying algorithms, such as fetching neighbor or edge existence in parallel. We also extend our work to propose parallel time-evolving differential compression of CSR using the prefix sum approach.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Graphs, in the real-world have become a pivotal tool to analyze data. But these real-world data are so large, it is nearly impossible to fit them into one's computer to perform analyses. Therefore, in this dissertation, we will focus on the ways, we can tackle the situation of storing these large data, and also provide algorithms to perform analyses, that supports the storage structure. Before we move on to looking at the storage techniques, let us first look at the definitions and some existing methodologies.

In this chapter, we first look at the definition of graphs in Section 1.1, how a graph can be represented as a matrix in Section 1.2, the operations that can be performed on the matrices in Section 1.3, later we discuss how graphs are exhibited in the real-world in Section 1.4, in Section 1.5 we discuss how graphs evolve over time, and what operations can be performed on these graphs in Section 1.6. Then we introduce the need for compression in Section 1.7, and at last we introduce the need for parallelism for compressing large graphs in Section 1.8.

1.1 Graphs

Graphs are fundamental mathematical structures used to represent relationships between objects. They consist of a set of vertices (also called nodes) and a set of edges that connect pairs of vertices. Graphs are commonly used to model and analyze various real-world systems and networks.

In a graph, vertices represent entities or elements, while edges represent the connections or relationships between them. These connections can be directed or undirected, depending on whether the edges have a specific direction or not. Directed edges indicate a one-way relationship, while undirected edges represent a two-way relationship.

Graphs can be further classified based on their characteristics:

- Weighted Graphs: In weighted graphs, each edge is assigned a numerical value or weight, representing some attribute or measure associated with the connection. These weights can denote distances, costs, or any other relevant metric.

- Unweighted Graphs: In unweighted graphs, edges do not have any associated weights. They simply indicate the presence of a connection between vertices without considering any specific attributes.

- Connected Graphs: A connected graph is one in which there is a path between every pair of vertices. In other words, there are no isolated vertices or disconnected components within the graph.

- Directed Acyclic Graphs (DAGs): DAGs are directed graphs that do not contain any cycles. A cycle is a sequence of edges that, when traversed, leads back to the starting vertex. DAGs are commonly used in modeling processes, dependencies, and directed flow of information.

- Bipartite Graphs: Bipartite graphs are graphs whose vertices can be divided into two disjoint sets, such that there are no edges connecting vertices within the same set. This property makes bipartite graphs useful for modeling relationships between two distinct groups of entities.

1.2 Matrices

One of the ways to store and represent a graph is through a matrix. Matrices can be used to represent graphs in a structured and systematic manner. In the context of graphs, a matrix representation is often employed to capture the connectivity and relationships between vertices.

The most common matrix representation for graphs is the adjacency matrix. An adjacency matrix is a square matrix, where the rows and columns correspond to the vertices of the graph. The entry at position (i, j) in the matrix indicates whether there is an edge between vertices i and j.

In an undirected graph, the adjacency matrix is symmetric, meaning that if there is an edge from vertex i to vertex j, there is also an edge from vertex j to vertex i. In a directed graph, the adjacency matrix may not be symmetric, as the presence or absence of an edge can differ depending on the direction.

The entries in the adjacency matrix can be binary, representing the presence or absence of edges, or they can be weighted to denote the strength or distance associated with the edges. A value of 1 usually indicates the presence of an edge, while a value of 0 indicates the absence of an edge. For weighted graphs, the values in the matrix represent the weights assigned to the edges.

1.3 Linear Algebra Operations

Linear algebra operations on matrices are vital for graph algorithms as they allow for concise representation, efficient manipulation, and analysis of graph data. These operations reveal important graph properties, enable the implementation of various graph algorithms, and facilitate solving systems of equations. Optimizations based on linear algebra techniques further enhance algorithm efficiency, making them essential for effective graph algorithm design and optimization.

- Matrix Addition: Matrix addition is an operation performed on two matrices of the same dimensions. It involves adding the corresponding elements of the matrices together. The resulting matrix has the same dimensions as the original matrices, and each entry in the result is the sum of the corresponding entries from the original matrices.

- Matrix Subtraction: Similar to matrix addition, matrix subtraction is performed on two matrices of the same dimensions. It involves subtracting the corresponding elements of one matrix from the corresponding elements of the other matrix. The resulting matrix has the same dimensions as the original matrices, and each entry in the result is the difference between the corresponding entries from the original matrices.

- Matrix Multiplication: Matrix multiplication is an operation performed on two matrices, but with specific requirements on their dimensions. The number of columns in the first matrix must be equal to the number of rows in the second matrix. The result is a new matrix where each element is computed as the sum of the products of the corresponding elements from the rows of the first matrix and the columns of the second matrix. Matrix multiplication is not commutative, meaning that the order of multiplication matters.

- Element-wise matrix operations involve performing an operation between corresponding elements of two matrices. In this type of operation, the elements in the same position (row and column) in both matrices are combined to produce a new matrix of the same dimensions.

- Matrix Transpose: The transpose of a matrix is obtained by interchanging its rows with columns. It is denoted by adding a superscript ”T” to the matrix. The resulting matrix has the dimensions reversed compared to the original matrix. In other words, if the original matrix is m x n, the transpose will be n x m. The transpose operation allows for various matrix manipulations and transformations and is useful in solving systems of linear equations, among other applications.

- Matrix Inverse: The inverse of a square matrix is a matrix that, when multiplied with the original matrix, yields the identity matrix. It is denoted by adding a superscript ”-1” to the matrix. Not all matrices have inverses; for a matrix to have an inverse, it must be non-singular or invertible. The inverse operation allows for solving systems of linear equations, calculating determinants, and performing other computations.

1.4 Types of Real-World Graphs

Real-world graphs, also known as complex networks, are graph structures that model relationships or interactions between entities in various real-world systems. These graphs capture the connections between objects, individuals, or entities in domains such as social networks, transportation networks, biological networks, and information networks.

Here are some examples of real-world graphs:

- Social Networks: Social networks represent relationships between individuals in social systems. Examples include online social networks like Facebook and Twitter, where vertices represent users, and edges represent friendships or connections between them.

- Transportation Networks: Transportation networks model the connections between locations or transportation infrastructure. Examples include road networks, where vertices represent intersections or road segments, and edges represent the connections between them, or airline networks, where vertices represent airports, and edges represent flight routes.

- Biological Networks: Biological networks represent interactions between biological entities, such as proteins, genes, or species. For example, protein-protein interaction networks model interactions between proteins, while gene regulatory networks represent interactions between genes and their regulatory elements.

- Information Networks: Information networks capture the flow of information or data between entities. Examples include the World Wide Web, where web pages are represented as vertices, and hyperlinks between them form the edges, or citation networks, where scientific papers are represented as vertices, and citations between papers form the edges.

- Collaboration Networks: Collaboration networks represent collaborations between individuals or entities. Examples include co-authorship networks, where vertices represent authors, and edges represent collaborations between them, or co-occurrence networks, where vertices represent words, and edges represent co-occurrences of words in documents.

1.5 Time-Evolving Graphs

Time-evolving graphs, also known as temporal graphs or dynamic graphs, are graph structures that capture the evolution of relationships or interactions over time. Unlike static graphs, which represent a snapshot of connections at a particular moment, time-evolving graphs provide a temporal dimension, allowing for the analysis of how relationships change and evolve over time.

In time-evolving graphs, the edges or attributes of the graph can change over time, reflecting the dynamic nature of the underlying system being modeled. For example, in a social network, edges may represent friendships, and the graph can capture the formation and dissolution of friendships as time progresses. Similarly, in a communication network, edges can represent interactions between individuals, and the graph can depict the communication patterns that emerge and change over time.

Analyzing time-evolving graphs has several significant implications. It allows for a deeper understanding of the dynamics, evolution, and temporal patterns of complex systems. By studying how relationships change over time, researchers can gain insights into the underlying processes, identify trends, and predict future behaviors. Temporal analysis of graphs also facilitates the detection of evolving communities, influential nodes, and structural changes within the network.

Furthermore, analyzing time-evolving graphs poses unique challenges compared to static graphs. Techniques for storage, indexing, querying, and visualization need to be adapted to handle the temporal dimension. Temporal graph mining algorithms and analysis methods need to be developed to extract meaningful insights from the evolving graph structures.

In summary, time-evolving graphs provide a powerful framework to study the dynamics of relationships and interactions over time. Analyzing these graphs offers valuable insights into evolving systems, facilitates prediction and decision-making, and drives advancements in various domains.

1.6 Graph Operations

Graph operations allow us to analyze and understand the structure, properties, and relationships within complex systems represented as graphs. By performing operations like traversal, clustering, or community detection, we can gain insights into the organization, connectivity, and patterns of interaction in various domains such as social networks, biological systems, or transportation networks.

Let's explore some common graph operations on both static (non-temporal) and time-evolving (temporal) graphs:

Graph operations on Static Graphs:

- Graph Traversal: Traversing a static graph involves exploring its vertices and edges to visit or search for specific nodes or paths. Popular graph traversal algorithms include depth-first search (DFS) and breadth-first search (BFS).

- Shortest Path: Finding the shortest path between two vertices is a fundamental operation in graph analysis. Algorithms like Dijkstra's algorithm and Bellman-Ford algorithm can be employed to determine the shortest path based on edge weights.

- Connected Components: Identifying connected components in a graph involves grouping vertices that are connected to each other via paths. This operation is useful for understanding the connectivity structure of a graph and detecting isolated clusters.

- Clustering and Community Detection: Graph clustering algorithms aim to partition the graph into clusters or communities based on the similarity or connectivity of vertices. Popular methods include modularity optimization and spectral clustering.

Graph Operations on Time-Evolving Graphs:

- Temporal Traversal: Traversing a temporal graph involves navigating through its vertices and edges while considering the temporal dimension. This operation requires taking into account the timestamps or intervals associated with the edges to track the temporal evolution of the graph.

- Temporal Path Analysis: Finding temporal paths in a time-evolving graph involves identifying paths that follow a specific temporal order of edges. This operation is useful for studying temporal dependencies or tracing temporal sequences of events.

- Temporal Reachability: Analyzing temporal reachability focuses on determining whether a vertex is reachable from another vertex within a specific time window. This operation helps understand the temporal accessibility or influence between nodes.

- Temporal Community Detection: Temporal community detection algorithm aims to identify communities or clusters in time-evolving graphs based on the temporal patterns of interactions. These algorithms take into account both the structural properties and the temporal dynamics of the graph.

- Temporal Graph Matching: Matching or aligning temporal graphs involves finding correspondences between vertices or edges across different time slices. This operation allows for tracking entities or relationships across time and studying their evolution.

1.7 Compression

Graphs are widely used to model complex systems and relationships, ranging from social networks and biological interactions to transportation networks and information systems. However, as the size of these networks continues to grow exponentially, the need for efficient storage and processing of graph data becomes crucial. Graph compression techniques offer a solution by reducing the space required to store and operate on large-scale graphs without sacrificing important structural and connectivity information.

Graph compression is the process of representing and storing graphs in a compressed form, allowing for efficient storage, retrieval, and analysis of graph data. It aims to minimize the storage footprint while preserving the key characteristics of the graph, such as vertex and edge relationships, connectivity, and graph properties. By compressing graphs, it becomes possible to handle extremely large graphs within limited resources, enabling more scalable and faster graph processing.

The main objective of graph compression is to achieve a trade-off between storage space and query performance. Different graph compression techniques employ various strategies to reduce redundancy and exploit patterns within the graph structure. These techniques leverage concepts from data compression, graph theory, and algorithms to optimize the representation and storage of graph data.

Compression in general is divided into two category, lossy and lossless.

1.7.1 Lossy Compression

Lossy compression is a compression technique that achieves higher compression ratios by selectively discarding or approximating certain parts of the data that are considered less important or perceptually less significant. In lossy compression, some amount of information is lost during the compression process, and the decompressed data is not an exact replica of the original data. Lossy compression techniques are often used in scenarios where some degree of loss or degradation in quality can be tolerated without significantly affecting the overall usefulness of the data. Common examples of lossy compression include image compression techniques like JPEG, audio compression techniques like MP3, and video compression techniques like MPEG.

1.7.2 Lossless Compression

Lossless compression is a compression technique that allows for the exact reconstruction of the original data from the compressed form. In lossless compression, no information is lost during the compression process. The compressed data retains all the original data, and when decompressed, it is identical to the original data. Lossless compression techniques typically exploit redundancies and patterns in the data to represent it in a more compact form. Examples of lossless compression algorithms include Run-Length Encoding (RLE) (Robinson and

Cherry, 1967), Huffman coding (Huffman, 1952), and Lempel-Ziv-Welch (LZW) (Ziv and Lempel, 1978) compression.

The choice between lossless and lossy compression depends on the specific requirements of the application and the trade-offs between compression ratio and fidelity. Lossless compression is typically preferred when exact preservation of the data is essential, such as in text documents or program files. Lossy compression, on the other hand, is commonly used in multimedia applications where a certain level of imperceptible loss can be accepted to achieve higher compression ratios, such as in images, audio, and video files.

In the context of graph compression, similar principles apply. Lossless graph compression techniques aim to preserve the exact structure and connectivity of the graph, allowing for accurate analysis and query operations. Lossy graph compression techniques may sacrifice certain details or properties of the graph to achieve higher compression ratios, which can still be useful in scenarios where approximate graph information is sufficient for the intended analysis or application.

1.7.3 Queryable Compression

Queryable compression on graphs refers to the capability of compressed graph representations to support efficient and effective query operations. It involves compressing graph data in a way that preserves important structural information and allows for querying and retrieving specific information or performing graph operations on the compressed data without the need for full decompression. Some of the well-known lossless queryable compressions were introduced by Nelson et. al., (Nelson et al., 2017)(Nelson et al., 2018)

In queryable compression, the compressed graph representation is designed to enable various types of queries or operations on the graph, such as graph traversal, neighborhood exploration, shortest path computation, or subgraph matching. The goal is to achieve a balance between compression ratio and query performance, where the compressed graph occupies less storage space while still providing fast access and retrieval of relevant information.

To achieve queryable compression, different techniques can be employed:

- Indexing: Indexing structures can be built on top of the compressed graph to facilitate efficient query processing. These indexes store additional metadata or auxiliary structures that allow for fast lookup and retrieval of specific graph elements or properties

- Preprocessing: Preprocessing techniques can be applied during the compression process to extract and store important graph properties or summaries that are relevant for query operations. These precomputed summaries or properties enable faster query processing by avoiding the need to decompress the entire graph.

- Compression Trade-offs: The choice of compression algorithm and parameters can have an impact on the query performance. Some compression techniques, while achieving high compression ratios, may introduce higher overhead in terms of query processing. It becomes important to strike a balance between compression efficiency and query performance based on the specific requirements of the application.

1.7.4 Matrix-Based Compression

Matrix-based compression treats the graph as an adjacency matrix, where rows and columns represent vertices, and the matrix entries indicate the presence or absence of edges between vertices. This approach leverages compression techniques specifically designed for matrices. Some of the most common matrix-based compression are $ck^d - tree$ Compression (Caro et al., 2016), Suffix-Array Strategy Compression (Brisaboa et al., 2014a), and so on.

Advantages of Matrix-based Compression:

- Compact Representation: Matrix-based compression techniques can achieve high compression ratios by exploiting the sparsity of the graph matrix, as many real-world graphs exhibit sparsity.
- Efficient Matrix Operations: Once compressed, matrix-based representations allow for efficient matrix operations like matrix multiplication, which can be advantageous for certain graph algorithms and computations.
- Well-Studied Techniques: Matrix compression techniques have been extensively studied and optimized, with various algorithms and tools available for compression and decompression.

Disadvantages of Matrix-based Compression:

- High Memory Overhead: The compressed matrix may still require substantial memory to store, especially for large graphs. In some cases, the compression ratios achieved may not be sufficient to fit the graph within the available resources.

- Limited Query Flexibility: Querying individual elements or properties of the graph may require decompressing the entire matrix, which can be inefficient for certain types of graph operations.

- Lack of Scalability: Matrix-based compression techniques may struggle to handle extremely large graphs due to memory limitations and computational complexity.

1.7.5 Row-by-Row Compression

Row-by-row compression approaches compress the graph by compressing individual rows of the adjacency matrix independently. Each row represents the adjacency information of a vertex, and compression techniques are applied to each row separately. Some of the most common row-by-row compression are Backlinks Compression (Chierichetti et al., 2009), Web-Based Compression (Boldi and Vigna, 2004), and so on.

Pros of Row-by-Row Compression:

- Selective Decompression: Row-by-row compression allows for selective decompression of specific rows or vertices, enabling targeted retrieval of information without decompressing the entire graph.

- Flexibility in Querying: The compressed representation supports efficient querying of specific rows or neighborhood information, making it suitable for various graph operations and algorithms.

- Lower Memory Overhead: Row-by-row compression can achieve better memory utilization compared to matrix-based compression, as only the necessary rows need to be decompressed for a given operation.

Cons of Row-by-Row Compression:

- Limited Matrix Operations: Unlike matrix-based compression, row-by-row compression may not directly support efficient matrix operations like matrix multiplication, which can be disadvantageous for certain graph algorithms that heavily rely on matrix operations.
- Lower Compression Ratios: In some cases, row-by-row compression may achieve lower compression ratios compared to matrix-based compression techniques, as it focuses on compressing individual rows rather than the entire matrix.
- Trade-off Between Query Efficiency and Compression Ratio: The level of compression achieved with row-by-row compression may impact the efficiency of query operations, and finding the optimal balance between compression and query performance can be challenging.

The choice between matrix-based compression and row-by-row compression depends on the specific characteristics of the graph, the desired query and computation tasks, and the available resources

1.8 Parallel Graph Compression and Computation

As the size of graphs continues to grow exponentially, the need for efficient compression techniques becomes crucial to handle the storage and processing challenges associated with large-scale graph data. Parallelism in graph compression accelerates the compression process by leveraging multiple processors or computing units. It divides the compression tasks into smaller subtasks that can be processed simultaneously. These computing units can be CPU cores, GPUs, or even distributed computing systems. Each unit works on a subset of the graph data or performs compression operations on different parts of the graph simultaneously, enabling faster compression compared to sequential processing.

Data parallelism and task parallelism are common approaches, distributing the workload across computing units or stages of the compression algorithm. Distributed computing frameworks can be used for extremely large graphs. Parallel compression offers faster compression, and scalability, and utilizes modern hardware. Challenges include load balancing and synchronization overhead.

Overall, parallelism provides a powerful solution to compress large graphs efficiently and reduce compression time.

The remainder of the dissertation is organized as follows. In Chapter 2, we discuss existing techniques to store time-evolving graphs, and tensors, and existing work on matrix-matrix multiplication, matrix-factorization, and finally parallel-based graph compression. In Chapter 3, we introduce techniques to store time-evolving graphs, and we also introduce improvements on the existing technique. In Chapter 4, we introduce multiple techniques to store tensors. In Chapter 5, we introduce matrix-matrix multiplication on the compressed structures, and in Chapter 6, we use the multiplication algorithm along with a few other matrix operations to perform matrix factorization. In Chapter 7, we introduce a technique to store static graphs in parallel on a highly sequential data structure, and also we propose a technique to store time-evolving graphs in parallel. We conclude our work in Chapter 8, by summarizing all the contributions.

Chapter 2

Literature Survey

In this chapter, we discuss all the existing techniques present for the work on time-evolving graph compression, tensor compression, matrix-matrix multiplication on the compressed structure, matrix-factorization on the compressed structure and the parallel construction of the compressed structures.

2.1 Time-Evolving Graphs

A time-evolving graph can be represented as a sequence of static graphs (snapshots), with each of the snapshots representing the graph at a particular point in time. Since a snapshot can be represented as a 2D matrix, a time-evolving graph can therefore be represented as a 3D matrix, also known as *presence matrix* (Ferreira and Viennot, 2002).

In 2009, Chierichetti et al. (Chierichetti et al., 2009) modified the web compression method developed by Boldi and Vigna's WebGraph (Boldi and Vigna, 2004) called Backlinks Compression (BLC). The compression is based on the social network's property of reciprocity. The compression technique makes use of intrinsic ordering heuristics based on shingles, which improves on the WebGraph format.

Nelson et al. (Nelson et al., 2017) in 2017 introduced a compressed data structure as an indexed array of Compressed Binary Tree (CBT). The data structure eliminates the necessity of an intermediate structure to create the compressed binary tree. The data structure also makes use of row-by-row compression which enables faster access to the edge existence, neighbor query, and the streaming operation.

In 1976, Compressed Sparse Row (CSR) was first documented by Snay (Snay, 1976b), and is one of the most common data structures used to represent a graph. The compressed sparse row is also a row-by-row compression that involves two arrays for the compression of each node. All the information is efficiently packed in the array for quick traversal of the data structure. The first array shows the degree of each node, and the following array shows the edge incidence for each node. Here, the degree of a node v is the number of edges incident to v, and is denoted as $d(v)$.

In 2016, Caro et al. (Caro et al., 2016) developed $ck^d - trees$. They define a contact as a quadruplet (u, v, t_i, t_j) and then compress the 4D binary matrix corresponding to the time-evolving graph defined by a set of these contacts. It is done by representing the 4D matrix as a $k^d tree$ and then distinguishing white nodes as those without any contacts, black nodes as those that contain only contacts, and gray nodes as those that contain only one contact. This work was preceded by Brisaboa et al. $k^2 - trees$ (Brisaboa et al., 2014b) in 2014.

G^* database (Labouseur et al., 2015) is a distributed index that solves the space issue of the presence matrix by only storing new versions of an arc when its state changes, i.e. as a log of changes. This is done by storing versions of the vertices as adjacency lists and maintaining pointers to each time frame. If an arc changes in the next frame, a new adjacency list is created for that vertex's arc and a pointer is added to the new frame.

Caro et al. (Caro et al., 2015) proposed a compressed adjacency log structure based on the *log of events* strategy called EveLogs. It consists of two separated lists per vertex, one for the time frames, and another for representing the arcs related to the event. The time frames are compressed using gap encoding, and the arc list is compressed with a statistical model. Caro et al. (Caro et al., 2016) show that query times suffer when the log is sequentially scanned.

Ren et al. (Ren et al., 2011), developed the FVF (Find-Verify-Fix) framework which includes a copy+log compression that also supports shortest-paths and closeness centrality queries. More preliminary work is done in (Álvarez-García et al., 2014) (Bernardo et al., 2013), which describes three different methods to index time-evolving graphs based on the *copy+log* strategy.

Two *log of events* strategies, CAS and CET, are proposed in (Caro et al., 2015) to address the problem of slow query times when processing a log. CAS orders the sequence by vertex and adds a Wavelet Tree (Grossi et al., 2003) data structure to allow logarithmic time queries. CET orders the sequence by time, and the authors develop a modified Wavelet Tree called Interleaved Wavelet Tree to also allow logarithmic time queries.

In 2014, Brisaboa et al. (Brisaboa et al., 2014a) adapted Compressed Suffix Arrays (CSA) as in (Caro et al., 2015) for use in temporal graphs (TGCSA) by treating the input sequence as the list of contacts. They use an alphabet consisting of the source/destination vertices and the starting/ending times.

2.2 Tensor Representation

There have been extensive work done in the field of either data compression or matrix multiplication on a HPC, at a time. But, few have managed to club both of these techniques together. The ones that have both compression and matrix multiplication on the compressed structure, often contain an intermediate structure to hold the result of the multiplication before storing the resultant matrix.

The work on compressed sparse matrix formats is extensively discussed in (Langr and Tvrdik, 2015; Zhang and Gruenwald, 2018; Zachariadis et al., 2020; Gopal Krishna et al., 2021). We will discuss some of the most popular formats. The Coordinate list technique (COO) is a sparse matrix format that stores all non zero elements in the matrix as an array along with each of their index positions (i, j).

In the context of parallel computing, the Coordinates list technique (COO) has been demonstrated to be one of the least efficient formats (Zachariadis et al., 2020). Therefore, it has been extended and improved in (Zhang and Gruenwald, 2018) with bmSparse. Furthermore, the authors in (Zhang and Gruenwald, 2018) demonstrated that bmSparse is 32 times more efficient than COO.

Compressed Sparse Row (CSR) (Snay, 1976a) is another popular compression scheme that has been adapted for a variety of platforms, including GPGPUs (NVIDIA, 2022) and has been adapted by the authors in the compressed context for time-evolving graphs in (Krishna et al., 2021; Gopal Krishna et al., 2021).

We proceed to discuss more about various matrix multiplications from a parallel processing perspective. We start by exploring the first usage of CBT for performing matrix operations. CBT (Nelson et al., 2019) by sequentially multiplying two binary matrices. The interesting contribution here is that the resultant is also stored in the CBT structure directly without usage of any intermediate structures. To improve that work, the authors in (Krishna et al., 2021) extended CBT to a value-based sequential matrix-matrix multiplication and compared it with CSR-based matrix-matrix multiplication.

Sparse General Matrix Multiplication (spgemm), being a fundamental kernel, has a multitude of applications and implementations (Lin et al., 2014; Deveci et al., 2018; Wolf et al., 2017). However, one of the challenges of multiplying two sparse matrices is managing the intermediate results of the output matrix.

In order to address that issue, the Expansion, Sorting, and Contraction algorithm (ESC) from (Dalton et al., 2015) takes intermediate results during the calculation of matrix-matrix multiplication and stores them in a tuple format (index and value) that later is sorted and combined based on the common index positions.

Zachariadis etȧl(Zachariadis et al., 2020) provide a matrix-matrix multiplication technique that takes these CUSP and ESC methods and modifies them to an approach called *tSparse*, which uses a modified ESC called Sort-Expand and Compress (SEaC).

The usage of the ESC strategy is not necessary in our multiplication technique as we provide a way that out intermediate results are directly stored as CBT structures.

We investigate to verify the claims we make about the storage used by all the formats mentioned above. We consider the *friendster* dataset from (Leskovec and Krevl, 2021), which has 65.6 million nodes (n_f) and 1.8 billion edges (nnz_f) (edges correspond to nonzero elements and nodes correspond to the number of rows and columns). We assume that all these representations are binary and we

Table 2.1: Comparing the storage for various Boolean matrices in various representations/compression techniques.

	Number of rows and columns (n)	Number of nonzero values (nnz)	Adjacency matrix	COO	CSR	tSparse (modified bmSparse)	CBT
Friendster	65,608,366	1,806,067,135	489.36 T	26.91 G	13.95 G	13.58 G	4.4 G
Live journal	3,997,962	34,681,189	1.82 T	529.19 M	295.10 M	272.22 M	64.2 M
10 million	10,000,000	19,999,996	11.37 T	305.18 M	228.88 M	171.66 M	59.6 M
100 million	100,000,000	10,000,000	1,136.87 T	152.59 M	839.23 M	267.03 M	47.68 M
1 billion	1,070,000,000	91,070,000,000	127.10 PB	1.32 T	680.51 G	680.51 G	9.5 G

do not consider the values stored in these matrices. The indices and/or numbers are all assumed to be stored as 64-bit integers. The baseline memory footprint would be an adjacency matrix representation to store the Friendster matrix. This would require 7.38 TB of memory to store. When looking into the COO format to store the above matrix, we calculate the memory required using the formula $nnz_f \times 2 \times int64$. This would result in needing 26.82 GB. We continue to calculate the memory requirement for CSR using the formula $n_f \times int64 + nnz_f. \times int64$, which would yield a memory footprint of 13.89 GB.

When considering the bmSparse technique with the assumption from tSparse having an 8×8 block, we calculate the memory footprint using the formula used in (Zhang and Gruenwald, 2018), $n\_block \times int64 + n\_block \times int64 + nnz_f \times int64$. This would require 13.5 GB of memory. Finally, when comparing this with the actual memory used by CBT, by experimental evaluation, we find that it takes 4.4 GB. This is much less than the others mentioned and is so because of the way the CBT is data-dependent.

The storage capacities are further compared in Table 2.1 considering other datasets (real world and synthetic).

2.3 Matrix-Matrix Multiplication

Matrix compression and multiplication have been traditional problems that have been existing since the early 1800s. The recent popular application is NMF (Berry et al., 2007)(Lee and Seung, 2001)(Shitov, 2017).

The problem of *Nonnegative Matrix Factorization(NMF)* can be formally defined as the follows: Given a nonnegative matrix $A \in \mathbb{R}_+$ of dimension $m \times n$ and an integer $k > 0$, find the factor matrices if any, $W \in \mathbb{R}_+$ of dimension $m \times k$ and $H \in \mathbb{R}_+$ of dimension $k \times n$ such that:

$$A = WH$$

These matrices W and H are also nonnegative and the value of k which is referred to as the *inner dimension* is found to have as lower bound the positive rank($rank_+$) of the matrix A. This problem of finding the factors that satisfy the condition $A = WH$ with the $rank_+(A) = k$ is proved to be an NP-hard problem. The proof for the NP-hardness of NMF by Vavasis(Vavasis, 2010), has been a standard reference in applied mathematics. Shitov(Shitov, 2017) gives a short proof for this theorem. There is a huge spectrum of applications that use NMF ranging from text mining and computer vision to clustering in machine learning. And a large set of these applications are satisfied and work with approximations of this factorization problem. Now the problem can be rewritten as the following with the $\|\|_F$ as the Frobenius norm:

$$\min_{W \geq 0, H \geq 0} \|A - WH\|_F$$

The paper by Lee and Seung (Lee and Seung, 2001) discusses two variations to the multiplicative update algorithm amongst many, for NMF. They differ based on the update functions of the factors. The implementation of these algorithms is often considered easier than other similar algorithms. However, the compromise is the time it takes for convergence. This just means that the algorithm needs to run for more iterations performing repeated matrix multiplications, $W^T A$ and $A^T H$. *The beauty of our algorithm from this paper is that these updated A matrices after each iteration are also stored in our compressed structures. At each iteration, the repeated multiplication uses no intermediate structures as well.*

Another perspective for improving the efficiency to do NMF is to take a parallel implementation route. This has been looked into extensively in (Kannan et al., 2017) and (Kaya et al., 2018).

As the order/dimensions of these input matrices get into the billions, to perform these algorithms in small spaces, we look into the compression techniques that are present. Specifically, the two algorithms that are used here are row-by-row compression algorithms. Compressed Sparse Row (CSR) has been a popular compressed data structure to store matrices since the first mention in (Snay, 1976a). This specifically is a great way to store value-based matrices because of the way the values are stored alongside the coordinates in the matrix where they are found. There have been other row-by-row compression algorithms that are more specialized for compressing binary matrices. This leads us to the history of the Compressed Binary Tree (CBT) data structure (Nelson et al., 2017). The data structure is an improvement from the k^2 - tree data structure (Brisaboa et al., 2014b). CBT was the first to store each row as a binary tree. They also use different encoding schemes to compress and store two matrices using differential techniques.

Binary multiplication performed using CBT structure (Nelson et al., 2019), does this with a differential row-by-row implementation of CBT as well. Value-based multiplication has been done using Single Tree Adjacency Forests (STAF) (Nishino et al., 2014) but has a constraint that the matrices need to follow the column nonzero property. In (D'Azevedo et al., 2005), multiplication is performed using CSR by using a vectorization approach to speed up the process. A lot of approaches have been implemented and presented in the direction of a parallel implementation for multiplying CSR matrices, but we do not discuss that here.

All Pair Shortest Path algorithms (Floyd, 1962) are another application where repeated matrix updates happen in the order of n^3. When using the compressed data structures from this paper, the matrix computations use no intermediate structures to perform these in a repeated manner.

The partial sum idea was first used on a data structure called Single Tree Adjacency Forest (STAF) in (Nishino et al., 2014). The implementation and the idea is to be used during matrix-matrix multiplication to reduce the number of addition and multiplication operations during the dot product intermediate steps. Although the partial sum idea was used here, a major assumption of the STAF data structure is that the matrices have to meet the column-scaled nonzero property. In this paper, we use this partial sum idea to reduce the number of getRow() queries in our data structure which reduces the resources by a lot.

2.4 Matrix Factorization

The foundation for the Non-negative matrix factorization was laid by Lee and Seung (Lee and Seung, 2001) in 1999, opening the opportunity to hundreds of research journals. Before Lee and Seung, few other notable contributions were made in the area of NMF, but none came close to the fame of Lee and Seung. Paatero and Tapper, 1994 (Paatero and Tapper, 1994), produced the work on positive matrix factorization. Lee and Seung cite the work of Paatero and Tapper in their work. Articles have shown the significance of Paatero's work prior to Lee and Seung but have gone unnoticed.

Since Lee and Seung's NMF was one of the first ones to be popular, it became a baseline for many research. Several researchers have proven that the multiplicative update algorithm proposed by Lee and Seung (Lee and Seung, 2001), is slower to converge, which means that it takes many more iterations to complete compared to the gradient descent method and the alternating least squares. Each implementation required a total of 12 matrix operations, of which six require $O(n^3)$ matrix-matrix multiplication, and the rest require $O(n^2)$ matrix-matrix element-wise operations.

To overcome this issue, other researchers, such as Gonzalez and Zhang in 2005 (Gonzalez and Zhang, 2005), proposed an alteration to the multiplicative update, but it ended up having the same convergence issue. Another researcher named Lin (Lin, 2007) in 2007 proposed a modification that ended with earlier convergence but at the cost of more operations per iteration.

Theoretically performing 12 matrix operations on a matrix is time- and space-consuming; performing the same operation on larger matrices would require a great deal of memory. For example, a $65,536 \times 65,536$ requires about 32 GB of storage in its raw format. To overcome this, in this paper, we use our novel CBT (Nelson et al., 2018), which works well with binary matrices and the bit-packing algorithm proposed in (Gopal Krishna et al., 2021) to store integer values. We also propose to store the matrix in CSR (Snay, 1976a), a common data structure for storing matrices.

To perform factorization or any operation on large sparse matrices, one must efficiently store the matrices so that the entire data can be loaded onto the main memory in one go. Given a matrix of size n rows and m column, the total number of possible elements in the matrix is the size of the matrix itself, which $m \times n$, therefore, the cost of storing a matrix in raw format would require $(m \times n) \times 64$ number of bits, where 64 is the number of bits required to store a number. But in a sparse matrix, this number tends to be very small, where the number of non-zero elements is extremely less compared to the number of zeros.

Therefore, the sparsity of a matrix is defined as the ratio of the number of non-zero elements to the number of all possible elements that can be in the matrix.

$$Sparsity = \frac{nnz}{m \times n},$$

where nnz is the number of non-zero elements in the matrices, n is the number of rows and m is the number of columns.

This type of behavior in the matrices are found in the real-world, such as social networks, biological network, topological network, and so on. The cost of storing zeros in such cases becomes expensive and redundant to an extent, as they do not contribute to the analysis.

Therefore, to store large sparse matrices, in this paper, we are using existing structures such as CSR (Gopal Krishna et al., 2021)(Snay, 1976a), and CBT (Nelson et al., 2017).

2.5 Parallel Compression of Graph Structures

The Compressed Sparse Row (CSR) (Tinney and Walker, 1967) data structure is widely utilized for graph representation. CSR involves compressing each row of the graph into two arrays for each node, allowing efficient packing of all the necessary information into a single array for fast traversal of the data structure. Figure 3.2 shows the CSR representation of the graph shown in Table 7.1. While CSR has the disadvantage of being a static storage format that can require shifting the entire edge array when adding an edge, its cache-friendliness inspired the development of Packed Compressed Sparse Row ($PCSR$) (Winter et al., 2017). $PCSR$ substitutes the edge array in CSR with a Packed Memory Array (PMA) (Itai et al., 1981), (Bender et al., 2000), which offers an (amortized) $O(log_2|E|)$ update cost and asymptotically optimal range queries. In this paper, we do not take the packed CSR route to compress the given graph.

Calculation of the prefix sum is one of the crucial steps in the construction of CSR for the calculation of the degree array. The prefix sum operation (Blelloch, 1990; Wheatman and Xu, 2021) takes an array A as input of length $n$and outputs an array A' where $\forall i \in \{0, 1, ...n-1\}$,

$$A'[i] = \sum_{j=0}^{i} A[i]$$

There have been parallel in-place algorithms for finding prefix sum (Blelloch, 1990) that take $O(n)$ work and $O(logn)$ in time. Because of the high dependency on computing parallel degree arrays in CSR, there are many challenges involved. Parallel Packed Compressed Sparse Row ($PPCSR$) (Wheatman and Xu, 2021), designs and analyzes a parallel PMA approach and compares it to other similar approaches (Shun and Blelloch, 2013; Dhulipala et al., 2019; Shun et al., 2015).

Chapter 3

Compression Techniques for Time-Evolving Graphs

A graph $G = (V, E)$ is an ordered tuple where V is a non-empty set of elements called vertices (nodes), and E is a set of an unordered pair of elements called links (edges), and a time-evolving graph is a change in the states of the edges over time. Extremely large graphs are such graphs that do not fit into the main memory. One way to address the issue is to compress the data for storage. The challenge with compressing data is to allow for queries on the compressed data itself at the time of computation without incurring overhead storage costs. Our previous work on Compressed Binary Trees (CBT), which was shown to be efficient both in time and space, compresses each node and its neighbors (termed as row-by-row compression). This chapter first provides encoding to store the arrays in the Compressed Sparse Row (CSR) data structure and extends the encoding to store time-evolving graphs in the form of CSR. The encoding also enables accessing a node without decompressing the entire structure, meaning the data structure is queryable. We have performed an extensive evaluation of our structures on synthetic and real networks. Our evaluations include time/space comparison with both time-evolving compressed binary tree and ck^d data structures, including the querying times.

3.1 Introduction

Graphs can be used to represent real-world data from a wide variety of domains. The relationships among the data are captured by the characteristics of the graph. For most real-world data, the relationships change over time. This results in the graph evolving from its initial state to the current one. A graph $G = (V, E)$ is represented by a set of vertices V and a set of edges E. For real-world data, the graph G evolves with time and can be statically represented using a series of graphs $G_t = (V_t, E_t)$ where the time t indicates an instant that is spread over a certain interval.

Therefore, a time-evolving graph can be defined as a graph that changes or evolves over time. Consider, as an example, pages on Wikipedia. Each page evolves over time with the addition and deletion of content. The current state of the page is the one that contains the content of the page at present. However, all the edit information is also saved for the page. Using the edit information, the state of the page at previous instants of time can be checked. Hence, having such information preserves the integrity of the page while being open to editing. Now, the information for the Wikipedia pages with the time-evolving data can be represented and stored as graphs. Storing such information is useful for performing various kinds of analyses. For example, one might want to know what changes have occurred to a document from the beginning to the current state.

Another related query can be what changes occurred to a document within a certain interval of time; this would be specifically interesting to study if the document represents some current socio-economic or political events. Also relevant would be to know the number of changes that occur to the documents from one time to another.

Time-evolving graphs represent data from different domains, such as social networks and communication networks. Various analyses can be performed on such data based on the availability of the same over time. For example, such graphs can be used to perform descriptive, diagnostic, predictive, and prescriptive analytics, among others. Therefore, to execute such operations on time-evolving graphs, the data must be stored. Generally, graphs are stored in one of the three different representations: adjacency matrix, adjacency list, and edge list. In the adjacency matrix, the graph is represented as a matrix of n^2 elements, where $|V| = n$; edges are represented using 1's and lack thereof as 0's. For a representation of an adjacency list, for each node $v \in V$, a list of adjacent nodes is stored. Finally, for an edge list representation, all edges are stored in a pair format (v_i, v_j) where an edge exists from v_i to v_j; the number of entries in the edge list is $|E|$. However, most real-world graphs are very large in size. Hence, the memory requirements for storing the data are significant. For example, if we consider a time-evolving graph, like Wiki-edits and Yahoo Netflow, the data sizes in the edge list format are 5.7 GB and 19 GB, respectively. With such sizes, the data might not fit in the main memory for analysis. Therefore, to store the data for time-evolving graphs and perform computation on the same, the data must be compressed.

In this chapter, we propose techniques to perform compression on time-evolving graphs. There are normally two methods for compressing the adjacency information for a graph: one is to consider the entire graph together, and the other considers portions of the graph at a time.

For our methods, we exploit row-by-row compression for node data separately. Specifically, we utilize two combinations of data structures to store the time and adjacency information. In the first one, the time tree provides information regarding the instants of time the graph evolved; for each node, there is an additional tree to store the adjacency information for each instant in the time tree, which is CBT. Rather than storing the entire adjacency information for the nodes, a differential approach is leveraged to reduce the memory requirements. In the second approach, for each time frame, the edges are stored in an unsigned bit array and, similar to the CSR-CSR compression, the latter information is stored in a differential approach to save the memory which is CSR. Our contributions also show that depending on the characteristics of the graph being compressed, using a combination of techniques rather than a single method for the data structures yields better compression.

The remainder of the chapter is organized as follows. We propose the improvements performed on the already existing Compressed Binary Trees in Section 3.3.1. We propose our methods for compressing and storing time-evolving graphs in Section 3.3. In Section 3.4, we examine the various algorithms that provide efficient compression for time-evolving graphs. We report the empirical results and analysis in Section 3.5.1 and the summary in Section 3.6.

3.2 Simple data structures for storing the graph information

One can select from various data structures to store the adjacency information of graphs in the memory. Each data structure has different storage requirements. Additionally, certain operations on graphs require accessing the stored data using one of the available data structures. However, choosing a memory-efficient data structure may result in worse time complexity for accessing the necessary data for computations. Thus, selecting the appropriate data structure for storing the adjacency information of graphs involves a trade-off between memory requirements and access time complexity. This section analyzes the space required by different data structures and their time complexity for performing common graph operations. It is worth noting that throughout the chapter, boolean data consisting of a single bit is used for calculations, and it does not refer to the data type available in programming languages.

3.2.1 Adjacency Matrix

For a graph, $G = (V, E)$ with $|V| = n$, an adjacency matrix is a way to represent a graph as a square matrix, where each row and column correspond to a vertex of the graph. The value in each element of the matrix indicates whether there is an edge between the vertices corresponding to the row and column. Specifically, the value is 1 if there is an edge between the vertices and 0 if there is no edge.

For an undirected graph, the matrix is symmetric about the main diagonal, meaning the element at position (i, j) is the same as the element at position (j, i). However, for a directed graph, the matrix need not be symmetric.

The adjacency matrix is a simple and intuitive way to represent a graph, and it allows for easy implementation of certain algorithms, such as breadth-first search and Dijkstra's algorithm for finding the shortest path. Additionally, certain operations on graphs, such as determining if a graph is connected or calculating its diameter, can be performed efficiently using an adjacency matrix.

However, one downside of the adjacency matrix is that it can be memory-intensive for large graphs, as the matrix requires $O(n^2)$ memory for a graph with n vertices.

Overall, the adjacency matrix is a useful representation of a graph that has certain advantages and disadvantages depending on the application.

3.2.2 Adjacency List

The adjacency list is a common data structure representing the relationships between vertices in a graph. In this data structure, each vertex in the graph is associated with a list that contains all its neighboring vertices. This list represents the adjacency information for that particular vertex. Essentially, the adjacency list captures the connections or edges of a graph by storing them as linked lists or arrays. Therefore, the memory required to store the adjacency list is $n + m + log_2 64$, where n is the number of vertices, m is the number of edges, and $log_2 64$ refers to space required to store the pointers to the list.

One of the main advantages of using an adjacency list over an adjacency matrix is its efficient use of memory. In the adjacency matrix representation, the adjacency information is required to store a two-dimensional matrix of size $n \times n$ (where n is the number of vertices). This means that even if the graph has only a few edges, the matrix would still consume a significant amount of memory. On the other hand, the adjacency list representation only requires memory proportional to the number of vertices and edges in the graph, making it more space-efficient.

Another advantage of the adjacency list is its flexibility and efficiency when dealing with sparse graphs. Sparse graphs are those that have relatively few edges compared to the total number of possible edges. In such cases, the adjacency list performs better than the adjacency matrix because it avoids the need to allocate space for all possible edges. Instead, it only stores the edges that actually exist, resulting in improved efficiency for operations such as finding neighbors or traversing the graph.

However, the adjacency list is no short of its drawbacks. The drawbacks of the adjacency list data structure include its relatively slower performance for dense graphs and certain operations, such as checking the existence of an edge. Additionally, it requires more effort to modify the graph compared to the adjacency matrix. These factors should be considered when deciding whether to use an adjacency list or another graph representation.

In summary, the adjacency list is a data structure representing the connections between vertices in a graph by associating each vertex with a list of its neighboring vertices. It offers advantages over the adjacency matrix regarding memory efficiency and performance for sparse graphs. By leveraging these benefits, the adjacency list is widely used in graph algorithms and applications where memory usage and efficient traversal are crucial factors.

3.2.3 Edge List

The edge list is a straightforward data structure that represents a graph by listing all the edges in the graph. It consists of a list of tuples or objects, where each tuple represents an edge and contains the pair of vertices that the edge connects. Therefore the total space required to store an edge list is $2 \times m$.

One of the main advantages of the edge list data structure is its simplicity. It is easy to understand and implement since it directly represents the edges in the graph without any additional data or pointers. This simplicity makes it a suitable choice for small or simple graphs where quick and straightforward access to the edges is required.

However, the edge list has a few limitations. One drawback is its inefficiency for certain operations. Finding the neighbors of a vertex or checking the existence of an edge may require iterating through the entire edge list, resulting in a time complexity of $O(m)$. In comparison, other data structures like the adjacency list or adjacency matrix can perform these operations more efficiently.

Another disadvantage of the edge list is its space complexity. The space required to store an edge list is proportional to the number of edges (m) in the graph. This means that even for graphs with a small number of vertices but a large number of edges, the edge list can consume a significant amount of memory. In contrast, other data structures like the adjacency matrix or adjacency list have space complexities that depend on the number of vertices and edges, allowing for more efficient memory usage in certain cases.

In summary, the edge list data structure provides a simple representation of a graph by listing all the edges. It is easy to understand and implement but may be inefficient for certain operations and can consume more memory compared to other graph representations. The choice of using an edge list depends on the specific requirements of the graph and the operations to be performed.

3.3 Graph Compression

In this chapter, we represent the time-evolving graphs based on the neighbors of each node over time. This requires two data structures; the first one stores the time information, which can be represented as a time array, and the second one stores the neighbors of the node at each of the instants given in the time array. Now, the time array can be thought of as a stream of 0 and 1 bits: a 0 indicating no change from the previous instant of time and a 1 indicating changes in the neighborhood of the node from the previous time stamp. Since the time instants taken into account are finite and relatively small, the size of the time array could be in the range of 10,000 for an example graph. For the same graph, the size of the node array, which would contain the neighborhood information for the specific node over 10,000-time instants, could be 1,000,000,000 elements.

The density of the time array and the node array can be different. Given that the time array and the node array are bit arrays, these can be compressed for storage using different methods. In this chapter, we propose compressing the binary arrays using one of two techniques: a) CBT and b) CSR. Now, depending on the size and the sparsity of the graphs, the sizes of the compressed structure vary.

3.3.1 Compressed Binary Tree

Compressed Binary Tree or CBT, was first developed by Nelson et. al., (Nelson et al., 2017) for a static graph, and later the concept of compressing the graph was extended to a time-evolving version by Nelson et. al., (Nelson et al., 2018).

Direct construction

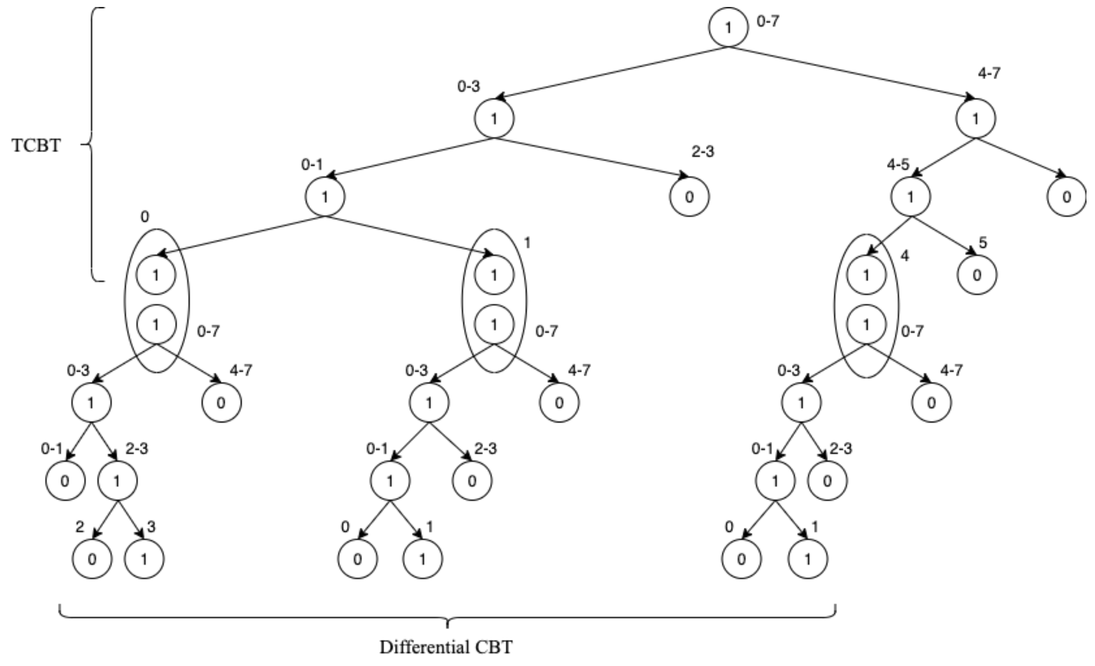
If we are given the sorted input graph all at once (as discussed above), we can construct each tree's preorder bitstring directly. This way, we can achieve an initial compression with a time complexity of $O(c \times \log(n))$. This process is the same as in Algorithm (Nelson et al., 2018).

Algorithm 1: Construction of CBT'

```
   Input: The graph as a sorted list of triplets $(u, v, t)$
   Output: The compressed graph as a bitstring
1  begin
2      $length \leftarrow 0$;
3      $BitString$ finalBitString;
4      foreach node $u$, do
5          // Fetch the time frames $t_i$ of row $u$, if row $u$ is changed from $t_{i-1}$.
6          beg = 0;
7          end = list.size();
8          $BitString$ temp = preorderTraversal(list($t$), beg, end);
9          finalBitString.AppendBitString(temp);
10         length + = temp.GetLength()
11         beg = 0;
12         foreach time $t$ in $u$, do
13             // Fetch all the edges $(u, v) \in t$ to a list.
14             end = beg + $list_t$.size();
15             temp = preorderTraversal($list_t(v)$, beg, end);
16             finalBitString.AppendBitString(temp);
17             length + = temp.GetLength()
18             beg = end;
19     return finalBitString;
```

In Algorithm 1, for each node, we maintain two CBT′ trees, one CBT′ for storing the time frames where the change occurred in the neighborhood of the node.

The second CBT is the differential CBT as shown in Figure 3.1. We first iterate through each node u (line 4) to collect all the time frames the node u



BitString CBT': TCBT = 11111011100 + Differential CBT = 1101010 1110100 1110100

Figure 3.1: CBT' for row A_0 for all the time frame.

changed and this is placed in a list (line 5). Using this list, we can directly construct the CBT implicitly with those elements and retrieve the bit string of the corresponding preorder traversal of the CBT (line 10). Later, we append the bitstring obtained in line 10 into the finalBitString variable (line 11).

In lines 14-20, we construct the differential CBT's. We will start at time frame 0 and construct the CBT. Again, we will use the implicit construction and obtain the preorder traversal bit string of the binary tree. The second iteration (for time frame 1), for the same node, we will supply a list of edges which have changed (added or removed). Please visit Figure 3.1 and the corresponding explanation. For the new set of edges (added or removed), we construct a new CBT (this is the differential CBT). At the end of each iteration, we append the bit string corresponding to the preorder traversal of the CBT that is constructed implicitly.

Note that the finalBitsString contains the bit string of the preorder traversals of all differential CBT's for a particular node along with the for the time dimension.

Improved Space Saving Structure (CBT'_N)

In this paper, we also designed a variation of the CBT' structure. The $TCBT$ structure informs us in which frame a node has changes in its neighborhood. In a new variation denoted CBT'_N, we get rid (only for storage purposes) of the $TCBT$ structure, instead we keep track of the nodes that have changes in its neighborhood in each time frame. Please note that we will use the CBT' (that has uses $TCBT$) structure for querying and the structure CBT'_N described below for storing. We also note that it is easy to convert CBT'_N structure into the CBT'. We will do this after we read in the CBT'_N into the main memory.

In each time frame, we will store all nodes that have changes made to them (edges added or removed). This set of nodes forms a bit string. We will store this bit string as a CBT structure. We can retrieve the nodes from this CBT structure (actually from the preorder traversal of the CBT structure).

We have shown through experimentation (see Table 3) that this structure CBT'_N has a lower space requirement in comparison to CBT'. The new structure CBT'_N while occupies less space, it does not allow for quicker operations. For example, to execute the edge (u, v) existence query in a given time interval, we need to check every frame in that time interval to determine if node u is in the data structure $nCBT$. To reduce the cost of querying on this new structure CBT'_N, we read the structure from the file and convert it to CBT' structure for faster querying and streaming.

Experimental Result

While examining the compression results, we refer to Table 3.1. The first three columns describe the dataset with its name, size as a raw text file, and size of the text file compressed with gzip. Note that these text file sizes are after we have performed our preprocessing on the graph. The next column gives the percentage of memory saved by CBT'_N compared to the ck^d-tree, for easy reading.

Table 3.1: Shows a comparison between the sizes of the raw graph, CBT', CBT'_N, and ck^d-tree compression. The *.txt* files are the preprocessed triplet representations of the graph, as described earlier. The *.txt.gz* are those files after being gzipped. This data is important because we have implemented a method of compressing the graph directly from the smaller gzip files.

	.txt	.txt.gz	$\frac{CBT'_N}{ck^d}$	CBT'_N	CBT'	ck^d-tree
I-Comm.Net*	271.6MB	51.0MB	50%	15.5MB	15.19MB	31MB
I-Powerlaw*	546.9MB	132.2MB	50.1%	68.9MB	70.37MB	138MB
I-Yahoo-Netflow	19.3GB	3.9GB	40.9%	1.24GB	2.79GB	2.1GB
G-Flickr-Days	860.0MB	130.5MB	25.5%	63.32MB	70.38MB	84.9MB
P-Wiki-Edit	5.7GB	1.6GB	10.3%	1.31GB	1.31GB	1.46GB

3.3.2 Compressed Sparse Row

The Compressed Sparse Row (CSR) format is a widely used method for representing sparse matrices, where most elements are zero. It was first introduced in Richard, Snay (Snay, 1976a). The CSR format stores a sparse matrix using three arrays: the values array contains the non-zero elements, the column indices array (jA) indicates which columns the non-zero elements belong to, and the row pointers array or the degree array (iA) points to the starting position of each row in the values and column indices arrays. Since, we focus on the undirected graph, we ignore the value array associated to the structure.

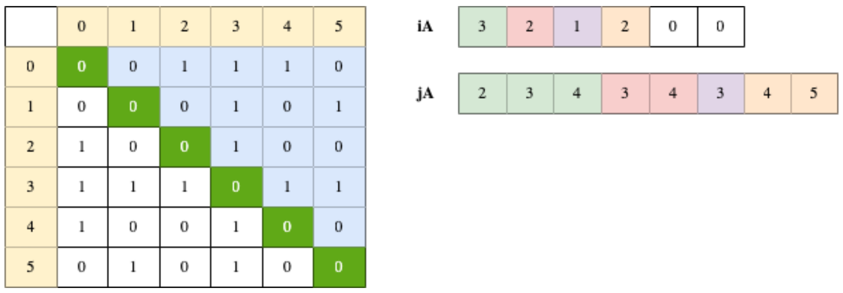


Figure 3.2: An example of CSR representation of a boolean matrix.

Since, these numbers have to be stored as integers, each number is bound to consume either 32-bits or 64-bits depending on the configuration. To avoid the wastage of bits to store these number, we present an integer encoding mechanism called BitPacking, which efficiently stores numbers one next to the other in contiguous array locations.

BitPacking

The term bit-packing works on the number of bits required to represent each number. For a given array of unsigned integers, represent each number of the array in bits and store them as an unsigned bit array. For example, consider an array of unsigned integers 1, 3, 5, 10, 16, and 26. The maximum number of bits required to store each integer is the maximum of the log of the maximum element in the array. An array location can store a maximum of 32 or 64 bits depending upon the system, and all the bits are stored in a little-endian format. Table 3.2 shows all the numbers stored above in a single unsigned bit array location.

Table 3.2: The single-bit array needed to represent all integers.

Degree Array	1	3	5	10	16	26	
BitPacked	00001	00011	00101	01010	10000	11010	00

If the entire number does not fit into an array location, a part of the number

Table 3.3: Shows the partial packing of the leftover bits in the first array location, and the remaining bits in the next.

Degree Array	1	3	5	10	16	26	30
BitPacked	00001	00011	00101	01010	10000	11010	11
	110	00000000000000000000000000000					

can be stored in one array location, and the rest of the number can be stored in the starting bits of the following memory location. This ensures that there are no unoccupied bits in the bit array. Table 3.3 shows the bits carried over for a 32-bit integer.

Algorithm 2 explains the working of the bitPacking method. The algorithm takes an unsigned integer array, the number of elements in the array, and the number of bits required to represent each number in the array. The variable arraySize indicates the number of array locations needed to store the unsigned bits of all the numbers in the array. Line 7 indicates the start of converting the unsigned integer to the bit representation; m indicates the number of unsigned integers that an array location can accommodate. Lines 9 through 11 convert the unsigned integers to unsigned bits and store them in the array location k. The remaining bits in the array location k are filled by the most significant bits of the next number, as shown in lines 13 through 20.

3.4 Combining CBT and CSR

For every input of the time-evolving graphs G, the input is divided as an ordered triplet (u, v, T_τ), where u and v are the nodes that form an edge at time T_τ. If the edge appears again later in another time frame $T_{\tau+i}$, the edge is considered to be deactivated in the time frame. For the CSR-CSR and CBT-CSR combinations, we are assuming that the datasets are sorted with respect to the time frames

Algorithm 2: Algorithm for bitPacking

```
Input: An unsigned integer array (uArray), number of elements
       (numElements), and the number of bits (numBits) required to
       convert
Output: The converted bit array.
1  begin
2      totalBits = 64;
3      balance = 0;
4      arraySize = $\frac{numberofElements}{numberofBits} * totalBits$;
5      Initialize an unsigned bit array (bArray);
6      k = 0;
7      for $index = 0$ to $(index < numOfElements)$ do
8          m = $availBits/numBits$;
9          for $i = 0$ to $(i < m$ && $(index + i) < numElements)$ do
10             bArray[k] | = $(uArray[index + i] <<$
                 $((i * numBits) + balance)))$;
11             i++;
12         index + = $m$;
13         if $index < numElements$ then
14             remBits = $availBits\%numBits$;
15             if $((remBits > 0)\&\&(m < numElements))$ then
16                 bArray[k] | = $(uArray[index] << (totalBits - remBits))$;
17                 availBits = $totalBits - (numBits - remBits)$;
18                 k++;
19                 bArray[k] | = $(uArray[index] >> remBits)$;
20                 balance = $(numBits - remBits)$;
21                 index++;
22             else
23                 k++;
24                 balance = 0;
25                 availBits = totalBits;
26     return $bArray$;
```

and then sorted by node numbers for each time frame. For the CSR-CBT and CBT-CBT combinations, the datasets are first sorted with respect to the source node and then sorted with respect to the time frame for each source node.

Algorithm 3: Algorithm for Compressed Sparse Row at time T_0

```
  Input: Unsigned integer array of contacted nodes (v), unsigned integer
         array of a degree of each node (u), the maximum degree of the
         graph ($\delta$(G))
  Output: Unsigned bit array
1 begin
2     logD = log2(maxDegree) + 1;
3     logN = log2(numNodes) + 1;
4     degreeBitArray = bitPacking(degreeArray, numNodes, logD);
5     csrBitArray = bitPacking(vArray, numEdges, logN);
6     finalBitArray.append(degreeBitArray);
7     finalBitArray.append(csrBitArray);
8     return finalBitArray;
```

3.4.1 CSR-CSR

This is a novel combination. In this algorithm, we compress the graph based on the edges appearing in each time frame. For the time frame T_0, we compress the graph row-by-row in a conventional compressed sparse row format using Algorithm 3. For each row, the first array consists of each node's degree in the time frame T_0, and the second array consists of the upper triangular destination edge id v.

For the time frame, T_1 *to* T_τ, storing the edge in the conventional CSR format will cost more space as not all the edges from the original start edges change. To overcome this, we encoded all the edges using Algorithm 4. For every time frame T_i, CSR is made up of three arrays, where the first array is the unique source node u involved in the graph's changes, the second array consists of the degree

Algorithm 4: Algorithm for Compressed Sparse Row from time T_1 to T_τ

```
Input: Unsigned integer array of contacted nodes (v), unsigned integer
       array of a degree of each node (u), unsigned integer array of
       contact nodes (u)
Output: Unsigned bit array
1  begin
2      for time $t = 1$ to $t = \tau - 1$ do
3          logU = log2(maxContactNode) + 1;
4          logD = log2(maxDegree) + 1;
5          logN = log2(maxContactedNode) + 1;
6          bitUArray = bitPacking(uArray, uArray.size(), logU);
7          bitDegreeArray = bitPacking(degreeArray, uArray.size(), logD);
8          bitVArray = bitPacking(uArray, vArray.size(), logN);
9          finalBitArray.append(bitUArray);
10         finalBitArray.append(bitDegreeArray);
11         finalBitArray.append(bitVArray);
12     return finalBitArray;
```

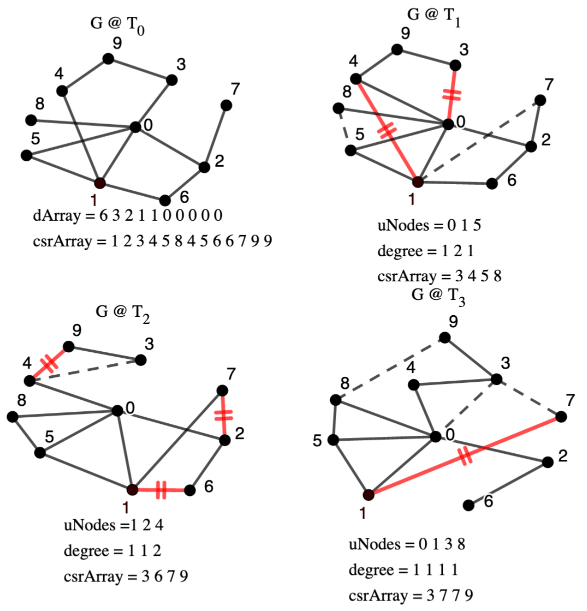
of the source nodes, and the third array consists of the destination nodes v.

This yields the time complexity of $O(\tau*(n*log(\delta)+mlog(n)))$, where τ is the number of time frames, n is the number of nodes, m is the number of contacts, and δ is the maximum degree of the graph.

Figure 3.3 shows the overall structure of the CSR-CSR compression. The dotted line in the graph at each time frame represents the edge being added and the double-crossed red line represents the edge being deleted at the time frame.

3.4.2 CBT-CSR

This is a novel combination. In this combination, we compress the first time frame T_0 using the existing CBT algorithm (Nelson et al., 2017) as shown in Figure 3.4. The input to the algorithm (Nelson et al., 2017) are the edges associated with time frame T_0. For the time frames T_1 *to* T_τ we follow the method used in the



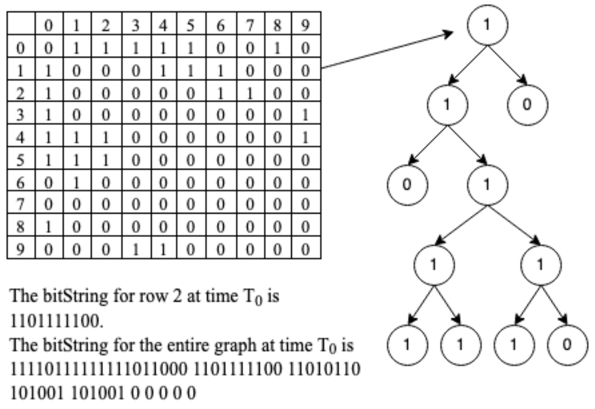
CSR as unsinged char: 01101100010010001000
10000100110000101010001001010111000010011000010101 0 000100101
100010100 110000101010001 100010001 100100010 0010111000010101000
0000100011000001 1111 1100111011101001

Figure 3.3: The structure of the time-evolving CSR

CSR-CSR compression, as shown in Figure 7.4. This yields the time-complexity of $O(\tau * (d(v)log(\delta) + mlog(n)))$.

3.4.3 CBT-CBT

In this combination, we followed the algorithm mentioned in (Nelson et al., 2018). The input to this algorithm is first sorted based on the source node u, and for each source node, the data is sorted based on the time T_i. With this type of input comes two arrays for each node, the first being the time frames at which the node has an edge, followed by all the destination nodes v for each time T_i. Therefore, each node's total number of trees will be the number of time frames for the node u and one tree to represent all the time frames. This yields the time-complexity



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

The bitString for row 2 at time T_0 is
1101111100.
The bitString for the entire graph at time T_0 is
1111011111111011000 1101111100 11010110
101001 101001 0 0 0 0 0

Figure 3.4: The structure of the time-evolving CBT at time frame T_1

of $O(\tau * (d(\tau)log(\delta) + mlog(n)))$, where $d(\tau)$ represents the degree of each time frame.

3.4.4 CSR-CBT

This is a novel combination. In this combination, we follow the same input type as the CBT-CBT combination, but here we first compress the time array using bit-packing algorithm 2, and to compress the destination edges for each time frame of the source node u, we follow the CBT algorithm (Nelson et al., 2017). This yields the time-complexity of $O(\tau * (d(v)log(\delta) + mlog(n)))$, where $d(v)$ denotes the degree of each node v.

3.5 Experimental Results

For our analyses, we have the results of compression size and the time taken to compress from all the combinations using the datasets mentioned in Table 3.4, which is compared with the results of $ck^d - tree$ (Caro et al., 2016) and CBT (Nelson et al., 2018) as shown in Table 3.5.

If the edges in the graph exist from time $[t_i, t_j)$, then such graphs are called

Table 3.4: The graph datasets, including the type, number of nodes, number of edges, time frames, and the size of the input file both in .txt and gzip format.

Graphs	Type	Nodes	Edges	Contacts	Time Frames
CommNet	Interval	10000	15940743	19061571	10001
PowerLaw	Interval	1000000	31979927	32280816	1001
Flickr-Days	Incremental	2585570	33140018	33140018	135
Wiki-edits	Point	21504191	561754369	266769613	134075025
Yahoo Netflow	Interval	32904819	122075170	1123508740	58735

Table 3.5: The compression size and the time taken to compress each dataset. Please note that ck^d does not allow streaming operations

Graphs	.txt	.txt.gz	CSR-CSR		CSR-CBT	
CommNet	271.6 M	51 M	34 M	10.25 s	16 M	56.19 s
PowerLaw	546.9 M	132 M	80 M	18.94 s	80 M	162.23 s
Flickr-Days	860 M	130 M	107 M	34.16 s	91 M	208.39 s
Wiki-edits	5.7 G	1.8 G	2.0 G	1158 s	1.8 G	2042.88 s
Yahoo Netflow	19 G	4.9 G	4.3 G	1372 s	3.2 G	1770.71 s

Graphs	.txt.gz	CBT-CSR		CBT-CBT		CkD	
CommNet	51 M	16 M	55.80 s	15.9 M	65.5 s	30 M	119 s
PowerLaw	132 M	70 M	141.21 s	73.80 M	149 s	128 M	254 s
Flickr-Days	130 M	82 M	120.015 s	73.8 M	179 s	89 M	235 s
Wiki-edits	1.8 G	2.0 G	1126.85 s	1.4 G	3081s	1.2 G	2059 s
Yahoo Netflow	4.9 G	4.2 G	1874.95 s	2.99 G	3506 s	2.5 G	5471 s

interval graphs. If the edges in the graph appear once and live till the last time frame, such graphs are referred to as incremental graphs. If the edges appear for a single time frame, then such graphs are referred to as point graphs.

The CommNet graph and the PowerLaw graphs are synthetically generated datasets based on the data available from (Caro et al., 2016). CommNet graph simulates short communication between random vertices. PowerLaw graph simulates the power-law degree distribution in the graph.

Flickr dataset is an incremental graph (Fli, 03/2020). This graph represents the user interaction derived from the Flickr social network for a span of days from 11-02-2006 to 05-18-2007.

Wiki-edits is a bipartite point graph (Wik, 03/2020). This graph shows when the user edited an article in Wikipedia. The time is stored in seconds since the creation of Wikipedia.

The last dataset for our analysis is a Yahoo-Netflow graph (Yah, 03/2020). This graph is an interval graph, where the data are the interaction between the users and the Yahoo server. The time is measured in seconds and the first occurrence of the data was on 04-29-2008.

All the experiments were run on an Intel(R) Xeon(R) CPU E5520 @ 2.27GHz (16 Cores) with 64 GB of RAM, and the programs are written in GNU C/C++.

The source code for this work is available to download at (git, 02/2021).

3.5.1 Compression Results

Table 3.5 shows the compression results for the dataset in Table 3.4 over all the combinations and $ck^d - trees$. We have used compression size, time taken to compress each dataset, and querying times as metrics to evaluate.

Table 3.5 clearly shows the space and time tradeoff between the compression results of CSR-CSR and CBT-CBT. While the CBT-CBT consumes 30% less space compared to CSR-CSR, CSR-CSR consumes around 40% less time for the Yahoo-Netflow graph(Yah, 03/2020). The combination of CBT-CSR and CSR-CBT has shown similar or better results with datasets with fewer or no changes in both compression size and time taken to compress.

3.5.2 Querying Results

For 1000 randomly chosen vertices,

- Neighbor Query: What are the neighbors that exist at time T_i.

Table 3.6: the average time needed to query a node to fetch all the neighbors at a given time T_i and the time interval T_i through T_j

Graphs	CSR_Ti (ms)	CBT_Ti (ms)	CKD_Ti (ms)	CSR_Ti_Tj (ms)	CBT_Ti_Tj (ms)	CKD_Ti_Tj (ms)
CommNet	0.78 ∓ 0.005	1.33 ∓ 1.44	48.89 ∓ 11.56	0.93 ∓ 0.049	1.43 ∓ 0.47	64.46 ∓ 0.43
PowerLaw	2.07 ∓ 0.006	2.99 ∓ 0.55	374.23 ∓ 50.72	2.10 ∓ 0.012	5.70 ∓ 1.05	374.64 ∓ 50.66
Flickr-Days	1.31 ∓ 0.02	11.29 ∓ 8.52	35.34 ∓ 10.39	2.08 ∓ 0.31	38.49 ∓ 8.34	45.22 ∓ 5.78
Wiki-edits	0.40 ∓ 0.007	1.24 ∓ 1.911	3.0 ∓ 3.0	0.403 ∓ 0.001	1.42 ∓ 2.12	4.39 ∓ 0.72
Yahoo Netflow	2.19 ∓ 0.45	43.13 ∓ 0.263	231.9 ∓ 82.1	1.51 ∓ 0.06	51.21 ∓ 4.67	254 ∓ 92.06

Table 3.7: the average time needed to query an edge exists between two nodes at a given time T_i and the time interval T_i through T_j

Graphs	CSR_Ti (ms)	CBT_Ti (ms)	CKD_Ti (ms)	CSR_Ti_Tj (ms)	CBT_Ti_Tj (ms)	CKD_Ti_Tj (ms)
CommNet	0.78 ∓ 0.001	0.39 ∓ 0.66	49.6 ∓ 3.4	0.82 ∓ 0.002	0.39 ∓ 0.55	49.7 ∓ 0.24
PowerLaw	2.06 ∓ 0.011	0.64 ∓ 0.12	216.0 ∓ 5.3	2.08 ∓ 0.06	1.6 ∓ 0.03	226.13 ∓ 14.38
Flickr-Days	1.31 ∓ 0.013	4.23 ∓ 2.81	35.2 ∓ 1.2	2.21 ∓ 0.2	5.44 ∓ 10.11	37.2 ∓ 2.3
Wiki-edits	0.39 ∓ 0.08	1.15 ∓ 0.18	2.62 ∓ 1.7	0.39 ∓ 0.008	1.15 ∓ 0.19	2.98 ∓ 0.25
Yahoo Netflow	1.38 ∓ 0.014	30.32 ∓ 2.36	211.8 ∓ 89.0	1.52 ∓ 0.042	31.2 ∓ 5.37	212.32 ∓ 71

- Neighbor Query: What are all the possible neighbors of a node between time interval of T_i through T_j.
- Edge Existence: Does an edge exist at time T_i.
- Edge Existence: Does an edge exist between the time interval of T_i through T_j.

From Tables 3.6 and 3.7, we can infer that the CSR takes the least amount of time to query a random node both for edge existence and to fetch all the neighbors.

3.6 Summary

In this chapter, we focused on creating a time-evolving structure to support one of the most utilized data structures which is Compressed Sparse Row. We also introduced a mechanism to encode a fixed length integers to save memory. We also introduced combination of compression techniques to yield the space and time trade-off seen in CSR and CBT.

Chapter 4

Compressed Structure for Representing Tensors

Tensors, their representations, and computations are gaining popularity with an increasing number of machine learning algorithms for applications. In this chapter, we show several techniques for storing compressed sparse tensors. One of our approaches using compressed binary trees (CBTs) shows that the large sparse tensors occupy nearly 1/3 of the space occupied by the storage technique used in *tsparse*. We show how tensor multiplication can be performed *directly on our proposed structure* thereby making it space efficient – allowing us to perform tensor multiplication on extremely large tensors on GPU computing platforms. Our results show that the smaller the block size (up to a point), the larger the size of the representation, and the larger the level of parallelism, and vice versa.

4.1 Introduction

A tensor is a n-dimensional array – or a multidimensional array – typically viewed as a generalization of a matrix into n-dimensions. In the simplest cases, a 1-dimensional tensor is referred to as a vector or an array, whereas a 2-dimensional tensor is a matrix. The power of a tensor lies in its ability to concisely capture complex relationships in multidimensional data. Such is the case in physics, quantum chemistry, and more recently in machine learning and graph analytics. For example, any social media platform where a user can tag another user will produce 3-dimensional data that record user-item-tag data (Smith et al., 2015).

In practice, these tensors can be extremely large in nature and often tend to push the limits of storage and computational power needed to perform analysis. High Performance Computing (HPC) comes into play to help parallelize the resource-heavy operations that can be performed using tensors. Deep learning data are represented using tensors and are often used in the model training phase. Various verticals of research from computational physics to quantum chemistry as well use HPC to perform operations on tensor data specific to their field.

This approach of using HPC raises another problem, being able to fit these extremely large datasets into our desired memory for computation. General-purpose graphic processing units (GPGPUs), being the most popular and accessible HPC accelerator hardware, present a large set of challenges as well. Global memory in such architectures is very limited and is typically smaller than what is needed to store a large tensor dataset. Thus, the predominate challenge blocking these datasets and arranges the computation so that the necessary subset of the data, or working dataset, is in the global when it is needed.

The working dataset can be swapped between the host's memory and the

GPGPUs global every so often. However, this is a very costly operation that often becomes a bottleneck because of the bandwidth limitations of these transfers. One solution to overcome this issue and make maximum use of the bandwidth is to compress the data on the host before transferring the working dataset to the device and directly access the desired data on the compressed structure without having to decompress. This enables us to bring more data with constant bandwidth, thus reducing the number of transfers that occur. Additionally, the compressed data allows us to place larger amounts of data in the space limited shared memory that is available on the streaming multiprocessors. By keeping data closer to the processor (rather than accessing the global memory) it allows to overcome memory access latency.

In this chapter, we address this problem for performing tensor computations on compressed data using GPGPUs. Our work builds Compressed Binary Trees (CBT) – a row by row adjacency matrix compression technique for storing graphs – that are used to perform computations on tensors that have been flattened to matrices. The advantage of this compression scheme is that it is lossless and provides the ability to query data without first decompressing the entire dataset. We provide a hierarchical granular approach to use this compression, tailored to fit the hardware architecture of the GPGPU.

Our contributions in this work are the following.

- We developed several compression techniques for storing sparse tensors with billions of elements. These techniques use CBT as the baseline technique. We have shown that our compression technique uses only about 30% of the space required by any other block compression technique that has been proposed in the past.

- Previously, it was shown that dividing the matrix into blocks is amenable to parallel computation, as computation on each block can be performed in parallel. We have improved this concept in several ways. The computation can now be performed on our compressed blocks. Since the size of each block is very small (due to compression), one or more blocks can be stored in a memory of a single streaming multiprocessor, thereby avoiding or reducing memory access time issues dealing with data in the global memory.

- We have provided an approach that refines parallel computation from coarse grain involving row blocks and column blocks in multiplication to finer grain that includes computing several block to block computations in parallel and also performing in parallel within computations that arise within single block to block.

- Our algorithms are such that all computations are performed directly on the compressed structure - there is no need to decompress. This avoids the crucial step in previous algorithms which uses the Expand, Sort, and Compress technique. We do not use intermediate data structures to store our partial results. All results can be stored directly in compressed structures.

- We have experimentally evaluated the performance of a tensor-tensor multiplication algorithm that takes into account compressed structures of various block sizes.

The remainder of the chapter is as follows. Section 4.2, explains various ways one can represent different modes of tensor. In section 4.3, we propose a various ways to we can store these tensors using our compression techniques. In Section 4.4, we show the compression results of different tensors. In Section 4.5, we propose a parallel algorithm for matricized tensor multiplication, and finally in Section 4.6, we summarize our work in this chapter.

4.2 Tensor Representation

One of the most common ways to store a tensor for any application is to first represent them as one large high-dimensional matrix. Once represented as a matrix, a tensor could be either split by decomposing into lower dimensions for ease of computation and perform the necessary operations on the lower-dimensional matrices. This comes with a loss of information, as matrix decomposition methods are highly approximate. However, to achieve an exact result, one must have to store the high-dimensional matrices in a compressed format without any loss of data. This chapter approaches the storage of tensors in three different ways, a) approaches to store tensors of even dimensions, b) approaches to store tensors of odd dimensions, and c) an approach to store tensors of any dimension.

4.2.1 Even-Mode Tensor

Figures 4.1, 4.2 shows two ways to compress the even mode tensors. For the figure 4.1, the process first starts by compressing the outermost dimension A, which denotes all non-zero locations where the corresponding A_i are present. Then, the corresponding A_i, the matrix of the following dimensions. The figure 4.2, shows the second approach for compressing an even-mode tensor. In this approach, we first start by compressing the nonzero matrix A, followed by compressing the further dimensions A_is in the same way as a time-evolving graph shown in (Nelson et al., 2021).

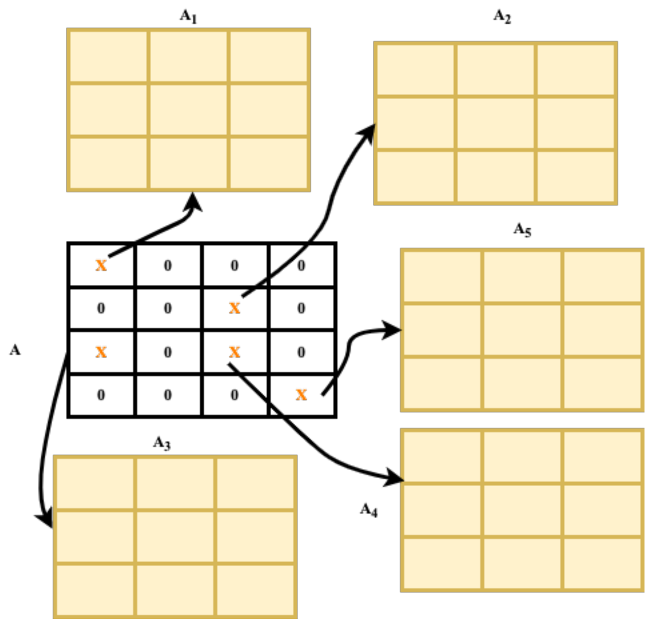


Figure 4.1: Represents a 4 mode tensor, where a dimension in compressed at time, starting from the outer-most dimension.

4.2.2 Odd-Mode Tensor

For the tensor of odd modes, we first compress all but the last dimension using the even-mode tensor approach discussed in Section 4.2.1, and every non-zero row of the last dimension is then compressed using the desired compression algorithm. Figure 4.3.a, illustrates a three-mode tensor, where the nonzero element of the matrix A consists of an array in the third dimension.

4.2.3 Unfolded Tensor

However, one of the most common methodologies for representing a tensor is to convert any mode tensor into a matrix. The reduction in dimensionality starts by merging the inner dimension with the penultimate dimension repeatedly until the total number of dimensions is equal to two, which is a matrix. This process

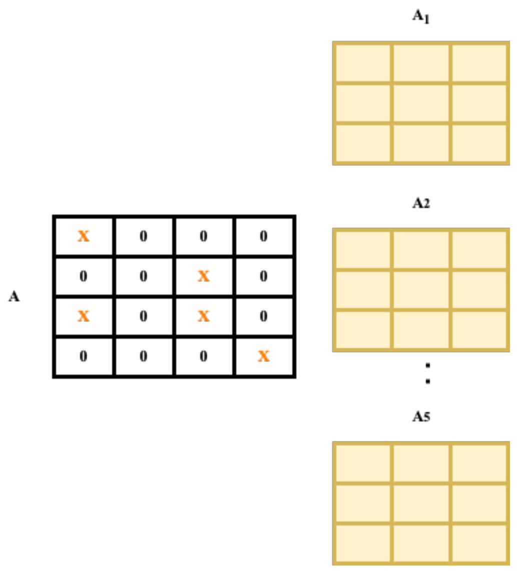


Figure 4.2: Represents a tensor of 4 mode, where the inner dimensions are considered as time-evolving graphs.

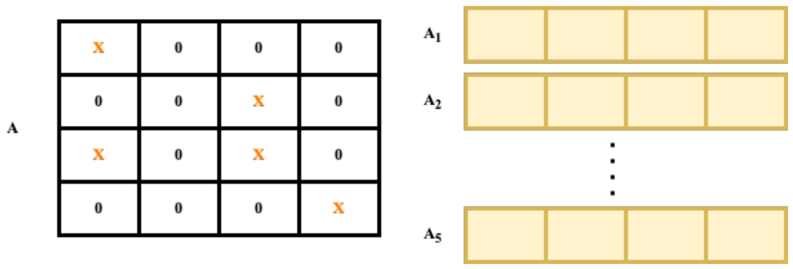


Figure 4.3: Represents a tensor of 3 mode, an array inside a matrix

represents a tensor, as a matrix is referred to as unfolding. For example, consider a three-mode tensor as shown in right of Figure 4.4 is of dimension $(I \times J \times K)$, the unfolded representation of the tensor will be represented as a matrix of dimensions $(I \times (J * K))$.

4.3 Compression Methods

This section discusses the various compression techniques that can be applied to any given tensor. The compression is based on the data structure called the

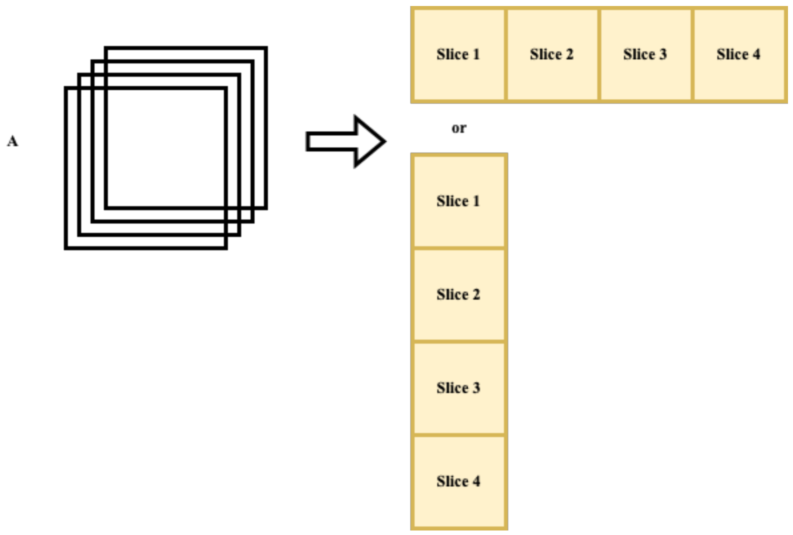


Figure 4.4: Shows the approaches where the tensors of any modes are represented

Compressed Binary Tree (CBT) (Nelson et al., 2018). Until now, compression techniques and operations on compressed structures have been limited to matrices (Nelson et al., 2018) (Nelson et al., 2019) (Krishna et al., 2021).

This chapter introduces four techniques to compress the tensors represented as shown in section 4.2.

4.3.1 CBT-Leaf_CBT

For a tensor of any mode, a $CBT - Leaf\_CBT$ will first construct a Compressed Binary Tree (CBT) (Nelson et al., 2018) for the outer most dimension of the tensor A. Consider the example as shown in the figures 4.1, and 4.3. Every nonzero entry in the outer most matrix will constitute an array or another matrix in the inner dimension. If a nonzero entry in the following dimensions are a set of arrays, then each array A_i is compressed separately using CBT (Nelson et al., 2018) (Krishna et al., 2021). If the nonzero entry in the following dimensions are a set of matrices, then each matrix A_i is compressed row-by-row using CBT

(Nelson et al., 2018) (Krishna et al., 2021).

Figure 4.5, shows the working of a $CBT-Leaf\_CBT$ on a three-mode Tensor.

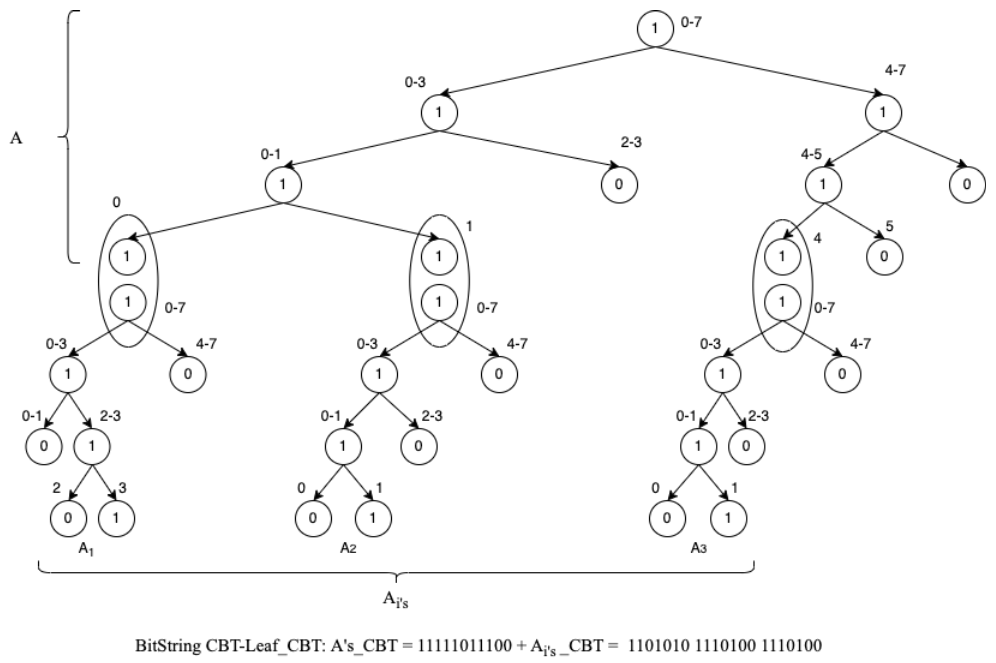

A
0-7
0-3
4-7
0-1
2-3
4-5
0
1
4
5
A1
A2
A3
A_i's
BitString CBT-Leaf_CBT: A's_CBT = 11111011100 + A_i's _CBT = 1101010 1110100 1110100

Figure 4.5: Shows the working of 3-mode compression using CBT-Leaf_CBT

4.3.2 Time-Evolving CBT

The term time-evolving comes from the common dimensionality in inner dimensions of the tensor. Here, the two-dimensional matrix A, is compressed row-by-row using the Compressed Binary Tree (CBT) (Nelson et al., 2018) similar to that in $CBT - Leaf\_CBT$ 4.3.1. Now, for the inner dimensions, all adjacent rows of $A_i's$ represented by $CBT's$ are obtained by $|A_{i+1_{row}} - A_{i_{row}}|$, for example, to compress the first row of an $A_i + 1$, we first computer $|A_{i+1_0} - A_{i_0}|$, which is the difference from the first row of A_{i+1} to the first row of A_i (Nelson et al., 2021).

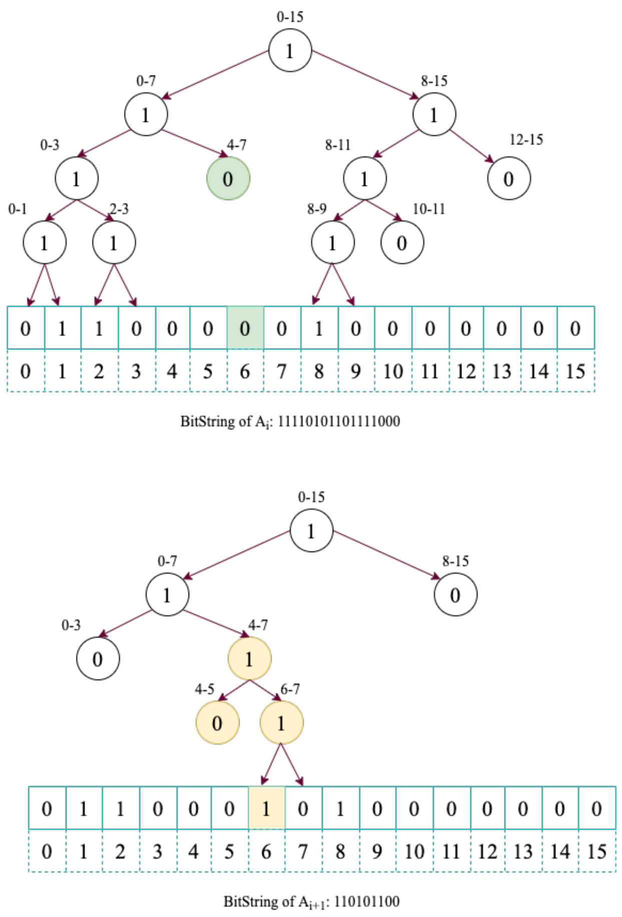

0-15
0-7
8-15
0-3
4-7
8-11
12-15
0-1
2-3
8-9
10-11
BitString of A_i: 11110101101111000
4-5
6-7
BitString of A_{i+1}: 110101100

Figure 4.6: Shows the time-evolving compression of a row A_{i+1}

Figure 4.6, shows the working of a time-evolving CBT way of storing the inner dimensions of the tensors.

4.3.3 Unfolding CBT

This is one of the most common techniques to store and represent a tensor. Since a tensor of three-mode and higher are often hard to visualize, one of efficient ways to do it is by spreading out the inner-dimensions of the tensor into the outer-dimension which is often referred to as unfolding. Unfolding is a process of reducing the dimensionality by converting the tensor into a two-dimensional matrix. The dimensionality can be reduced by considering any phase of the tensor. For a three-mode tensor of dimensionality $(I \times J \times K)$, ways to unfold are by holding one dimension constant and merging the other two dimensions. In our experiments conducted we have chosen to keep the I^{th} dimension constant and merge dimensions J and K to form a higher dimension, resulting in $I \times (J \times K)$. Once the tensor is unfolded into a matrix, the matrix is then compressed one row at a time using CBT(Nelson et al., 2018). This compression technique is referred to as Unfolded CBT or $UCBT$.

4.3.4 Block CBT

This compression technique is designed to store matricized tensor blocks that will serve to improve the temporal cache locality during the tensor operations.

This compression technique is called Block Compressed Binary Tree ($BCBT$). Nonzero blocks are continuously stored in the same way as the row-by-row nature of CBT (Nelson et al., 2018). For this compression, the rows in the block are represented in the row major order and then the block is compressed as one single CBT. The choice of using one single CBT with row major order is to take advantage of the special cases within the working of CBT, which eliminates the excess zeros and compresses the consecutive nonzeros. The size of the blocks for compression can be decided at the time of compression. The total number of blocks in the matricized tensor is defined as the ratio between the total number of matrix cells (number of rows × number of columns) to the square of block size.

$$\frac{(numRows \times numColumns)}{blockSize^2}$$

Figures 4.7, 4.8, and 4.9, shows the block wise compression of the matricized tensor using Compressed Binary Tree.

Figure 4.7, shows one of the ways in which we can covert a tensor into a matrix, in this approach, the third dimension k, is kept constant and the other two dimensions are merged together to form a high-dimension matrix. Once the tensor is unfolded, the tensor is then split into blocks of desired size, in this case the blocks are divided into the dimensions of size (2×2).

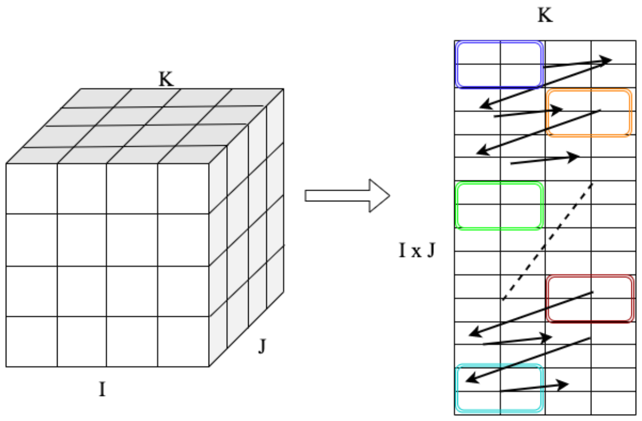


Figure 4.7: Shows the process to convert a 3-mode Tensor to a Matricized Tensor

Figure 4.8, shows the procedure to store each block in a compressed format. Before compressing the block, the block is first converted to a row of data using the row-major order. In this example, for simplicity, we have considered the block size to be (3×3).

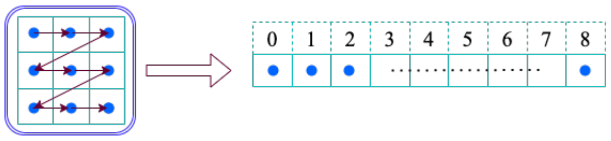


Figure 4.8: Shows the process of converting a block into a row-major ordered row of data.

Once represented as a row, the data is now compressed using CBT (Nelson et al., 2018), as shown in figure 4.9.

During the multiplication process for multiplying two blocks, we cannot directly get the element in the specific row and column since we have stored them

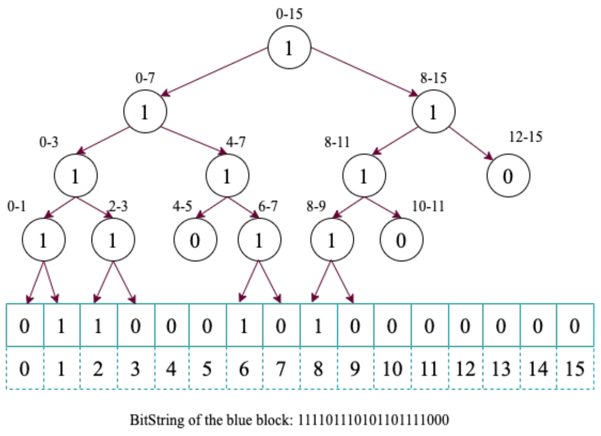


Figure 4.9: Shows the process of representing a row-major ordered data in Compressed Binary Tree format.

in compressed row major order. For this, we first query the column numbers (neighbor query) from $BCBT$ and then reverse map the row and column value from the index position. This can be done by first finding the column number (j) with a mod operation with the number of columns and then finding the row number by dividing with the number of columns and then adding j to that value.

4.4 Compression Results

The data used in this chapter are generated synthetically. Data generation is modeled after the real-world scenario of a sparse tensor. For the analyses, the data generated are in the scale of tens of million to more than a billion nonzero values in the tensor.

Since, the data are randomly generated, we performed an extensive run on a $500 \times 500 \times 400$ tensor, with the sparsity $3.39 \times E^{-7}$. We performed $CBT\_LeafCBT$, $Unfold\_CBT$, and $Block\_CBT$ on these tensors as shown in table 4.1.

It is evident from these runs that, the random tensors generated is not a the best-case data for the algorithm to work. With this information, we perform

Table 4.1: Average size and average time consumed to compress 30 different 3-mode tensors with same dimension, and same sparsity

	Average Size (MB)	Average Time (secs)
CBT_LeafCBT	3.5599 ∓ 0.0002	0.9422 ∓ 0.003
Unblock_CBT	3.5254 ∓ 0.0002	1.0442 ∓ 0.004
Block_CBT	3.5265 ∓ 0.0003	0.9834 ∓ 0.004

compression on more random datasets as shown in table 4.2.

Table 4.2 shows the various datasets that are used in the analyses and their corresponding memory sizes when we take the corresponding compression approaches. From the table, it does look like the memory required and the time taken to compress are very similar. However, depending on the use of these tensors, the appropriate compression approach must be chosen. The result for Unfold CBT on the billion-scale graph is missing due to the extremely high value of the outer dimension.

4.5 Tensor Multiplication

In this section, we explain the implementation of our matricized tensor multiplication in parallel. Operating on any data requires understanding the access pattern to take maximum advantage of parallelism. Sparse matrices, in general, are highly irregular, which relates to more memory accesses for fewer data. The GPU's efficiency depends on maximizing the amount of data read (lower communication cost) from the host to the device. To achieve maximum computation out of GPU's on the data, we are considering using block-based tensor compression discussed in Section 4.3.4.

Here, we have divided the implementation of multiplying two matrices into three algorithms. The algorithm 8 multiplies two blocks of matrices to obtain

Table 4.2: Shows the different odd and even tensor datasets along with their corresponding compression approach and the time taken to compress.

			CBT_LCBT		Unfold_CBT		Block_CBT	
Dimension	NNZ	Raw Size	Size	Time	Size	Time	Size	Time
$500 \times 500 \times 400$	32,967,574	409 MB	17.19 MB	15.85 s	17.04 MB	24.05 s	17.04 MB	16.61 s
$670 \times 670 \times 450$	100,354,540	1.3 GB	40.38 MB	38.27 s	40.11 MB	56.77 s	40.3 MB	36 s
$(2^9)^3$	134,217,728	1.7 GB	0.87 MB	8.18 s	0.81 MB	2.1 s	0.81 MB	2.60 s
$(2^{17})^3$	993,798,762	19 GB	4.22 GB	1616.57 s	–	–	2.18 GB	868 s
$400 \times 400 \times 300 \times 300$	53,251,472	864 MB	80.14 MB	38.59 s	79.94 MB	50.48 s	–	–
$589 \times 589 \times 322 \times 458$	66,407,269	1.1 GB	107.71 MB	46.91 s	111.85 MB	67.607 s	–	–
$256 \times 256 \times 128 \times 128$	252,783,285	4.1 GB	155.08 MB	121.2 s	154.07 MB	220.75 s	–	–

a partial result, algorithm 7 provides the implementation for multiplying a row block of A and a column block of B to obtain a block in C, and algorithm 7 provides the implementation for multiplying the matrix in parallel.

There are four levels of parallelism here. The first level is where each thread executes a row of blocks from A along with a column of blocks from B. The second level is when a thread takes two blocks (one from A and one from B) to be multiplied. The third deeper level is when a single thread takes an entire row of A and an entire column of B of elements to result in a single resultant value in C. The fourth final level of parallelism occurs when a single element is taken from the row of A to be multiplied with another element of B.

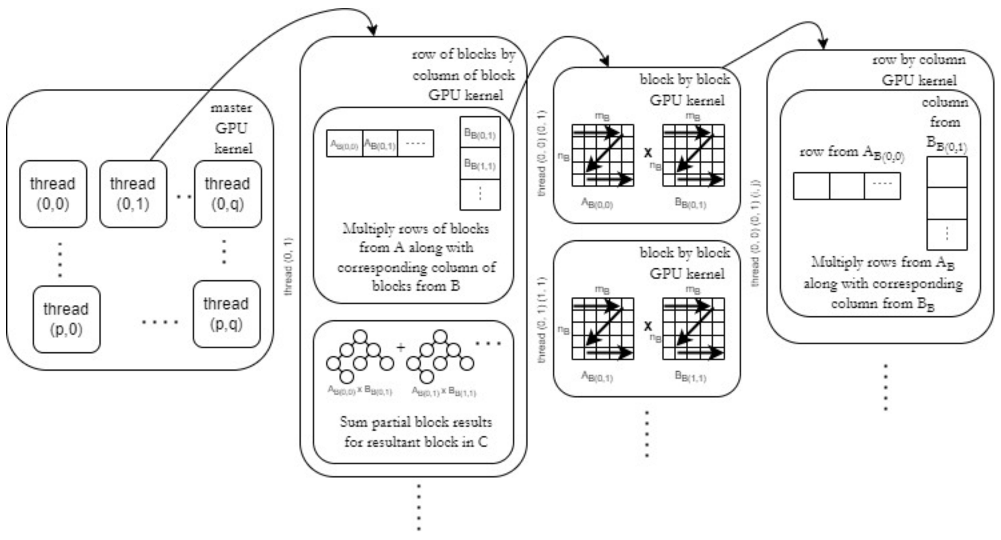

master GPU kernel
thread (0,0)
thread (0,1)
thread (0,q)
thread (p,0)
thread (p,q)
row of blocks by column of block GPU kernel
Multiply rows of blocks from A along with corresponding column of blocks from B
Sum partial block results for resultant block in C
block by block GPU kernel
row by column GPU kernel
column from B_B(0,1)
row from A_B(0,0)
Multiply rows from A_B along with corresponding column from B_B

Figure 4.10: Shows the two levels of parallelism in the GPU. p and q are the number of blocks on both the dimensions. $p = \frac{n}{n_B}$ $q = \frac{m}{m_B}$

Once we find the partial multiplication results of these values, we may use the scan operation to receive the resultant element in C. In this chapter, the first two levels of parallelism are implemented. The latter levels are the potential for future work.

4.5.1 Partial Multiplication

This algorithm takes a block of matrix A, and a block of matrix B as input and computes the partial resultant block of matrix C as shown in Figure 4.11. Since these blocks are of compute size, these block matrices are multiplied in parallel, where a row in block A and a column in block B, are multiplied to obtain a value in C. Once all threads finish multiplying the block, it is sent back to Algorithm 7.

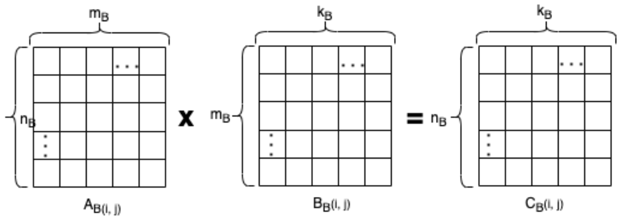


Figure 4.11: Shows the multiplication of blocks to obtain a partial result of resultant block.

Algorithm 5: multiply(nnz_A, nnz_B): A Parallel Partial Matrix Multiplication Algorithm

```
Input: Block in A, Block in B
Output: Partial Block of Resultant Matrix C
1 begin
2     do in parallel:
3         for a row of block_A: a do
4             for a row of block_B: b do
5                 if index_a == index_b then
6                     rowSum += a×b;
7         partialBlock_C += rowSum;
8     return partialBlock_C;
```

4.5.2 Compute Resultant Block

This algorithm computes the result of a block of matrix, this algorithm receives the starting index positions of each block in a row of matrix A, and starting index of each block in a column of matrix B. Since each block is independent of other blocks, these blocks can also be parallelized to obtain the results much faster. The algorithm is tweaked to take a blocked compressed data as an input and the starting index of the block, to obtain all non-zero elements in a row-major order. Each block of A and B is then sent to the algorithm 8 to obtain the partial resultant of the block. The blocks obtained from each thread are then reduced with respect to their indices to obtain the final result.

Algorithm 6: matrixMultiply(indicesA_tid, indicesB_tid): A Parallel Algorithm for Computing Resultant Block for Matrix C

```
Input: Row Indices of Matrix A, Column Indices of Matrix B
Output: Block of Resultant Matrix C
1 begin
2     /* Number of RowIndices == Number of Column Indices */
3     do in parallel: for each block in A and a block in B
4         nnz_A = matA[rowBlock].getNNZ()
5         nnz_B = matB[columnBlock].getNNZ()
6         partialBlock_C += multiply (nnz_A, nnz_B)
7     return block_C;
```

4.5.3 Parallel Matrix Multiplication

This algorithm is the final piece of matrix multiplication. Line 2 determines the number of threads required to compute the resultant matrix. From Line 3, each resultant block of C is computed by the thread associated with the row. The thread will handle the row of matrix A and a column of matrix B, and then this thread calls the procedure mentioned in algorithm 7 to obtain the final result of a block in C. Line 6, then computes the bitstring for the final resultant of the block in C (Gopal Krishna et al., 2021). These bitstrings are then appeneded one after the other in the order of the blocks in C.

Algorithm 7: Parallel Algorithm to Multiply Matrices

```
Input: Compressed Matrix A, Compressed Matrix B, Index Array id_A,
       Index Array id_B
Output: Resultant Matrix C
1 begin
2     numThreads = $\frac{(numRows_C * numColumns_C)}{block\_size^2}$
3     do in parallel: for each block in C
4         // call algorithm 7 to compute the resultant of each block
5         block_C = matrixMultiply(rowOf_A, columnOf_B)
6         bitString_Block_C = block_C.toBitString()
7     return ResultantMatrix_C
```

At the lowest level of these algorithms, when multiplying a row and a column to calculate the result element of the resultant matrix, there would be multiple queries to be performed, and this is going to result in multiple neighbor queries. To avoid this, a partial sum technique can be implemented that takes the two CBT structures and merges them by adding the common index values (Krishna et al., 2021).

4.6 Summary

In summary, we introduce ways to represent tensors of different modality. We also provide a mechanism to represent a tensor with any-mode, which can be used on any tensor. We introduced, a variable block matricized tensor compression on CBT, which in-turn aids to parallelize the matricized tensor multiplication.

Chapter 5

Matrix-Matrix Multiplication on Compressed Structure

Matrix multiplication is an essential operation in the field of mathematics and computer science. Many critical computations, such as matrix factorization and graph computations, cast the bulk of their computation in terms of this operation. Thus, it is crucial that this operation is tuned to the data being computed on. In the case of sparse domains, this translates to minimizing the traffic between the CPU and main memory as the amount of work is not necessarily sufficient to amortize the code of the data movement. The amount of memory required to store a nonnegative valued matrix of n rows and m columns requires $(n \times m) \times log_2(n)$ bits. When these dimensions are converted to real world scenarios, for example, a one billion by one billion matrix will require 1000 petabytes of memory, which is impractical. This hinders the ability to perform any operations on the matrix.

In this chapter, we propose techniques for performing Matrix-Matrix multiplication directly on compressed data stored in two different compression data structures. The structures we consider are the well-known compressed sparse matrix and the Compressed Binary Trees. We test our algorithm on extremely large matrices, in the order of 100s of millions with various levels of sparsity. We show for matrices of order 100 million with 10 million nonzero elements, the space required to store the matrices using the CBT representation is about 6.4MB and requires 13.52s to complete the multiplication using the sequential algorithms provided in this chapter.

5.1 Introduction

One of the most important operations in linear algebra is matrix multiplication. The naïve approach to multiply a $n \times n$ matrix requires $O(n^3)$ number of operations, which was first introduced in the year 1812 by a French mathematician Jacques Philippe. Later in the year 1969, Strassen (Strassen, 1969) introduced an algorithm to multiply matrices with at most $O(n^{2.81})$ number of operations, proving the naïve algorithm is not optimal.

Many real-world matrices that exhibit the small world phenomenon are *sparse* and extremely large. These matrices are usually in the range of millions to hundreds of millions or even billions. This means the number of nonzero elements in the matrix is meager compared to the total number of possible values in the matrix. For these datasets, sophisticated compressed data structures are employed to store only the nonzero elements and their associated indexing information. However, this space efficiency comes at the expense of complexity when computing over these structures. For compute-intensive classes of computations, such as Non-negative matrix factorization, this overhead is substantial. Thus, efficient implementation of these methods for compressed data structures is necessary for performing these operations over extremely large real-world matrices.

A large class of operations requires matrix factorization as one of the steps in achieving the result. Non-negative matrix factorization (NMF) is a computationally expensive operation as the method requires repeated matrix-matrix multiplication. The cost increases as the data gets larger and sparse.

In this chapter, we propose a technique to multiply two matrices using the compressed sparse row (CSR) (Gopal Krishna et al., 2021), and the compressed binary tree (CBT) (Nelson et al., 2019) with our novel integer encoding technique. The input to the algorithm is the compressed structure produced by either CSR or CBT, the matrices used for the analyses are both synthetically generated and real-world matrices with defined structures. The experimental analyses show the space and time trade-off between CSR and CBT, leaving the choice to choose the algorithm depending on the constraint of either space or time. Our method leverages the row-by-row compression of both CSR and CBT, enabling us to achieve the result much faster than the naïve matrix multiplication.

Our contribution is listed as follows:

- Value-based matrix-matrix multiplication algorithms on matrices that are stored as compressed binary trees (CBT).
- A novel bit packing mechanism to further improve compression sizes for both CBT and the CSR structures.
- We perform matrix-matrix multiplication without transposing the second matrix. We do this by a clever partial sum calculation.
- Our matrix-matrix multiplication algorithm stores the result (resultant matrix) directly into the compressed structure (CSR or CBT) without creat-

ing any secondary data structures.

- We have provided an extensive empirical evaluation of our algorithms on extremely large matrices that can be in 100's of millions in size (number of rows and columns) with various degrees of sparsity.

The rest of the chapter is divided as follows. Section 5.2 describes the various ways of storing the matrices, and this section also includes an introduction to various integer encoding techniques. Section 5.3 introduces the different approaches to multiply two matrices. This section also includes the detailed algorithm for CSR, and CBT multiplication. Section 5.4 describes the experimental evaluations performed on various matrices. and conclude the chapter with Section 5.5.

5.2 Matrix Representation

In this chapter, the matrices are represented based on the nonzero values present in each row of the matrix. This requires representing the coordinates of the nonzero elements and their respective values, which can be stored as a coordinate array and a value array. In this chapter, we use the Compressed Sparse Row (CSR) representation introduced by Snay, Richard (Snay, 1976a) with the variables defined in Krishna et al. (Gopal Krishna et al., 2021), and our novel Compressed Binary Tree (CBT) (Nelson et al., 2017), structures to store these matrices. In the previous work of matrix multiplication presented by Nelson et al. (Nelson et al., 2017), the matrices are considered boolean. Here, we extend the similar concept of storing the matrices with an additional integer encoding to represent all nonzero values of the matrix.

5.2.1 CSR Representation

Snay, Richard in 1976 (Snay, 1976a) first came up with the Compressed Sparse Row representation for sparse matrices. Since then, CSR has been one of the widely used sparse matrix representations in linear algebra and graph algorithms. CSR is a row-row compression that is represented using three arrays, the first one being the starting position of each row, the second being the column where the nonzero element is present in that particular row, and the third one being the value of each nonzero element.

To represent all these arrays as integers, the total number of bits required is $(2m+n)*64$. Instead, our work uses the novel bit-packing algorithm proposed in (Gopal Krishna et al., 2021) for a time-evolving graph. This reduces the number of bits required to $(m \times log_2(n)\ +\ m \times log_2(\gamma)\ +\ n \times log_2(m))$, where γ represents the maximum nonzero value.

$m \times log_2(n)$ to represent the column positions of nonzero elements, $m \times log_2\gamma$ to represent the values of all nonzero elements, and $n \times log_2(m)$ to represent the starting position of each row.

5.2.2 CBT Representation

In 2017, Nelson et al. (Nelson et al., 2017) introduced Compressed Binary Tree representation for an unweighted graph or boolean matrix. This data structure represents the matrix one row at a time, which eliminates the need for an intermediate data structure to convert from the input to the compressed format. The compressed structure of each row is represented as a preorder traversal of a tree.

To represent a matrix as CBT, the structure requires $n \times log_2(n)$ number of bits. However, for a weighted matrix, CBT representation would require $(n \times$

$log_2(n) + m \times log_2(\delta))$ (if we use bit-packing as an integer encoding scheme).

5.2.3 Integer Encoding

The values in the matrix stored as integers require 64 bits to represent each number, for example, a 1 billion by 1 billion matrices with sparsity $10^{(-9)}$, which will contain 1 billion values, and the data structure requires 64 billion bits or about 8 gigabytes. This contributes to a large chunk of the total memory required. To overcome this issue, all integers can be encoded/represented in a binary format to reduce the amount of space required to store each integer. In this chapter, we use a novel integer encoding, called Bit-Packing (Gopal Krishna et al., 2021), and then compare the performance with the well-known encoding techniques such as 1) Elias Gamma Encoding (Elias, 1975) and 2) Elias Delta Encoding (Elias, 1975).

Elias Gamma Encoding

Elias Gamma encoding was introduced in 1975 by Elias Peter (Elias, 1975), which postulates bit-encoding of positive integers. Gamma encoding is most commonly used when the upper bound of the integer values could not be determined at the start. The first step in gamma encoding involves computing the unary notation for the minimum number of bits required to store the integer value in binary format. The second step is to append the binary representation of the integer value, which is one more of the number of bits required to store the unary notation. Let x be the positive integer value, where $x \geq 1$, the unary notation takes $\lfloor log_2(x) \rfloor$ bits, and the binary representation requires $\lfloor log_2(x) + 1 \rfloor$ bits. For example $1 \rightarrow 1$, $2 \rightarrow 010$, $3 \rightarrow 011$, $1000 \rightarrow$ 0000000001111101000. Therefore, as the number increases, the gamma encoding becomes inefficient.

Elias Delta Encoding

Elias Delta encoding was introduced by Elias Peter (Elias, 1975) in 1975, which replaces the Elias Gamma encoding algorithm. In Elias Delta, the first step is to store the gamma encoding of $\lceil log_2(x+1) \rceil$, and then in the second step, delta encoding stores all but most significant bits of the binary notation of the number x. For example, if $x = 8$, $\lceil log_2(8+1) = 4 \rceil$, hence the gamma encoding of 4 00100, and the all but most significant bit of 8 is 000, therefore, 8 can be encoded as 00100000.

Bit-Packing

Bit-packing is a novel encoding scheme introduced in (Gopal Krishna et al., 2021), in this technique, given an array of integers, the numbers are represented based on the minimum number of bits the largest number consumes in the array.

Table 5.1, shows the number of bits required (in megabytes) to store random integers using Elias Gamma, Elias Delta, and Bit-packing algorithms. From the table, it is evident that bit-packing performs better than both Elias Gamma and Elias Delta Encoding.

This statement of bit-packing performs better than both Elias Gamma and Elias Delta encoding can be backed up by looking at the theoretical aspect of the storage. $(2 \times (\lfloor log_2(x) \rfloor) + 1) \geq (\lfloor log_2(x) \rfloor + 2 \times (\lfloor log_2 \lfloor \log_2(x) + 1 \rfloor) + 1) \rfloor \geq \lceil log_2(x) \rceil$.

Hence, in this chapter, all integer values in the matrices are stored as bits using the bit-packing algorithm.

Table 5.1: Shows the size of the memory required to store the integer values using various integer encoding schemes.

ArraySize	Elias Gamma	Elias Delta	Bit-Packing
	in MB		
1000	0.002	0.0017	0.0011
100000	0.36	0.27	0.20
10000000	52.02	36.13	28.61
100000000	599.24	400.94	321.86

5.3 Matrix-Matrix Multiplication

5.3.1 Naïve Algorithm

Given A and B matrices of dimension $n \times k$ and $k \times m$ respectively, our goal is to multiply them and store the result in a matrix C of dimension $n \times m$. We consider an example to illustrate the multiplication algorithm shown in 5.1. To compute the first element $(0, 0)$ in our resultant C matrix, we multiply the corresponding values in $r_0(A)$ (r_i – the i^{th} row of A) and $c_0(B)$ (c_j – the j^{th} column of B)and cumulatively add them. This operation is repeated by looping through each column of B to compute the values in $r_0(C)$. This is mathematically represented as

$$C = \sum_{p=0}^{n} \sum_{q=0}^{m} \sum_{r=0}^{k} C_{pq} \mathrel{+}= A_{pr} \times B_{rq} \tag{5.1}$$

$$\begin{array}{cc} \text{A} & \text{B} \\ \begin{bmatrix} 5 & 0 & 2 & 3 \\ 3 & 0 & 0 & 5 \\ 0 & 0 & 2 & 4 \\ 0 & 1 & 2 & 0 \end{bmatrix} & \begin{bmatrix} 2 & 2 & 0 & 3 \\ 4 & 0 & 1 & 0 \\ 3 & 0 & 1 & 2 \\ 1 & 1 & 0 & 0 \end{bmatrix} \end{array}$$

Figure 5.1: Matrices A and B

The time complexity in the worst case for this algorithm is $O(nmk)$ or $O(n^3)$ if A and B are square matrices. Breaking the time complexity to compute a single element C_{ij} in the resultant matrix C, it takes $O(k)$ once we fetch $r_i(A)$ and $c_j(B)$.

Taking the example shown in 5.1, where matrix A and B are shown and highlighted in blue are $r_0(A)$ and $c_0(B)$, we show the calculation that goes into computing the value of C_{00}. Values from $r_0(A)$ are matched with the values from $c_0(B)$ by matching the column numbers of values from $r_0(A)$ and row numbers of the values of $c_0(B)$. By doing the cumulative addition of these multiplied values, we get the value 19 for C_{00}, as shown in 5.3.

This, however, can be improved in the case of a sparse matrix considering the number of nonzero elements in $r_i(A)$. Which means we can skip the $n - numNonZero(r_i(A))$ multiplications and cumulative additions, where

$$numNonZero(r_i(A))$$

represents the number of non zero elements in row i of matrix A (Gustavson, 1978).

Although this above algorithm works well for the data structure of a 2D array/matrix, for the compressed data structure that we use in this chapter (Snay, 1976a)(Nelson et al., 2017), the algorithm can be optimized for better efficiency. More about the motivation and the partial sum algorithm follows.

$$A \times B = \begin{array}{c|cccc|} & 0 & 1 & 2 & 3 \\ 0 & 5 & 0 & 2 & 3 \\ 1 & 3 & 0 & 0 & 5 \\ 2 & 0 & 0 & 2 & 4 \\ 3 & 0 & 1 & 2 & 0 \end{array} \times \begin{array}{c|cccc|} & 0 & 1 & 2 & 3 \\ 0 & 2 & 2 & 0 & 3 \\ 1 & 4 & 0 & 1 & 0 \\ 2 & 3 & 0 & 1 & 2 \\ 3 & 1 & 1 & 0 & 0 \end{array} \tag{5.2}$$

$$\Rightarrow \begin{array}{r} r_0(A) \rightarrow \\ \\ c_0(B) \rightarrow \end{array} \begin{pmatrix} 5 \\ \times \\ 2 \end{pmatrix} + \begin{pmatrix} 0 \\ \times \\ 4 \end{pmatrix} + \begin{pmatrix} 2 \\ \times \\ 3 \end{pmatrix} + \begin{pmatrix} 3 \\ \times \\ 1 \end{pmatrix}$$

$$\Rightarrow r_0[C] = C_{00} = 19 \tag{5.3}$$

5.3.2 Partial Sum

The compressed data structures that are used in this chapter for storing and multiplying (Snay, 1976a)(Nelson et al., 2017), are both techniques that compress matrices row by row. Because of the row-by-row compression of these two data structures, querying becomes a challenge as we are able to only query the values per row. Therefore, the column queries ($c_i(B)$) would involve querying each of rows of the matrix and indexing the first values from them to build the first column. This means that the column query would have to be performed k times (number of rows) for matrix B in our example. This process is to just build one column ($c_0(B)$) to perform the multiplication to find the resultant value C_{00}, in our example. To compute the resultant row ($r_i(C)$), we would need to do row queries $k \times m$ number of times. This becomes a very costly step in our algorithm. One way to circumvent this issue would be to transpose the B matrix so that a column query would mean actually mean row query which would fetch us all the required values. This becomes a preprocessing step before multiplication and turns out to not be very convenient as well. The second approach would be to store B as a column compressed format, but this would mean that we are going to be storing A and B in two different formats and making it nonuniform.

The *partial sum* idea explained in algorithm 8, is to compute one entire row in the resultant matrix, $r_i(C)$ with just n row queries to B.

Taking the example given in equation 5.4, the goal is to compute one entire row in our resultant $r_0(C)$, from A and B matrices. The first step is to query $r_0(A)$ and identify only the nonzero indices. The corresponding row is queried from B. In our example, 5 is the first nonzero element from $r_0(A)$ at index (column number) 0 in A. The corresponding row is queried from B, $r_0(B)$. Now, a dot product is performed between 5 and $r_0(B)$ and the result is stored in a temporary row (resultantRow) which is of length m. This temporary row holds the partial sum of $r_0(C)$ and is shown in equation 5.5. The next query would be $r_2(B)$ since that is the next index position where the next nonzero element is found in $r_0(A)$. As seen in equation 5.5, a dot product is performed between the value 2 and $r_2(C)$. The resultant partial sum after this product is cumulatively added to resultantRow as shown in equation 5.6.

Once we loop through the nonzero indices of $r_0(A)$, we now have an entire row of the resultant matrix, $r_0(C)$. The number of queries performed to get the row of B is now n.

A similar idea about using partial sum calculations was used in value-based matrix-matrix multiplication using a data structure called Single Tree Adjacency Forest (STAF) (Nishino et al., 2014). This however had an assumption that the matrices would follow the column scaled nonzero property, which puts a big constraint in regard to value-based matrices. In this chapter, we use a compressed value-based matrix data structure to perform multiplication without any intermediate structures and stream the result into the resultant structure directly.

$$A \times B = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{bmatrix} 5 & 0 & 2 & 3 \\ 3 & 0 & 0 & 5 \\ 0 & 0 & 2 & 4 \\ 0 & 1 & 2 & 0 \end{bmatrix} \end{array} \times \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{bmatrix} 2 & 2 & 0 & 3 \\ 4 & 0 & 1 & 0 \\ 3 & 0 & 1 & 2 \\ 1 & 1 & 0 & 0 \end{bmatrix} \end{array} \tag{5.4}$$

$$\Rightarrow \begin{array}{c} 5 \times r_0(B) \Rightarrow 5 * \begin{pmatrix} 2 & 2 & 0 & 3 \end{pmatrix} \\ 2 \times r_2(B) \Rightarrow 2 * \begin{pmatrix} 3 & 0 & 1 & 2 \end{pmatrix} \\ 3 \times r_3(B) \Rightarrow 3 * \begin{pmatrix} 1 & 1 & 0 & 0 \end{pmatrix} \end{array} \tag{5.5}$$

$$\Rightarrow \quad \begin{array}{c} \begin{pmatrix} 10 & 10 & 0 & 15 \end{pmatrix} \\ + \\ \begin{pmatrix} 6 & 0 & 2 & 4 \end{pmatrix} \\ + \\ \begin{pmatrix} 3 & 3 & 0 & 0 \end{pmatrix} \end{array}$$

$$\Rightarrow r_0(C) \rightarrow \begin{pmatrix} 19 & 13 & 2 & 19 \end{pmatrix} \tag{5.6}$$

Algorithm 8: Computing the Resultant Row Using Partial Sum

```
Input: A row of matrix A, matrix B
Output: Resultant row of matrix C
1 begin
2     for everyNeighbor in A_row do
3         B_row = getRow(B, everyNeighbor)
4         for everyCol in B_row do
5             resultantRow[everyCol] += A.values[A_row]*B.values[B_col]
6     return resultantRow
```

5.3.3 *CBT* Matrix-Matrix Multiplication

Let A & B be two matrices to be multiplied, in this section, we adapt the partial sum algorithm from section 5.3.2 to multiply two matrices which are stored in a Compressed Binary Tree (CBT) format. Then finally store the resultant matrix C as a Compressed Binary Tree format. In this process of multiplication, the resultant indices are directly stored in CBT without any intermediate structures. The following algorithm walks through the implementation of CBT Matrix-Matrix Multiplication.

Algorithm 9, takes in the row of bits compressed using CBT, and the associated row number. The algorithm reads one bit at a time from the bitstring which is traversed in a preorder (line 9). If the bit read is equal to 1, that means there is at least one edge present in the range. Line 10 checks if the traversal has reached the leaf, if so, we append the index associated with the range to the final resultant rowIndices (Line 11). Lines 13 through 20 checks for special cases that are associated with CBT compression. Lines 13 through 17 retrieve the row number in the range of indices, and line 19 retrieves all row numbers in the range of indices.

Given matrix A, and matrix B as two compressed binary trees (Nelson et al.,

Algorithm 9: Get A Row From *CBT*

```
Input: A bitstring of a row ($rowString$), and the row number ($u$)
Output: All indices of non-zero elements in the row $u$
begin
    int[] rowIndices
    maxDepth = $log_2(numNodes)$
    beginCol = 0
    endCol = numNodes - 1
    index = 0 /*index to traverse the row*/
    preOrderStack.push(pair(beginCol, endCol))
    while !preOrderStack.empty() do
        nodeLabel = rowString.getBit(index)
        if nodeLabel == 1 &&$IsMaxDepth$() then
            rowIndices.append(beginCol)
            index += 1;
        else if nodeLabel == 1 && nodeLabel.leftChild == 0 &&
         nodeLabel.rightChild == 0 then
            if nodeLabel.getBit(index+3) == 1 then
                colNum = rowString.getRelativePath()
                rowIndices.append(colNum)
                index += 4 + lengthOfPath;
            else
                rowIndices.append(range(beginCol, endCol))
                index += 4;
        else
            nodeLabel.ignore();
    return rowIndices;
```

2017), algorithm 10 multiplies $A \times B$, to produce matrix C in the compressed binary tree. For each row in the matrix A_CBT, line 3 fetches all nonzero column numbers associated with the row using algorithm 9. The row obtained in line 3 from A_CBT, is then fed into algorithm 8 along with B_CBT, and the values associated with both the matrices encoded using bit-packing algorithm 2. The row of resultant integers obtained from the partial sum algorithm is the resultant row of matrix C. Furthermore, each index associated with the resultant row is then streamed onto the compressed binary tree C_CBT, and all resultant multiplied values are stored as bits using the bit-packing algorithm (lines 6-7).

Algorithm 10: *CBT* Matrix-Matrix Multiplication

```
  Input: A_CBT, B_CBT
  Output: C_CBT
1 begin
2     for row = 0 to numberOfRows do
3         Arow = getRow(A_CBT, row)
4         tempArray = comutePartialSum(Arow, B_CBT)
5         for nnzElements in tempArray do
6             C_CBT[row].streamEdge(nnzIndex)
7             C_CBT.values.bitPack(nnzValues)
8     return C_CBT
```

5.3.4 *CSR* Matrix-Matrix Multiplication

The approach to the CSR matrix-matrix multiplication is very similar to that of the CBT matrix-matrix multiplication 5.3.3. Consider two matrices A & B represented in the compressed sparse row format (Gopal Krishna et al., 2021), the objective of this algorithm is to compute the product $A \times B$ and store the resultant C in the compressed sparse row without any intermediate structures that aid the storage.

Algorithm 11 provides the methodology to retrieve a row from the matrix, compressed using CSR. The algorithm takes in the unsigned bit representation of the row u, the row number u, and the number of bits required to store each integer of the row. Line 16 retrieves the bits associated with the integer to the variable res, as the system architecture stores the numbers in the little-endian format, the bits in the res are stored in the most significant bit. Lines 17-18 push the bits into the appropriate position to represent the actual integer stored in the array (line 20). If the current array location contains a part of the number, lines 21-31 will retrieve the partial number from the current location and the remaining bits from the concurrent location to form an integer value. This process is repeated until all numbers are retrieved from the given $uArray$. Finally, line 35 will return all column numbers associated with the row where nonzero elements are present.

Now, the second step of the process is the matrix-matrix multiplication. Algorithm 12 takes in matrix A, A_CSR and matrix B, B_CSR as arguments. For all nonzero rows present in matrix A_CSR, line 3 obtains one row of elements from A_CSR, this is then fed into algorithm 8 along with B_CSR, and the values associated with both the matrices encoded using bit-packing algorithm 2 (line 4). The row of resultant integers obtained from the partial sum algorithm is the resultant row of matrix C, along with the number of elements (degree) of the resultant row (line 5). Furthermore, each index associated with the resultant row is then streamed on to the compressed sparse row C_CSR, and all resultant multiplied values are stored as bits using the bit-packing algorithm (lines 7-8).

Algorithm 11: Get A Row From CSR

```
Input: An array of unsigned bits $uArray$, the row number ($u$), and the
       number of bits ($numBits$)
Output: All indices of non-zero elements in the row $u$
begin
    int[] rowIndices
    totalBits = 64
    balance = 0
    arraySize = $\frac{numberofElements}{numberofBits} * totalBits$;
    for $i$ = 0 to arraySize do
        /* denotes how many numbers are stored in an array location */
        m = availBits/numBits
        /* leftover bits at the end of the array */
        remBits = availBits%numBits
        /* keep track of the number */
        ind = 0
        while ind < m do
            res = 0
            j = 0
            res = $uArray[i]$ >> (ind*numBits + balance)
            j = res << (totalBits - numBits)
            j = j >> (totalBits - numBits)
            ind ++
            rowIndices.append(j)
        if rem > 0 then
            /* First read the remaining bits in index */
            res = $uArray[i]$ ¿¿ (totalBits - rem)
            /* Increment the array index */
            i += 1
            /* Now get the remaining bit from the next array location */ j
             | = res
            res = b << rem
            res = res << (totalBits - numBits)
            res = res >> (totalBits - numBits)
            j | = res
            /* update the remaining bits in the new location */
        else
            i ++;
            /* reset all parameters */
    return rowIndices;
```

Algorithm 12: *CSR* Matrix-Matrix Multiplication

```
Input: $A\_CSR$, $B\_CSR$
Output: $C\_CSR$
1 begin
2     for row = 0 to numberOfRows do
3         Arow = getRow($A\_CSR$, row)
4         tempArray = comutePartialSum(Arow, $B\_CSR$, $A\_values$,
            $B\_values$)
5         $C\_CSR.degree[row]$ = nnz;
6         for nnzElements in tempArray do
7             $C\_CSR.indices[row]$.bitPack(nnzIndex)
8             $C\_CSR.values$.bitPack(nnzValues)
9     return $C\_CSR$
```

5.4 Experimental Results

For the analyses involving matrix-matrix multiplication, we consider the following datasets shown in table 5.2. For the analyses, the matrices are generated using either of the four following properties, 1) Power-Law distribution, 2) Erdő-Rényi random generator, 3) Random Walk generator, and 4) Real-World data. The first three properties are synthetically generated using the appropriate functions defined in the properties of random networks, and the real-world data are obtained from the Stanford Large Network Dataset Collection (Snap)(Leskovec and Krevl, 2021).

Erdő and Rényi introduced the notion of random graph (Erdos et al., 1960) (Erdős and Rényi, 1961) in the year 1960. There are three important parameters in generating a random graph, they are the number of nodes n, the density of the graph (number of edges) e, and the probability of an edge exist in the graph p. The $G(n, e)$ model random graphs, generate a uniform number of edges for the all the nodes n, which is not a feasible structure. Therefore, $G(n, p)$n model as

Table 5.2: Shows the various datasets used for evaluating the matrix-matrix multiplication.

Type of Graph	Number of Rows	Number of Elements	Sparsity	.txt (MB)	CSR in MB	CBT in MB
Synthetic Matrices						
Power Law Generator	100,000	99,999	$1 \times 10^{(-5)}$	1.2	0.33	0.26
		199,996	$2 \times 10^{(-5)}$	2.5	0.59	0.5
Erdos-Renyi Random Generator	1,000,000	500,894	$5 \times 10^{(-7)}$	7.5	1.76	1.69
		2,500,783	$2.5 \times 10^{(-6)}$	38	7.44	7.59
Power Law Generator	10,000,000	9,999,999	$1 \times 10^{(-7)}$	163	48.87	34.72
		19,999,996	$2 \times 10^{(-7)}$	326	80.15	67.11
Random Walk Generator	1,000,000	1,000,000	$1 \times 10^{(-6)}$	16	3.21	3.2
	10,000,000	1,000,000	$1 \times 10^{(-8)}$	17	6.79	4.77
	100,000,000	10,000,000	$1 \times 10^{(-9)}$	189	71.52	51.25
Real-World Matrices						
Amazon	403,394	3,387,388	$2.08 \times 10^{(-5)}$	49	9.07	7.61
Web-NorteDame	325,729	1,497,134	$1.41 \times 10^{(-5)}$	22	4.38	1.97
Facebook	4,039	88,234	$5.41 \times 10^{(-3)}$	0.1	0.143	0.083
Email-EU	265,214	420,045	$5.97 \times 10^{(-6)}$	5.2	1.41	1.1

described Erdos and Renyi, assigns the edge with the chosen probability p.

The power-law random graph generator takes a page out of the $G(n, p)$ model. The power-law random graph model $P(\alpha, \beta)$ is described as follows. Let v be the number of nodes with degree x, $P(\alpha, \beta)$ assigns uniform probability to all the nodes with $v = e^{\alpha}/x^{\beta}$.

The last random graph generator used in this chapter is obtained by using the technique of random walk. The graph generator produces two random numbers $(u,\ v)$ from the non-negative integer line $\mathbb{Z}$.

The amazon network (Leskovec and Krevl, 2021), is based on the product recommendation to customers who bought the similar products as a bundle. That is, if a product a is purchased frequently along with another product b by the customers, then there exists a directed edge $a \rightarrow b$. The Web-NorteDame network (Leskovec and Krevl, 2021), is obtained from the University of Norte Dame, the directed edge represents the hyperlinks within the nd.edu domain. The data was collected by Albert, Jeong and Barabasi (Albert et al., 1999) in 1999. The Facebook network (Leskovec and Krevl, 2021), consists of 'circles' (or 'friend lists') on Facebook. The dataset includes node features (profiles), circles, and ego networks. The Email-EU network (Leskovec and Krevl, 2021), the dataset was obtained from the communication with European universities. Overall, there are 3,038,531 emails between 287,755 different email addresses. A directed edge $u \rightarrow v$, corresponds to an email sent from u to another email address v.

Table 5.2, shows the statistics of the datasets used in this chapter. The table contains the number of rows in column 2, the number of nonzero elements in the matrix in column 3, the sparsity of the matrix in column 4, and followed by the size of the files expressed in megabytes (MB) in the text format mentioned in column 5, the compressed sparse row (CSR) in column 6, and in a compressed binary tree (CBT) in column 7.

The sparsity of a matrix is defined as the ratio of the number of nonzero elements to the number of all possible elements that can be in the matrix.

$$Sparsity = \frac{number of nonzero elements}{number of rows \times number of rows}$$

All experiments were run on an Intel(R) Xeon(R) CPU E5520 @ 2.27GHz (16 Cores) with 64 GB of RAM, and the programs are written in GNU C/C++.

Table 5.3, shows the results obtained from multiplying two matrices using compressed sparse row (CSR) and the compressed binary tree (CBT) format. From the results, the pattern of space and time trade-off is very evident. On average, CBT consumes 30% lesser space compared to CSR, where as CSR makes its ground by consuming on an average 40% less time compared to CBT.

Both CBT and CSR require substantially less amount of memory compared to the file stored in the text.

We also ran the experiments on random generated graphs. We performed an extensive run on a 1 Million by 1 Million, with the sparsity $2.39 \times E^{-6}$.

Where we saw, CSR completed the multiplication in 1.17 ∓ 0.05 secs, where as CBT completed in 5.49 ∓ 0.03 secs.

Meanwhile, CSR consumed 4.72 ∓ 0.04 MB, and CBT consumed $4.39MB$ to store the resultant matrix.

5.5 Summary

In summary, we introduced matrix multiplication on the compressed structures, CSR and CBT, using the Partial Sum algorithm, which yielded in a row-by-row

Table 5.3: Shows the space and time performance of CBT and CSR matrix-matrix multiplication.

Number of Rows	Number of Elements		Size after Multiplication			Time taken to multiply	
Synthetic Matrices							
	Before Multiplication	After Multiplication	.txt in MB	CSR in MB	CBT in MB	CSR in secs	CBT in secs
100,000	99,999	99,680	2.28	0.39	0.29	0.043	0.29
	199,996	396,732	9.08	1.23	0.71	0.14	0.98
1,000,000	500,894	167,974	3.84	0.97	0.67	0.25	0.82
	2,500,783	4,173,155	95.51	13.64	16.1	1.92	11.39
10,000,000	9,999,999	9,996,306	228.79	53.64	34.3	5.93	33.12
	19,999,996	39,962,182	914.66	163.09	133.82	17.79	111.77
1,000,000	1,000,000	999,326	22.89	3.57	3.37	0.75	2.8
10,000,000	1,000,000	99,492	2.27	3.919	0.589	0.82	1.34
100,000,000	10,000,000	1,000,194	22.87	39.56	6.31	8.59	13.52
Real-World Matrices							
403,394	3,387,388	16,258,436	372.12	37.16	24.25	5.43	68.95
325,729	1,497,134	16,801,350	384.55	38.63	3.18	7.42	25.83
4,039	88,234	337,529	7.72	0.48	0.17	0.26	2.66
265,214	420,045	46,273,509	1059.11	105.31	67.66	66.19	12.85

computation. We tested our algorithm on 100 million by 100 million matrices of different sparsity and also on real-world matrices.

Chapter 6

Matrix Factorization on Compressed Structure

Non-negative Matrix Factorization (NMF) is one of the algorithms with a wide range of applications, from dimensionality reduction and computer vision to text mining. The dimensions of these matrices can be of the order of several hundreds of thousands to millions, which is a raw format that would not fit in the main memory. Additionally, while performing matrix factorization on these extremely large matrices, the algorithms involving matrix operations such as transpose, multiplication, and subtraction; demand more storage for intermediate resultant matrices. In this chapter, we store the matrices in compressed structures (Compressed Binary Tree CBT and Compressed Sparse Row CSR) that allow factorization without decompression. We also perform factorization CBT without using any intermediate structures by performing a virtual transpose and streaming the intermediate resultant matrices of a sequence of matrix multiplications directly into the compressed structure for every iteration. As an example, for an input matrix A of dimension $65,536 \times 65,536$ with $1.46M$ number of non-zero elements, the peak storage in any iteration of the multiplicative update factorization algorithm is $32.98GB$ when using a 2D array, $200MB$ when using CSR and $14.8MB$ for CBT. The ability to stream (add and delete) into the CBT structure without reallocation is why CBT performs the best. Furthermore, we provide a heuristic to reduce memory usage that also aids in faster convergence.

6.1 Introduction

Non-negative Matrix Factorization (NMF) can be formally defined as follows: Given a non-negative matrix $A \in \mathbb{R}_+$ of dimension $m \times n$ and an inner dimension $k > 0$, find the factor matrices if any, $W \in \mathbb{R}_+$ of dimension $m \times k$ and $H \in \mathbb{R}_+$ of dimension $k \times n$ such that:

$$A = WH$$

The factor matrices W and H are also non-negative in nature. The rank of the input matrix A gives a lower bound for the inner dimension k. This inner dimension k is referred to as the Non-negative rank of a matrix. This problem of finding the factors that satisfy condition $A = WH$ with $rank(A) = k$ has proved to be an NP-hard problem (Vavasis, 2010) (Shitov, 2017). The short proof of (Shitov, 2017) tries to reduce the graph coloring problem and equates the NP-hardness of the graph chromatic number with the non-negative ranks of the input matrix, which is the smallest inner dimension for NMF.

There are various applications (Gillis, 2014) that use NMF from computer vision, text mining/information retrieval, email, and pattern recognition to clustering in machine learning (Xu et al., 2003), face recognition (Guillamet and

Vitria, 2002) and data mining (Berry and Browne, 2005; Zhang et al., 2006). Another application of NMF is that it can be used as a lossy compression algorithm to compress a large matrix. If the inner dimension k is small enough, then the input matrix A can be factored in $W \times H$, resulting in a lower number of elements in total. The number of elements in A to be stored will be $m \times n$, but if factorized, the number of elements to be stored will be $m \times k + k \times n$. The latter is assumed to be smaller when k is small.

The correctness of the factorization is calculated using the Frobenius norm suggested by (Lee and Seung, 2001) (Guan et al., 2012). Now, the problem can be rewritten as:

$$\min_{W \geq 0, H \geq 0} \|A - WH\|_F$$

Some of the well-known sequential algorithms to solve the non-negative factorization are, *Multiplicative Update Algorithms(Lee and Seung, 2001) (Gonzalez and Zhang, 2005), Gradient Descent Algorithms and Alternating Least Squares Algorithms(Berry et al., 2007) (Kim and Park, 2008).* There are several approaches as defined in (Wang and Zhang, 2012) that can be taken to solve this problem. In this chapter, we will evaluate the Multiplicative Update Algorithm defined by Lee & Seung (Lee and Seung, 2001).

To solve any of the sequential algorithms mentioned above for large matrices, the algorithms require a system configuration that can handle a huge number of gigabytes of data at a time. We present two state-of-the-art compressed structures (CBT (Nelson et al., 2019) and CSR (Gopal Krishna et al., 2021)) that are used to store these matrices and used for operations and algorithms. The input matrices and the factor matrices are all stored in either of these compressed structures. The matrices used for the analyses are both real-world and synthetically generated. We have also shown that there is a space-time trade-off between the two structures CBT and CSR. CBT taking lesser space and CSR having a shorter query time (Gopal Krishna et al., 2021).

Our contribution is as follows:

- We provide a method for factorizing matrices with the least memory footprint per iteration using compressed structures.
- Sections 6.2.1 and 6.2.2, explain various value-based matrix−matrix operations that are performed without decompression.
- We provide a matrix-transpose multiplication algorithm (Section 6.2.3) that provides results without transposing, by streaming the result directly into compressed structures.
- In Section 6.2.4, we explain how we sequence 3 or more matrix multiplication operations, without storing any intermediate matrices.
- Proposed a heuristic (Section 6.2.5) that eliminates unnecessary rows/-columns that leads to lower memory usage and faster convergence.

This chapter is divided as follows, In Section 6.2, we go through all the algorithms required to compute the factorization using multiplicative update algorithm. In Section 6.3, we show the experimental results of the factorization. We conclude our work in Section 6.4.

6.2 Matrix Factorization

There are several approaches that can be taken to factorize a given matrix. To mention a few of the popular ones, multiplicative update, gradient descent, and alternating least squares (Lee and Seung, 2001)(Berry et al., 2007). Here, we take the updated rules provided by Lee and Seung (Lee and Seung, 2001).

$$H \leftarrow H \frac{(W^T V)}{(W^T W H)}, \quad W \leftarrow W \frac{(V H^T)}{(W H H^T)}$$

Algorithm 13: Multiplicative Update Algorithm

```
   Input: Matrix to be factorized $A$.
   Output: Factorized matrix $W$ and $H$.
1  begin
2      $W = rand(m, k)$
3      $H = rand(k, n)$
4      for $i : maxiter$ do
5          $H \leftarrow H \ .* \ (W^T A) \ ./ \ (W^T W H + 10^{-9})$
6          $W \leftarrow W \ .* \ (A H^T) \ ./ \ (W H H^T + 10^{-9})$
```

Algorithm 13, shows the workings of how to factorize the given large matrix using the multiplicative update algorithm. The algorithm involves a series of operations to obtain the desired result of W and H. To clarify the various matrix element-wise operations, the $.*$ operation represents an element-wise multiplication, and a $./$ represents an element-wise division and matrix-based operations such as matrix-matrix multiplication. So, we continue this section by providing the algorithms for the various operations that are the building blocks of 13.

6.2.1 Matrix-Matrix Multiplication

One of the first and most important operations to be performed during the factorization process is matrix-matrix multiplication. The work on matrix-matrix multiplication has been published in (Krishna et al., 2021), which explains the working of how two matrices stored in either of the data structures CSR and CBT are multiplied without the need for an intermediate data structure.

6.2.2 Element-Wise Matrix Operation

The multiplicative update algorithm consists of several element-wise matrix operations. The operations involved in the algorithm are element-wise multiplication $.*$, element-wise division $./$, and element-wise subtraction $-$ to find the Frobenius norm. Apart from these three, we can also extend the algorithm for element-wise addition $+$.

Algorithm 14, explains the working of the element-wise matrix operation. The operation to be performed, ”Op,” is specified as input. The algorithm first checks if the dimensions of the two matrices are equal and, if not, throws an error. It then loops through each row of the matrices, and for each row, it checks if the size of the row is zero in either matrix. If it is, it appends a zero to the corresponding row of the resultant matrix C. If both matrices have a row of size zero, it also appends a zero to the corresponding row of C. If only one matrix has a row of size zero, it copies the elements from the non-zero row and appends them to the corresponding row of C. If neither matrix has a row of size zero, the algorithm performs the specified operation on each element of the corresponding rows of A and B and appends the result to the corresponding row of C. Finally, the algorithm returns the resultant matrix C.

Algorithm 14: Element-wise matrix Addition, Subtraction, Multiplication, and Division

```
  Input: Matrix A, Matrix B, Operation Op
  Output: resultan_matrix C
1 begin
2     if A.rowSize != B.rowSize or A.colSize != B.colSize then
3         Error: Matrix dimensions should be the same for both the
          matrices
4     for i in numberofRows do
5         if A[i].rows == 0 and B[i].rows == 0 then
6             C[i] = 0
7             continue to the next row
8         else if A[i] == 0 then
9             C[i] = B[i]
10            continue to the next row
11        else if B[i] == 0 then
12            C[i] = A[i]
13            continue to the next row
14        for aIndex in A[i] do
15            for bIndex in B[i] do
16                C[i][j] = A[i][j] "Op" B[i][j]
17                Where "Op" = "+ or - or .* or ./"
18    return C
```

6.2.3 Matrix Transpose

Another important operation required to perform matrix factorization is to transpose a given matrix. There are two ways we have handled this situation in this chapter, one way is to transpose the given matrix and store it as another matrix that occupies extra space, and another way to do it is to incorporate transpose during the required operation.

The multiplicative update algorithm contains matrix-matrix multiplication where either one of the matrices needs to be transposed. A way to achieve this operation would be to transpose the required matrix and use the algorithm mentioned in (Krishna et al., 2021), but this requires additional memory; here the additional memory is the transposed matrix. To avoid this issue, we perform an in-place transpose multiplication. This can be achieved by accessing the matrices with a different access pattern

$$A \times B^T = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{bmatrix} 5 & 0 & 2 & 3 \\ 3 & 0 & 0 & 5 \\ 0 & 0 & 2 & 4 \\ 0 & 1 & 2 & 0 \end{bmatrix} \end{array} \times \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{bmatrix} 2 & 4 & 3 & 1 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \\ 3 & 0 & 2 & 0 \end{bmatrix} \end{array} \tag{6.1}$$

$$\Rightarrow \begin{array}{c} r_0(A) \rightarrow \\ \\ c_0(B) \rightarrow \end{array} \begin{pmatrix} 5 \\ \times \\ 2 \end{pmatrix} + \begin{pmatrix} 5 \\ \times \\ 4 \end{pmatrix} + \begin{pmatrix} 5 \\ \times \\ 3 \end{pmatrix} + \begin{pmatrix} 5 \\ \times \\ 1 \end{pmatrix}$$

$$\Rightarrow c_0[C] = \{10\ 20\ 15\ 5\} \tag{6.2}$$

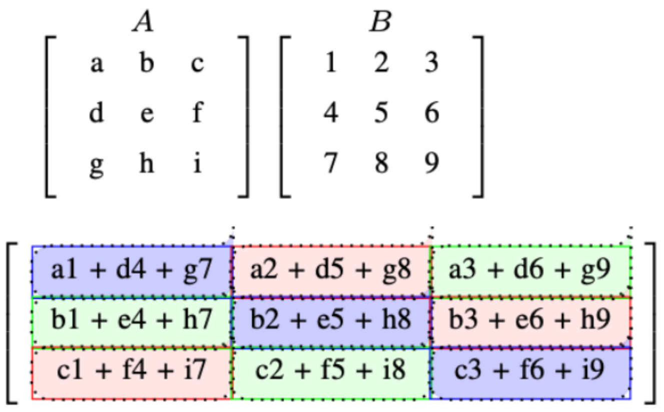


Figure 6.1: The working of $A^T \times B$, by storing the result in a pattern to eliminate the need to transpose the actual matrix.

Equation 6.2 shows an example of $A \times B^T$, where the partial resultant of column $c_0[C]$, is obtained after multiplying the first row $r_0[A]$ of A, and virtually transposed the first column of B, in this case, it is still $r_0[B]$.

Figure 6.1 shows the multiplication of $A^T \times B$ by virtually transposing A. Here, the colors along the diagonal show the order in which the resultant is obtained. Multiplying $r_0[A]$ with all rows of B, we obtain the main diagonal; continuing the process to the farther rows of A, we move the resultant to the upper triangle and wrap it around to the lower triangle, as shown in **red**, and **green**.

6.2.4 Sequence of matrix multiplications

Revisiting algorithm in (Krishna et al., 2021), where algorithms take two matrices as input and multiply them to produce the resultant matrix. However, data structures such as our novel versions of CBT and CSR are amenable to multiplying multiple matrices without storing the intermediate resultant matrix.

Algorithm 16 shows multiple matrix-matrix multiplication. Line 5 takes the output of line 3, the intermediate resultant row, and computes the resultant row on the third matrix. This process can be repeated through any number of input matrices. Therefore, this can be scaled to k as the number of matrices.

Figure 6.2 shows the pictorial representation of sequential multiplications of multiple matrices. A row of matrix A, A_i is multiplied by matrix B using the partial sum algorithm to obtain the intermediate resultant row Z_i, then Z_i is multiplied with the next matrix C to obtain the final resultant row D_i.

Algorithm 16: Matrix-Matrix Multiplication in Sequence

```
Input: Matrix $A$, Matrix $B$, Matrix $C$
Output: Resultant_Matrix $D$
1  begin
2      for row = 0 to numberOfRows do
3          aRow = getRow($A$, row)
4          tempArray = copmutePartialSum(aRow,$B$)
5          /* Call Algorithm in Sec 6.2.1 */
6          tempArray = computePartialSum(tempArray, $C$)
7          /* Call Algorithm in Sec 6.2.1 */
8          for nnzElements in tempArray do
9              $D$[row].streamEdge(nnzIndex)
10             $D.values$.bitPack(nnzValues)
11         /* If we are performing the matrix multiplication in $CSR$, then
             the number of non-zero elements in the resultant data for each
             row should be stored in C */
12     return $D$
```

6.2.5 Heuristic for faster convergence

One of the drawbacks of the multiplicative update approach is the convergence time and the iterations it takes to find an optimal solution. One of the ways to make the algorithm faster would be to reduce the number of non-zero values in the input matrix. If we are given a threshold number of index positions per row that

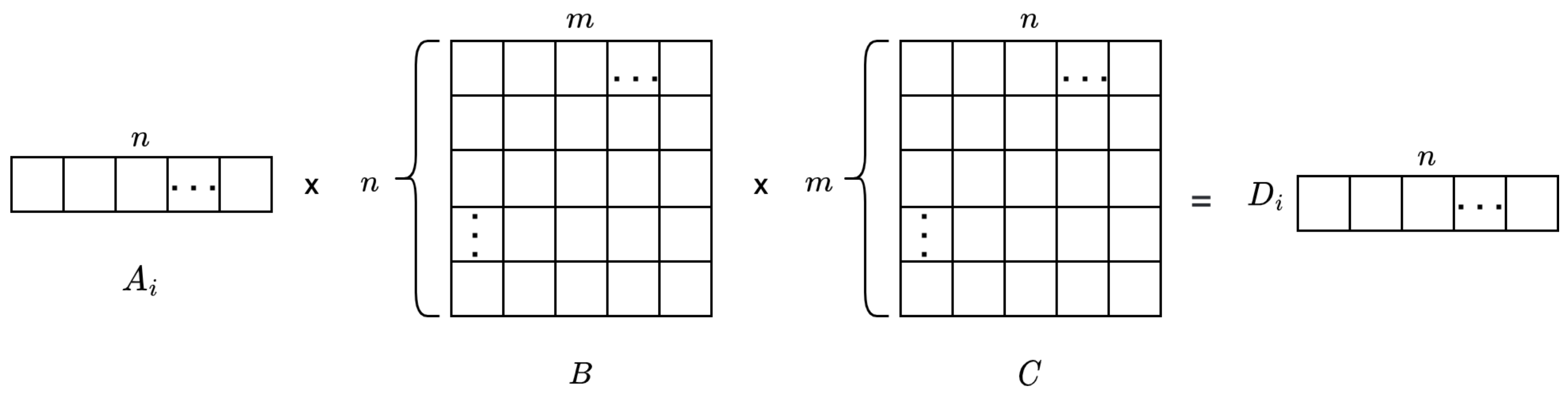


Figure 6.2: The working of sequential matrix multiplication, where a row of matrix A is multiplied with matrix B and matrix C, to obtain the resultant row of matrix D, using the partial sum multiplication.

can be made zero, we can come up with a heuristic approach to make specific values zero so that our compression is more efficient. One way to approach this is to remove the noise in the data; that is, we remove the data that do not contribute to the overall solution. This may lead to more loss, but the threshold will dictate the metric of the percentage of loss added to this already lossy factorization approach if we had not taken the heuristic approach. This will be a heuristic approach and will not be optimal. But it will lead to reduced resource utilization. Space is reduced in the already compressed structure and time to query the smaller CBT structure.

6.3 Experimental Results

This section evaluates matrix factorization on various matrices. For this experiment, we considered the variety of matrices with variable sparsity.

To factorize the matrices, we must first choose low-rank dense random W and H matrices. Choosing a low-ranking matrix leads to the formation of a smaller resultant matrix, which in turn consumes less space. Finding an optimal rank for factorization is a hard problem, as the algorithm has to go through the process of finding the number of orthogonal rows in the matrix. It is also more likely that the larger the inner dimension of the factors that we compute (W and H), the sparser these matrices will be, in which case CBT outperforms CSR even in terms of the storage of dense matrices. Therefore, in this chapter, we perform a brute-force analysis to obtain a minimal rank that would satisfy the criteria to reproduce the almost original matrix when W and H are multiplied.

Before we evaluate the algorithms on the other datasets, we first performed repeated factorization for a given matrix, since matrix W and H are random. In this experiment, we first considered a matrix of size $43,008 \times 43,008$, with $462,364$ nonzero elements, we were able to store the matrix using CBT in $4.44MB$.

We then ran the factorization repeatedly for any random matrices W and H. The resultant W and H were then multiplied to replicate the original matrices. In the end, we ended storing the $W \times H$ matrix in 4.21$\mp$0.073 MB, in 88.79$\mp$0.76 secs.

Table 6.1 shows the overall result of the computation performed in this chapter. The first set of columns in the table explains the basic details of the input, matrix dimensions in the first column, the number of non-zero elements in the second, matrix size when represented by using the $2-D$ matrix in the third, and the compressed sizes in the fourth and fifth respects. In the next part of the table, we present the inner rank of the factored matrices, followed by the result of $W \times H$ for both CBT and CSR, and the amount of memory required to process factorization at each iteration by CBT, CSR, and $2-D$ representation of the matrix.

In the results, one can notice that the memory required by the $2-D$ matrix is the highest. Still, the majority of the size is just the A matrix. Since the resultants can be streamed into a matrix in O(1) (constant), the extra memory used is very minimal. Still, as the inner rank increases, memory usage will increase accordingly.

However, when considering the two compressed structures, the proportion of memory consumed by CSR is much greater compared to the memory consumed by CBT (Gopal Krishna et al., 2021). This is due to the inability of CSR's to stream (add/delete), as the arrays need to resize, whereas CBT's ability to perform in-line operations, the advantage of one such operation is shown in the Figure 6.3, the figure compares the time taken to multiply three matrices in

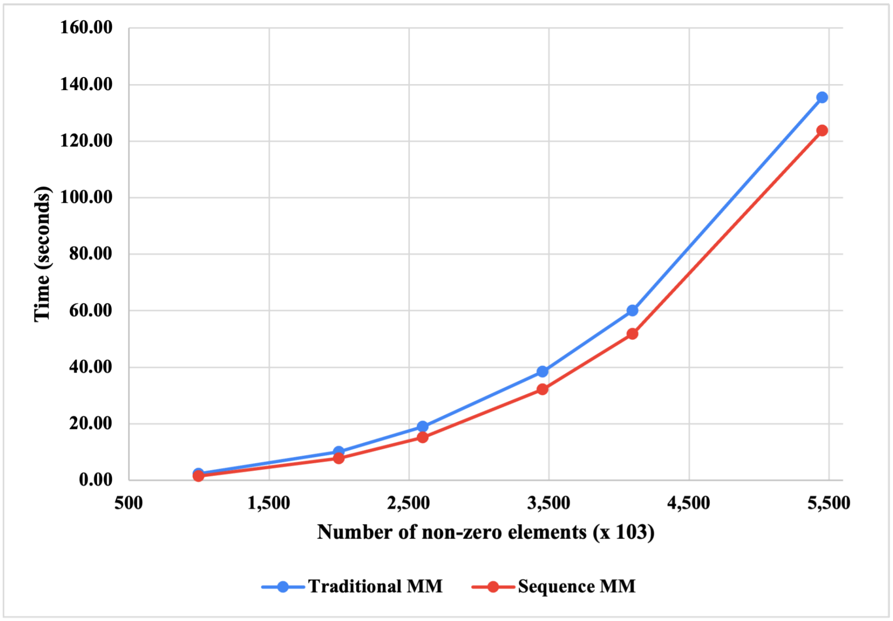


Figure 6.3: Comparison between the time taken to multiply three matrices in traditional two steps and uses our novel sequence multiplication in a single step for a Million-by-Million matrix.

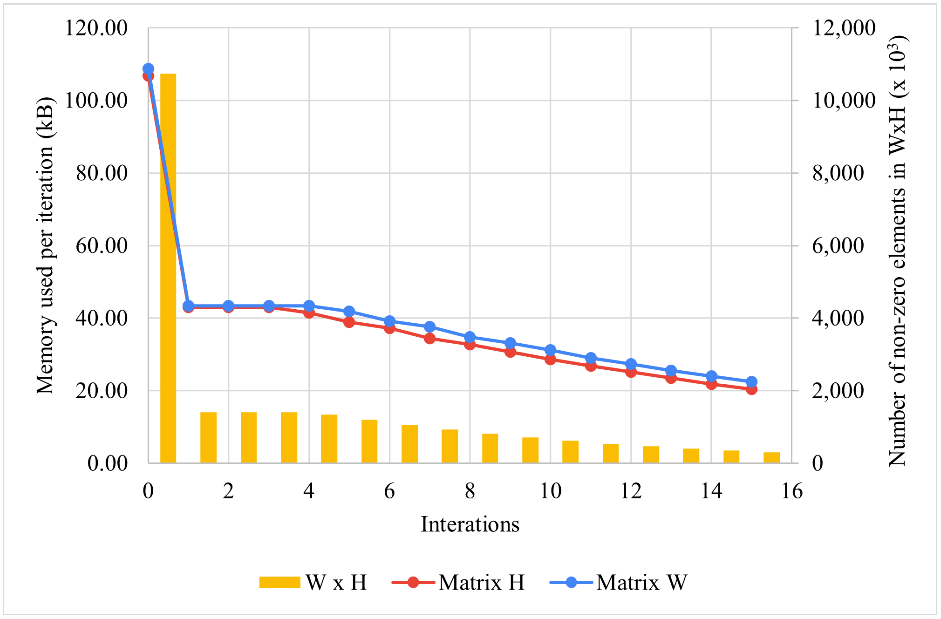


Figure 6.4: The evolution of W and H during the factorization for Matrix of size (21,504× 21,504, 1.36M nnz elements)

traditional two steps and uses our novel sequence multiplication in a single step for a Million-by-Million matrix of various levels of sparsity ranging from 1M to 5.4M elements. This memory usage will have a significant impact for a very large matrix, as shown in (Krishna et al., 2021).

Figure 6.4 shows the decrease in the memory required to store W and H as the iteration progresses, with the number of non-zero elements represented in the bars.

All experiments were run on an Intel(R) Xeon(R) W-2295 CPU @ 3.00GHz (16 Cores) with 64 GB of RAM, and the programs were written in GNU C/C++.

6.4 Summary

In summary, using our matrix multiplication algorithm, and proposal of other matrix operations required to a factorize matrix. We introduced a mechanism to solve the multiplicative update algorithm introduced by Lee and Seung (Lee and Seung, 2001) on the compressed structure. In this chapter, we also introduced multiple matrix multiplication, by eliminating intermediate resultant matrix.

Table 6.1: The factorization result using CBT and CSR and the memory required to process the factors.

						W × H		Avg Mem/Iter		
Matrix A	**NNZ**	**Matrix Size**	**CBT**	**CSR**	**Inner Rank**	***CBT***	***CSR***	***Matrix***	***CBT***	***CSR***
2688×2688	23,089	55.12 MB	217.36 KB	216.23 KB	448	216.58 KB	216.51 KB	73.5 MB	0.54 KB	0.67 MB
5376×5376	57,752	220.5 MB	547.53 KB	546.68 KB	255	513.87 KB	526.46 KB	241.41 MB	0.29 KB	30 MB
21504×21504	1,385,198	3.44 GB	12.7 MB	12.98 MB	512	12.65 MB	12.95 MB	3.6 GB	13.1 MB	150 MB
43008×43008	998,531	13.78 GB	9.45 MB	9.53 MB	670	9.1 MB	9.98 MB	14.21 GB	9.92 MB	87 MB
65536×65536	1,460,048	32 GB	14.23 MB	14.05 MB	665	13.45 MB	14.12 MB	32.64 GB	14.80 MB	200 MB

Chapter 7

Parallel Construction

The growing popularity of social networks and the massive influx of users have made it challenging to store and process the network/graph data quickly before the properties of the graph change due to graph evolution. Storing graphs or networks that represent entities and their relationships (such as individuals and their friends/followers in a social network) becomes more difficult as the number of users increases, resulting in massive graphs that are challenging to store in standard structures like matrices or adjacency lists. Research in this field has focused on reducing the memory footprint of these large graphs and minimizing the extra memory required for processing. However, there is a trade-off between time and space, as rigorous redundancy removal to achieve a small memory footprint consumes time, and querying becomes more time-consuming when traversing compressed structures compared to matrices or adjacency lists.

In this chapter, we introduce a parallel technique for constructing graphs using compressed sparse rows (CSR), which offers a smaller memory footprint and allows for parallel querying algorithms, such as fetching neighbors or checking

edge existence. We extend our work to include parallel time-evolving differential compression of CSR using the prefix sum approach. Additionally, we measure the speed-up gained by using multiprocessors to compress the graph data. To evaluate our techniques, we perform empirical analysis on massive anonymized graphs, including Live-Journal, Pokec, Orkut, and WebNotreDame, which are publicly available. Overall, our results demonstrate that our proposed methods achieve a smaller memory footprint and faster querying compared to traditional storage structures, with additional speed-up gained (up to 83% for the biggest graph with 3.07M nodes and 117.18M edges) through the use of multiprocessors.

7.1 Introduction

Graphs can represent real-world data from a wide variety of domains. The characteristics of the graph capture the relationships among the data. A graph is defined as $G = (V, E)$, where V is a non-empty set of nodes (vertices), and E is a set of relationships (edges). The most common examples of graphs that follow the definition would be a snapshot of a social network, transportation network, biological network, and infrastructure network. Since these graphs change in nature, the word snapshot captures a frame of the network at a particular point in time. Analyzing such graphs/networks offers a wide range of information, starting from how a user's influence would change his connections, the edge betweenness of the highways connecting major cities, analyzing the spread of infection, and designing an efficient routing algorithm based on the nature of the network. Efficiently answering these questions would be quite tedious as one has to deal with the balance between time and space required to process each one of these real-world networks.

In a real-world scenario, these graphs in a matrix format would require massive memory. For example, the Friendster network (Stanford Network Analysis Project, 2011), which contains 65 million nodes and 1.8 billion edges, requires about 30.02 Petabytes of storage space. This kind of storage availability is unheard of, and one way to address this issue is through compression. But once the data are compressed, the data should also be amenable to queries without completely decompressing. So, to come up with a good storage system that queries the graph without decompressing it has been proposed by (Boldi and Vigna, 2004; Nelson et al., 2017, 2018; Caro et al., 2016; Chierichetti et al., 2009; Gopal Krishna et al., 2021).

However, all these compression techniques proposed take time to compress the data. To speed up the process, in this chapter, we introduce a parallel algorithm to construct one of the most commonly used graph data structures, Compressed Sparse Row (CSR) (Tinney and Walker, 1967). Along with the compression, we also introduce graph querying techniques, where multiple queries can be performed in parallel.

A time-evolving graph is a graph that changes over time and can be represented using a series of graphs at different instances. A graph $G_t = (V_t, E_t)$ where time t indicates an instant that is spread over a certain interval. For instance, the pages on Wikipedia change over time with the addition and deletion of content, and the edited information is saved, allowing the preservation of the page's integrity while remaining open to editing. The information for such pages with time-evolving data can be stored as graphs, which can be useful for various kinds of analysis. However, most real-world graphs are large, and the memory requirements to store the data can be significant. Therefore, the data must be compressed to fit in the main memory for analysis.

The data for time-evolving graphs can be stored in three different representations: adjacency matrix, adjacency list, and edge list. Descriptive, diagnostic, predictive, and prescriptive analyses can be performed on such graphs based on the availability of the data over time. However, with large sizes of real-world graphs such as Wiki-edits and Yahoo Netflow, which are 5.7 GB and 19 GB in edge list format, respectively, storing and analyzing the data can be challenging. Therefore, compressing the data is necessary to store and perform computation on time-evolving graphs.

The contributions are as follows.

- We provide a parallel novel implementation to compress a given edge list into CSR in Section 7.3.
- A parallel prefix sum calculation approach is used to parallelize the construction of a cumulative degree array in Section 14.
- The algorithm provided in Section 14, computes the degree of each node concurrently.
- Algorithms to parallelly compress a time-evolving graph stored as a CSR are provided in Section 7.4.
- The neighbor query algorithm that is performed in parallel to get a set of neighbors, given a CSR is explained in section 7.5.1.
- Two approaches to query edge existence(Section 7.5.2):
  - Given an array of edge existence queries, perform subset queries in parallel.
  - Given a single query that parallelly accesses CSR.

- We evaluate (Section 7.6) the construction of CSR with respect to the number of processors, and experimented with inputs of million-scale social networks.

So rest of the chapter is organized as follows. In Section 7.3 explains the procedure to parallelly compress the graph. In Section 7.5, we explain the querying algorithms on the compressed structure, and in Section 7.6, we evaluate our structure with the publically available social networks, then we conclude our work in Section 7.7.

7.2 Related Work

The Compressed Sparse Row (CSR) (Tinney and Walker, 1967) data structure is widely utilized for graph representation. CSR involves compressing each row of the graph into two arrays for each node, allowing efficient packing of all the necessary information into a single array for fast traversal of the data structure. Figure 7.1 shows the CSR representation of the graph shown in Table 7.1. While CSR has the disadvantage of being a static storage format that can require shifting the entire edge array when adding an edge, its cache-friendliness inspired the development of Packed Compressed Sparse Row ($PCSR$) (Winter et al., 2017). $PCSR$ substitutes the edge array in CSR with a Packed Memory Array (PMA) (Itai et al., 1981) , (Bender et al., 2000), which offers an (amortized) $O(log_2|E|)$ update cost and asymptotically optimal range queries. In this chapter, we do not take the packed CSR route to compress the given graph.

Calculation of the prefix sum is one of the crucial steps in the construction of CSR for the calculation of the degree array. The prefix sum operation (Blelloch, 1990; Wheatman and Xu, 2021) takes an array A as input of length $n$and outputs an array A' where $\forall i \in \{0, 1, ...n-1\}$,

$$A'[i] = \sum_{j=0}^{i} A[i]$$

There have been parallel in-place algorithms for finding prefix sum (Blelloch, 1990) that take $O(n)$ work and $O(logn)$ in time. Because of the high dependency on computing parallel degree arrays in CSR, there are many challenges involved. Parallel Packed Compressed Sparse Row ($PPCSR$) (Wheatman and Xu, 2021), designs and analyzes a parallel PMA approach and compares it to other similar approaches (Shun and Blelloch, 2013; Dhulipala et al., 2019; Shun et al., 2015).

One way to represent a time-evolving graph is by using a sequence of static graphs, where each graph represents the state of the graph at a specific point in time. These individual graphs, also called snapshots, can be represented as 2D matrices. By stacking these matrices along a third dimension, we can construct a 3D matrix, commonly referred to as a *presence matrix* according to (Ferreira and Viennot, 2002).

Caro et al. introduced $ck^d - trees$ in 2016 (Caro et al., 2016). They define a contact as a quadruplet (u, v, t_i, t_j) and use this to compress the 4D binary matrix representing the time-evolving graph. This is achieved by treating the 4D matrix as a $k^d tree$ and differentiating between white nodes, which have no contacts, black nodes, which only have contacts, and gray nodes, which have only one contact. This approach is based on the work of Brisaboa et al., who introduced $k^2 - trees$ in 2014 (Brisaboa et al., 2014b).

The G* database (Labouseur et al., 2015) is a distributed index that addresses the space issue of the presence matrix by storing new versions of an arc as a log of changes instead. It accomplishes this by storing versions of the vertices as adjacency lists and maintaining pointers to each time frame. Whenever an arc changes in the next frame, a new adjacency list is created for that vertex's arc, and a pointer is added to the new frame. DeltaGraph (Khurana and Deshpande, 2013) is another distributed index that groups the different snapshots in a hierarchical structure based on common arcs.

EveLog (Caro et al., 2015) is a compressed adjacency log structure based on the "log of events" strategy, consisting of two separated lists per vertex, one for the time frames and another for the arcs related to the event. The time frames are compressed using gap encoding, and the arc list is compressed with a statistical model. However, query times suffer because the log must be scanned sequentially. To determine if an arc is active at a particular time frame in the log strategy, it is necessary to sequentially read the log of events (possibly deactivating/reactivating the arc) until the time frame is reached. This approach is slow for large time-evolving graphs since it takes linear time. Ferreira et al. (Ferreira and Viennot, 2002) follow this strategy by providing a quadruplet $(u, v, t, state)$ for each time an arc changes. In (Bui-Xuan et al., 2002), the authors present a data structure of adjacency lists where each neighbor has a sublist indicating the time intervals when the arc is active to improve query times. EdgeLog (Caro et al., 2015) compresses this idea using gap encoding.

In (Ren et al., 2011), the FVF (Find-Verify-Fix) framework is developed, which includes a copy+log compression that also supports shortest paths and closeness centrality queries. Three different methods to index time-evolving graphs based on the copy+log strategy are described in (Bernardo et al., 2013) and (Álvarez-García et al., 2014).

Two "log of events" strategies, CAS and CET, are proposed in (Caro et al.,

2015) to address the problem of slow query times when processing a log. CAS orders the sequence by vertex and adds a Wavelet Tree (Grossi et al., 2003) data structure to allow for logarithmic time queries. CET orders the sequence by time and develops a modified Wavelet Tree called Interleaved Wavelet Tree to also allow logarithmic time queries.

In 2014, Brisaboa et al. (Brisaboa et al., 2014a) adapt compressed suffix arrays (CSA) (Caro et al., 2015) for use in temporal graphs (TGCSA) by treating the input sequence as the list of contacts. An alphabet consisting of the source/destination vertices and the starting/ending times is used.

7.3 Compressed Sparse Row

The compressed sparse row, known as CSR, is one of the most common data structures for storing a graph. CSR was first introduced by Tinney et al. in 1967 (Tinney and Walker, 1967). Since then, the structure has been an integral part of research in the area of data storage. The representation consists of three arrays,

- iA: indicates the number of non-zero elements that are present in a row
- jA: indicated the column number where the non-zero element is present in that particular row
- vA: a value array (if the graph is weighted).

If the graph is unweighted, we ignore the third array since an unweighted array is also a boolean array.

Table 7.1: An example of a 10-node sparse graph.

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

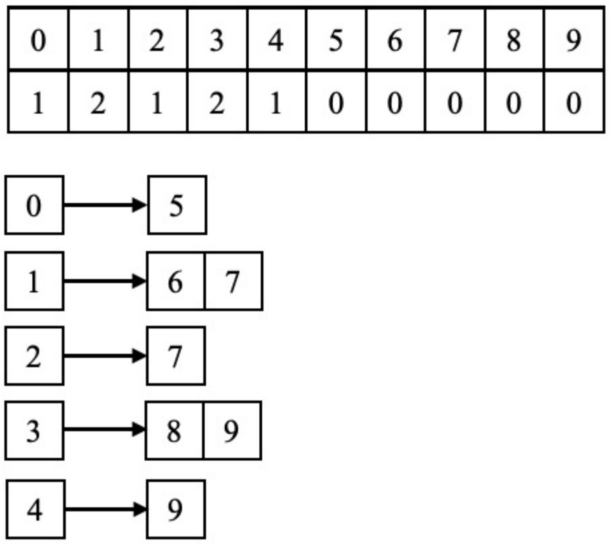


Figure 7.1: Compressed Sparse Row representation of the upper triangular matrix of the shown graph as degree array and neighbor list

7.3.1 Parallel Construction for CSR

Before we discuss the construction of CSR, we first explain one of the pivotal algorithms used in the construction which is Parallel Prefix Sum by Guy Blelloch (Blelloch, 1990). At first, we describe a way to use the prefix to compute the degree array, and later in the chapter, we also discuss the concept of prefix sum to construct time-evolving parallel CSR or $TPCSR$.

Algorithm 17: Parallel Prefix Sum calculation

```
Input: An array of unsigned integers $vec$, $startI$, $endI$
Output: Prefix sum calculated for $vec[startI : endI]$
begin
    for $i = startI + 1$ to $endI$ do
        $vec[i] += vec[i-1]$
    sync() // synchronize all parallel processors
    // lock to add and carry over the last prefix value to each processor
    Lock()
        if $startI > 0$ then
            $vec[end-1] += vec[start-1]$
    Unlock()
    sync()
    if $startI > 0$ then
        for $i = startI$ to $endI$ do
            $vec[i] += vec[start-1]$
    return $vec$
```

Prefix Sum Computation

Algorithm 17, of the Parallel Prefix Sum calculation, also known as the Scan algorithm. This algorithm is used to calculate the prefix sum of an input array in parallel, meaning that it can be executed by multiple processors or threads simultaneously.

Given an array of unsigned integers, *vec*, and two indices *startI* and *endI*, which define the range of the array to operate on as inputs, this range is referred to as a chunk. The output of the algorithm is the prefix sum calculated for the elements in the specified range. The algorithm starts by iterating over the elements in the input array, starting at index $startI + 1$ and ending at index $endI$. For each element i, the value of $vec[i]$ is updated by adding the value of $vec[i-1]$ to it, thus calculating the prefix sum up to that point, as shown in lines 2-3.

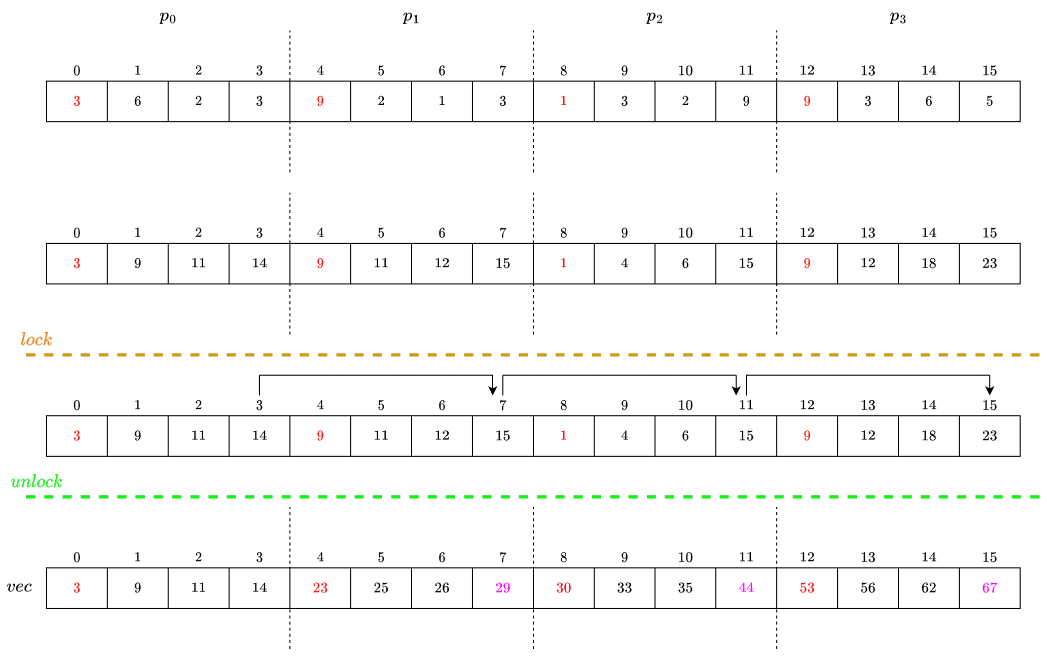


Figure 7.2: Shows the steps for obtaining the prefix sum of an array in parallel: The first array is shown in the input, and the dotted lines show the chunks. The second array shows the prefix sum calculated per chunk. The third array, after locking the last element in each chunk is sequentially added to the last element of the next chunk. After unlocking, the processors in parallel add the last element of the previous chunk to all but the last element in the current chunk.

Once the update to the chunk is completed, the algorithm synchronizes all parallel processors. This ensures that all processors have completed their updates and are ready to proceed, lines 4-5. Now, we lock the execution and add the last prefix value to each processor. These updates happen for chunks not starting from 0, as chunk 0 will have no dependency. This is necessary to carry over the prefix sum value from the previous range, lines 6-7. After the process finishes adding the corresponding values, the execution unlocks and synchronizes once again to proceed further to line 8. Finally, all processors except the first pick the previous processor's last element and add it to the range in its chunk of the array. This process is repeated until $endI - 1$ since the end value was updated in the previous step, lines 9-11. The algorithm returns the modified input array, which now contains the prefix sum values for that given range, line 12.

Figure 7.2 shows the detailed visualization of the work of the prefix sum.

Degree Computation

The degree array (iA) in CSR often stores the starting index of each row. To compute the starting index, one must first compute the degree of each node and then compute the sum. To compute the sum sequentially, in the worst case, one would require $O(n^2)$ units of time, where n is the number of nodes. But to avoid spending $O(n^2)$ units of time, we use the prefix sum algorithm, which is set to solve in $O(logn)$ units of time with $O(n)$ processors.

To compute the degree of our graph, we first split the given array of size n into p chunks, p being the number of processors. Then each processor takes in a chunk of data to compute the occurrences of each element in that particular array, and write it into the $globalDegArray$, which contains the degree of each node. To avoid the concurrency which might occur when two processors are competing to write the degree of same node, we construct a $tempGlobalDegree$ array of size p, then finally merge $globalDegArray$ and $tempGlobalDeg$.

The construction of the degree array is split into two algorithms.

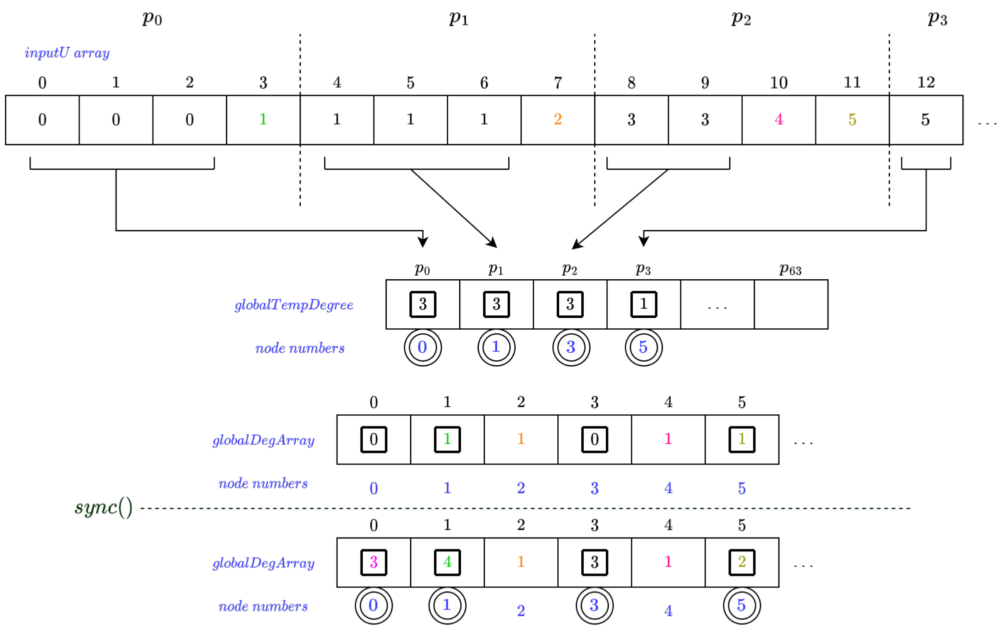


Figure 7.3: Shows the working of degree computation in parallel: The first array shown is the input array. The frequency of the node in each chunk is stored in a global temporary degree array. The frequency of the remaining nodes in each chunk is stored in the global degree array. After all, processors finish the degree count for their chunk, the processors are synchronized to ensure that all global degree arrays are up-to-date. Then, we add the frequency of the first appearing node of each chunk to their corresponding degree in the global degree array.

Algorithm 18 computes the degree of a chunk and stores the result in a global array. The algorithm starts by computing the node numbers for the start and end of the chunk. This is done by dividing the array into smaller nodes or subarrays and assigning node numbers to each of them. Then, the variable *uStart* is assigned the node number for the start of the chunk, and *uEnd* is assigned the node number for the end of the chunk.

Algorithm 18: Computation of degree array per chunk

```
  Input: An array of unsigned integers $A$, $startI$, $endI$
1 begin
2     $uStart$ = node number at the start of chunk
3     $uEnd$ = node number at the end of chunk
4     $globalTempDegree[A[uStart]]$ = Count the number of consecutive
        occurrences of the first node in each chunk and store it in a
        secondary global degree array.
5     for $i = startI + 1$ to $endI$ do
6         $nodeI = A[i]$
7         /* denotes looping through each value in $A$ */
8         $globalDegArray[nodeI]$ = count number of consecutive
            occurrences of $nodeI$
```

Next, the algorithm counts the number of consecutive occurrences of the first element in the chunk, which is $A[uStart]$, and stores it in a secondary global degree array called *globalTempDegree*. This is done by looping through the chunk and incrementing a counter variable every time the first element is encountered consecutively.

After that, the algorithm enters a loop that iterates through each element in the specified range, starting from the second element ($A[startI + 1]$). For each element, the algorithm counts the number of consecutive occurrences of the element and stores it in the global degree array called *globalDegArray*. This is done by looping through the chunk and incrementing a counter variable every time the current element is encountered consecutively.

Finally, the algorithm terminates after looping through all the elements in the specified range and counting the number of consecutive occurrences of each element in the global degree array.

Algorithm 19, builds the degree array for the CSR structure. Figure 7.3, shows how we merge the $globalDegArray$ with the $globalTempDegree$ array. Since each chunk receives a sorted list of edges, it is for sure that there would only be at most one overlap between two processors.

For example, in the figure, the chunk 1 has received edges from nodes $0-1$, the chunk 2 has received edges from nodes $1-2$, the chunk 3 has received edges from nodes $3-5$, and finally, chunk 4, has element 5. Each processor in the algorithm will always save the frequency of the first element in the temporary global degree array, and the remaining elements are written directly to the global degree array. Once all processors have finished, they synchronize their computations using the **sync()** function to ensure that all the global degree arrays are up-to-date.

After that, each processor updates the global degree array $globalDegArray$ by adding the temporary degree count $globalTempDegree[pid]$ for the first element in its chunk, which is calculated as $A[pid * chunkSize]$. This is done to account for the overlap between consecutive chunks in the CSR format.

Finally, the algorithm returns the global degree array $globalDegArray$, which is needed to represent the degree of each node in the graph in CSR format.

Build CSR

Once we retrieve the degrees of all nodes in the graph, these graphs are now compressed. For further compression, we are using our novel technique to store the integer numbers associated with both the degree array iA, and the edge

```
Algorithm 19: Build $CSR$ degree array
   Input: An array of unsigned integers $A$, $p$ number of processors
   Output: Degree array $degArr$
1  begin
2      $globalDegArray$ // an array of size $n$
3      $globalTempDegree$ // an array of size $p$
4      $chunkSize = n/p$
5      do in parallel:
6          call Algorithm 18 for each chunk
7          sync()
8          $globalDegreeArray[A[pid * chunkSize]] += globalTempDegree[pid]$
9      return $globalDegreeArray$
```

column array jA.

```
Algorithm 20: Build bitPacked $CSR$
   Input: EdgeList, $p$ number of processors
   Output: BitPacked $CSR$
1  begin
2      do in parallel: for each processor
3          $chunkSize$ = Compute the chunk based on the processor
             availability.
4          call bitPack algorithm from (Gopal Krishna et al., 2021), for each
             chunk.
5          The resultant bit array is then stored in a global location.
6      $finalBitArray$ = merge all bitArrays from global location
7      Repeat the process once for degree array $iA$,
8      and once for edge column array $jA$ return $CSR$
```

Similarly to our previous approach of diving the array into chunks and providing them to each processor, we continue the same approach to call our bit packing algorithm mentioned in (Gopal Krishna et al., 2021), to compress the CSR. We repeat this process separately for the degree array and again for the edge column array, as shown in Algorithm 20.

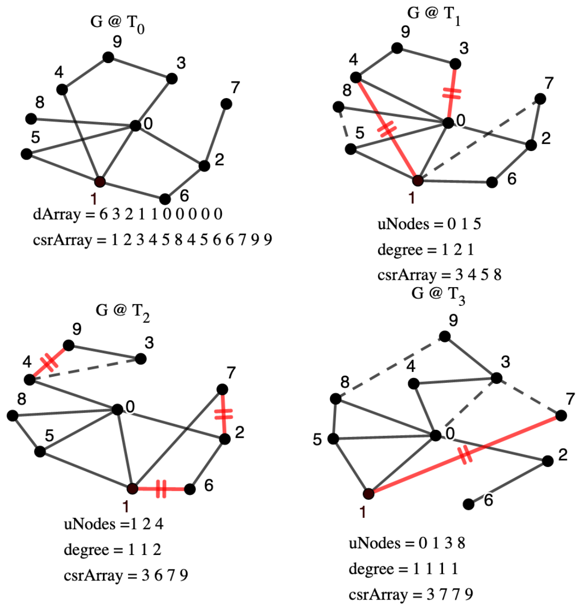
7.4 Parallel Construction of Time-Evolving CSR

For every input of the time-evolving graphs G, the input is divided as an ordered triplet (u, v, T_τ), where u and v are the nodes that form an edge at time T_τ. If the edge appears again later in another time frame $T_{\tau+i}$, the edge is considered to be deactivated in the time frame. We assume that the datasets are sorted with respect to the time frames and then sorted by node numbers for each time frame.

Figure 7.4 shows the design of how storing a differential time-evolving graph works. The graph shown in the figure evolves for 4 time frames in every time frame, we could either see an edge being added or an edge being deleted or no change. To illustrate, we have shown the edge being deleted in red color and the edge being added as a dotted line. For time frame T_0 (first time frame), we construct the CSR with both the degree array and the column index array, and for the following time frames, T_i, we store the difference in the graph with respect to the time frame $T_{(i-1)}$.

Since the process is serial and has dependency over the previous time frame, constructing a time-evolving graph in parallel would need a different approach. Figure 7.5, shows the working on parallelly constructing time-evolving CSR. The input to the compression method is a time-sorted edge list. Therefore, we now divide the entire edge list. Once divided, we compute CSR on these chunks. Note that there could be an overlap similar to that of computation of degree in Section 14. Similarly to degree merging, we merge the overlapped CSR to obtain one CSR for every time frame.

Storing the CSR this way is space-consuming, as not all nodes have changed



CSR as unsinged char: 0110110001001000100 10000100110000101010001001010111000010011000010101 0 000100101 100010100 110000101010001 100010001 100100010 0010111000010101000 0000100011000001 1111 1100111011101001

Figure 7.4: Captures the graph evolving over 4 time frames. The edges in red show the edge being deleted from one time frame to another, and the dotted edge indicates the edge being added.

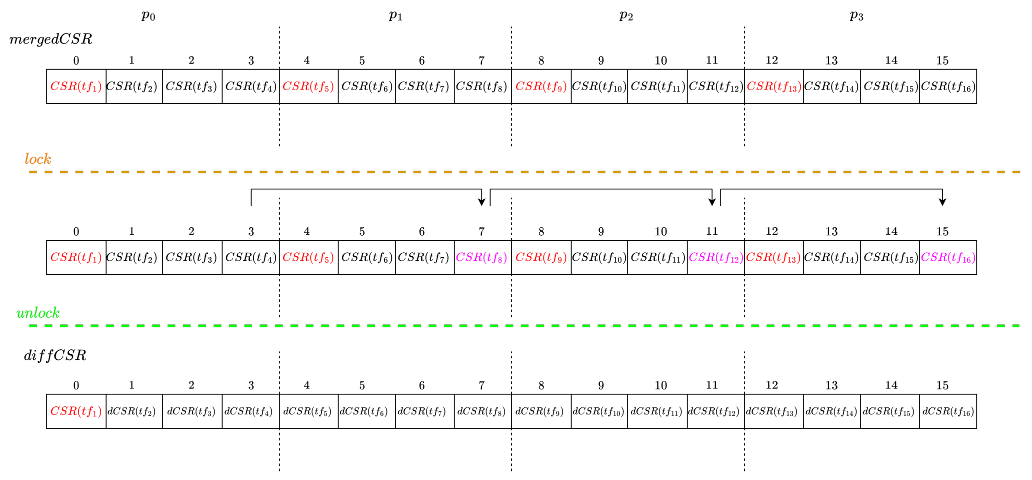

p_0
p_1
p_2
p_3
$mergedCSR$
$CSR(tf_1)$ … $CSR(tf_{16})$
$lock$
$unlock$
$diffCSR$
$CSR(tf_1)$ $dCSR(tf_2)$ … $dCSR(tf_{16})$

Figure 7.5: Shows the construction of time-evolving differential CSR using Prefix Sum, which is used to compute the difference in the consecutive time frames. This follows a similar approach to Figure 7.2.

state from one-time frame to another. Therefore, in the next step, we perform a differential operation parallelly. To perform this differential operation, we seek to prefix the sum computation. Now, we divide the array of $CSR's$ into chunks and process the differences. The first time frame in every chunk is kept as is, and the differences are computed on the remaining $CSR's$ in the same chunk.

Once we find the differences in each chunk, we perform the sync and lock operation to propagate the end difference. Once the end differences are propagated, we unlock and resync to perform the final differential operation, similar to prefix sum.

The differences here are the edges added or deleted in each time frame. Within a given time interval, if an edge appears an even number of times, the edge is set to be inactive, and if the count is odd, then the edge is set to be active. The working of $TCSR$ is shown by Gopal et al. (Gopal Krishna et al., 2021).

Algorithm 21: Build $TCSR$ degree array

```
  Input: EdgeList with time-intervals, $p$ number of processors
  Output: Degree Array $degArrT$ as differential $TCSR$
1 begin
2     do in parallel:
3         Divide the input edge list, and construct $CSR$ for each time
            frame in the chunk.
4         Merge overflowing $CSR's$ between chunks
5         Perform differential $CSR$ for every time frame using the prefix
            sum algorithm.
6     return BitArray $TCSR$
```

7.5 Parallel Querying Algorithms for CSR

One of the most important operations on a social network is about getting to know if there is a connection between two individuals or checking who are all the acquaintances of a given user. These operations translate to checking if there is an edge between two nodes and fetching all neighbors of a given node.

These two searches are performed quite frequently, which means that we can search for multiple queries at once. For a social network with millions and billions of users that are using it at once, it is quite time-consuming to perform one query at a time. Instead, if multiple processors involve querying multiple items at once, the time required to search reduces.

In this chapter, we propose two querying algorithms, one is to perform an array of neighborhood queries in parallel, and the second one is to perform an array of edge existence queries in parallel. In addition, we propose a quicker way to query the edge existence by splitting the bit array of a node into multiple chunks and making multiple processors check for the edge.

7.5.1 Neighborhood Query

The first querying algorithm we perform is neighborhood querying. Given an array of queries (node numbers $uNodes$), an array of unsigned bits A, the start and end indices in $uNodes$, and the number of bits $numBits$. The $numBits$ tells us the number of bits to process in the bit array to obtain a node number.

The algorithm starts by iterating through the range of nodes in $uNodes$ from the specified start index $startI$ to the end index $endI$. For each node $uNodes[i]$ in this range, the algorithm calls the $GetRowFromCSR$ function mentioned in (Krishna et al., 2021), passing in the array of unsigned bits A, the starting index of the node $uNodes[i].startingIndex$, the degree of the node $degrees[uNodes[i]]$, and the number of bits $numBits$.

The $GetRowFromCSR$ function takes as input a compressed sparse row

(CSR) representation of the graph and returns the row corresponding to the specified node. This row contains the indices of the node's neighbors in the graph.

The row of neighbors returned by *GetRowFromCSR* is then assigned to the corresponding element in the resultNeighbors vector, using the node number as the index.

Algorithm 22: Get neighbors of $uNodes$

```
Input: An array of unsigned bits $A$, an array of $uNodes$ containing
       neighbor queries, $startI$ index in $uNodes$, $endI$ index in
       $uNodes$, and the number of bits $numBits$
Output: Vector of vectors $resultNeighbors$ of all the neighbors of
        $uNodes$ from $startI$ to $endI$
1 begin
2     for $i$ = $startI$ to $endI$ do
3         /* denotes looping through one neighbor from $uNodes$ */
4         $resultNeighbors[uNodes[i]] = GetRowFromCSR(A,$
            $uNodes[i].startingIndex, degrees[uNodes[i]], numBits)$
```

Finally, once the loop ends and the queries are completed, the vector *resultant-Neighbors* will be left with an array of neighbors. During the call to this method, the array of queries is split among processors, and the chunks of neighbors are obtained at once.

7.5.2 Edge Existence

In this section, we are going to discuss two types of edge existence; the first one being, given an array of edges, query the existence, and the second one being, given an edge, divide the array into chunks and process for existence.

Algorithm 23, takes an array of unsigned bits A, an array of edges *edges* containing edge queries, *startI* index in *edges*, *endI* index in *edges*, and the number of bits *numBits* as input, and outputs the existence of edges between all pairs of nodes u and v in *edges*.

The algorithm iterates through each edge query in the given range of indices from *startI* to *endI*. For each edge query, it first retrieves the neighbors of the source node u in the given array A using the function *GetRowFromCSR* function mentioned in (Krishna et al., 2021) which takes A, the starting index of the row of the source node, the degree of the source node u and the number of bits *numBits* as input and returns the list of neighbors of the source node.

Next, the algorithm iterates through each neighbor of the source node u and checks if it is equal to the target node v. If a match is found, the algorithm outputs the presence of an edge between the nodes u and v.

Algorithm 23: Edge existence of an array of edges

```
Input: An array of unsigned bits $A$, an array of edges containing edge
       queries, startI index in edges, endI index in edges, and the
       number of bits numBits
Output: Existence of edge between $u$ and $v$ for all edges
1 begin
2     for $i$ = startI to endI do
3         uNeighs = GetRowFromCSR(A, edges[i].startingIndex,
            degrees[edges[i]], numBits)
4         for $s_n$ in uNeighs do
5             /* denotes looping through each neighbor of $u$ */
6             if $s_n == v$ then
7                 output presence of edge $(u, v)$
```

Overall, the algorithm checks for the existence of edges between all pairs of nodes u and v in the given range of edge queries. It does this by retrieving the neighbors of the source node for each edge query and then checking if the target node is present in the list of neighbors.

For the algorithm 24, we narrow down the search space for an edge query by first retrieving the neighbor of u and splitting the neighbor array among processors to search for v. This could also be extended to a binary search to speed up the process.

```
Algorithm 24: Single Edge Existence
  Input: An array of neighbors of $uNeighs$, $startI$ in $A$, $endI$ in $A$, node $v$
  Output: Existence of $v$ in $uNeighs$
1 begin
2     for $s_n$ in $uNeighs$ within the range do
3         /* denotes looping through each neighbor of $u$ */
4         if $s_n == v$ then
5             output presence of edge $(u, v)$
```

So far, we have seen how each of the querying algorithms works. However, the work lies in the design of the call to algorithms parallelly. Algorithm 25 shows the call to all querying algorithms parallelly.

The first parallel do, explains the call to Algorithm 22, the algorithm fetches all neighbors associated with the input array of nodes by splitting the input array into p parts. Once the input is split, each processor takes in the compressed CSR and the start and end of the query array to fetch all the neighbors. The result for every node queried will be returned as an array of arrays with all the neighborhood information.

The second parallel do, explain the call to Algorithm 23, where given an array of edge inputs, how do we parallelly see if the edge exists or not? The approach is similar to a neighborhood query, where we divide the array into p chunks and divide the input amongst multiple processors. Each processor then reads the input and processes the adjacency list associated with the edge to see if the edge exists.

The third parallel do, explain the call to Algorithm 24, where given a single edge (u, v), query to check if the edge exists in parallel. For this, we first retrieve the neighborhood list of the node u, then split the list into p parts, and parallelly subject each processor to look if v exists in the given chunk.

Overall, this algorithm is designed to efficiently query a compressed CSR data structure in parallel by dividing the work among multiple processors. By doing this, the algorithm can speed up the process of querying large matrices or graphs.

Algorithm 25: Call to Querying Algorithms

```
   Input: Compressed $CSR$
 1 begin
 2     // Given an array of nodes, and $p$ processors,
 3     // fetch all neighbors associated with these nodes.
 4     do in parallel:
 5         Split the input array into $p$ parts, and call Algorithm 22
 6         This algorithm takes in the $CSR$, and the start and end of the
             query array for each processor
 7         The result for every node queried will be returned as an array of
             arrays with all the neighborhood information.
 8     // Compute edge-existence given multiple edges
 9     do in parallel:
10         Give an array of edges to be queried,
11         Split the edge array into to $p$ parts,
12         Call Algorithm 23, with the query array and array starting and
             ending index.
13     // Given the list of adjacency for a node $u$, and $p$ processors
14     // To see if $v$ exists.
15     do in parallel:
16         Split the list into $p$ parts, and call Algorithm 24
17         This algorithm takes in the starting and ending index of the list
             along with the node to be searched.
18         One of the processors will return true if the edge exists,
19         If not all return false
```

Table 7.2: Shows the experiments performed on the different types of graphs.

Graphs	**# of Nodes**	**# of Edges**	**EdgeList Size**	**CSR**	**# of Processors**	**Time (ms)**	**Speed-Up (%)**
LiveJournal	4,847,571	68,993,773	1.1 GB	24.73 MB	1	164.76	-
					4	57.94	64.83
					8	48.35	70.65
					16	40.09	75.67
					64	17.613	89.31
Pockec	1,632,803	30,622,564	405 MB	197.83 MB	1	67.41	-
					4	28.19	58.18
					8	20.95	68.92
					16	18.21	72.99
					64	6.53	90.31
Orkut	3,072,627	117,185,083	1.7 GB	313.19 MB	1	235.52	-
					4	75.09	68.12
					8	58.38	75.21
					16	55.15	76.58
					64	38.09	83.83
WebNotreDame	325,729	1,497,134	22 MB	3.82 MB	1	7.13	-
					4	2.02	71.67
					8	1.1	84.57
					16	0.577	91.91
					64	0.27	96.21

7.6 Experimental Evaluation

In this section, we evaluate the performance of parallel CSR. For evaluation, we have considered publicly available social networks provided by Stanford SNAP (Stanford Network Analysis Project, 2011).

Table 7.2 shows the compression result on various numbers of processors. The first three columns explain the properties of the graphs, the number of nodes, and the number of edges present in each graph. The fourth column shows the space required to store the graph if the graph is stored in an edge list. The size might seem small compared to storing the graph in a matrix. However, the edge list consumes more time in querying compared to CSR. Therefore, in the fifth column, we have the space required to store the same graph in bit packed CSR. The sixth column shows the different numbers of processors on which the graph was tested; if the number of processors is equal to 1, then the algorithm is said to run in serial mode. Therefore, the seventh column shows the time required to process the same graph when there are a different number of processors working to achieve the resultant CSR. The final column shows the speed-up gained using multi-processors over a single processor, measured in percentage (%), as shown in figure 7.7.

Figure 7.6 shows the pictorial representation of the time taken to compress the graph versus the number of processors used to compress it.

A rapid decline is seen when going from 1 processor to 4, then a steady decline with 8 and 16, followed by a decent drop in time with 64 processors. The steady decline within multiple processors is due to the inherent sequential steps.

All experiments were run on AMD Ryzen Threadripper 3970X 32-Core Processor, with 128 GB of memory, and the programs are written in GNU C++17.

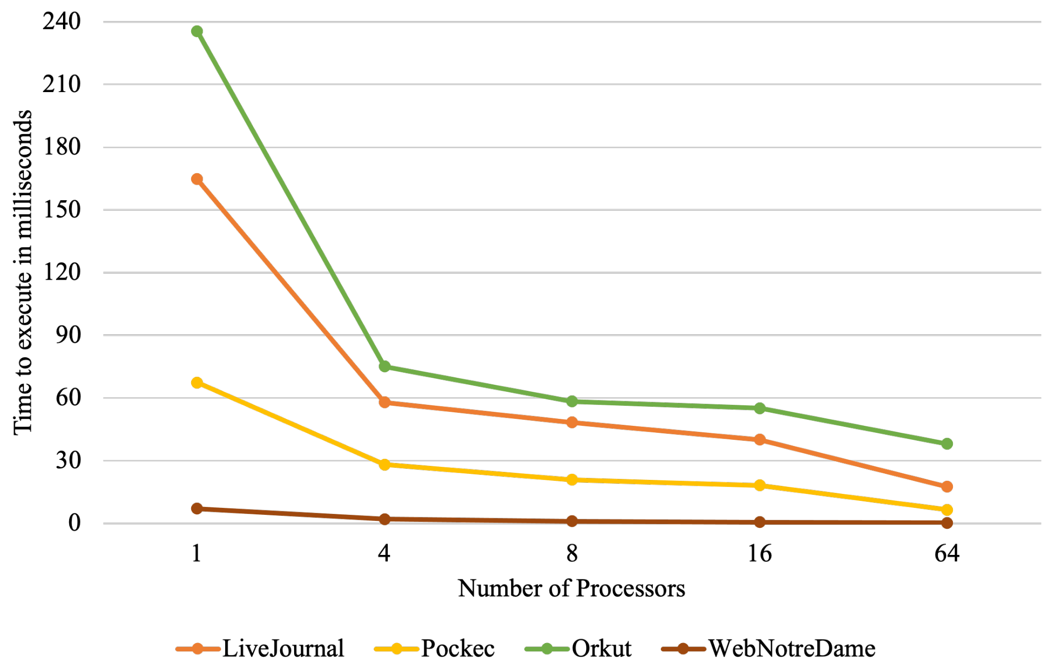


Figure 7.6: Shows the execution times for different number processors for various graphs. We see the time taken to construct CSR decreases significantly when parallelized.

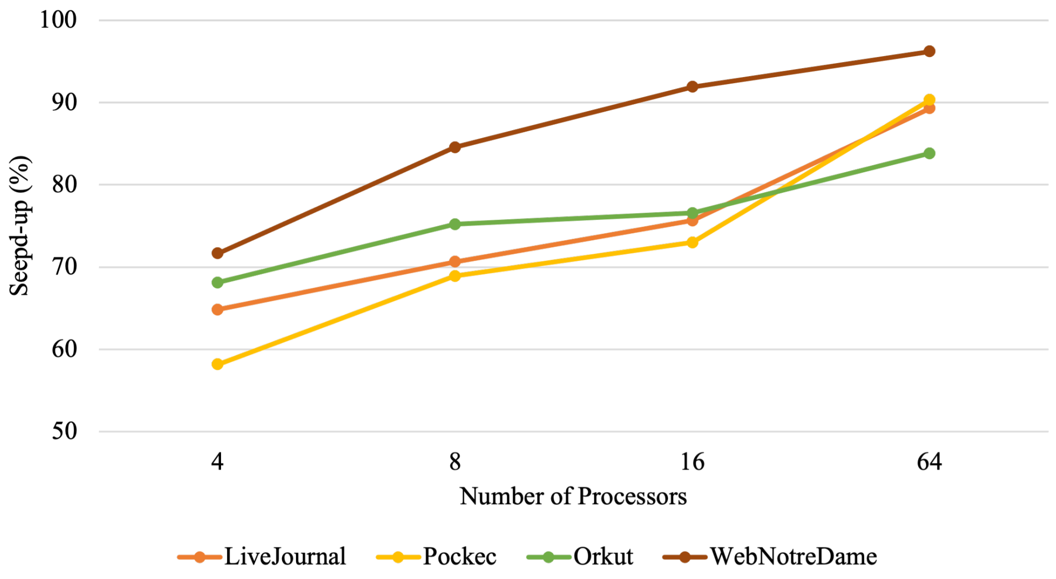


Figure 7.7: Shows the speed-up gained using multiple processors to compress the graphs to CSR.

7.7 Summary

In summary, we developed a parallel degree computation, which is by nature sequential. Which is one of the most integral part of constructing a CSR. Then we introduced a parallel prefix sum in a multi-threaded environment, with which we based the differental time-evolving CSR.

Chapter 8

Conclusions

We conclude by summarizing the concepts we covered in this dissertation.

8.1 Chapter 3

Valuable insights can be gained from the analysis of time-evolving graphs. However, due to the large size of such graphs, the memory requirements are significant and it is a challenge for computing using the main memory. Therefore, in this paper, we propose compression techniques for time-evolving graphs.

Our techniques show that a significant reduction in memory requirements can be achieved by exploiting the topological characteristics of graphs, specifically, the adjacency information for each node also known as row-by-row compression.

With the help of the characteristics of the graphs, we were also able to combine two identical or different techniques to compress a graph, as proposed in section 3.4. We also compare our compression results with state-of-the-art compression CBT-CBT and $ck^d - trees$, as shown in Table 3.5.

We implement our algorithms on real-world datasets and show significant improvements in the time required to query edges or any node's neighbors at a given time over the existing techniques, as shown in Tables 3.6 and 3.7, thereby showing a clear space/time tradeoff between the compression size and the querying time.

8.2 Chapter 4

In this chapter, we focus on extending the work proposed by Nelson et. al, (Nelson et al., 2018), (Nelson et al., 2021) to tensor representation. In this work, we proposed different ways to represent a tensor. In the first method, we spoke about two ways in which one can store an even-mode tensor, the second method is to store an odd-mode tensor, and the third way is the most common way one store a tensor is to unfold a multi-mode tensor into a 2-mode tensor a.k.a matrices. We also propose a block-wise tensor compression, with variable block-size making the structure amenable to parallelism, and having the block of data into the shared memory for faster computations. One of our approaches using $CBTs$ shows that the large sparse tensors occupy nearly 1/3 of the space occupied by the storage technique used in *tsparse*. To utilize the compressed structure, we performed a matrix-matrix multiplication on the metricized tensors to show the operations that are capable on a matrix of such a large scale. Our compression technique also allows one to eliminate the need to use ESC methodologies to update the partially computed matrix. We also show the ability to compress the metricized tensors into blocks of size greater than 8×8 as needed in (Zachariadis et al., 2020) (Zhang and Gruenwald, 2018). The multiplication technique can be further improved by constructing CBT on the partial results obtained in the algorithm 8, and combining these $CBTs$ at the end of the algorithm 7. This helps to multiply larger blocks, as the partial results are compressed in the computation.

8.3 Chapter 5

In this chapter, we have adapted our previous work of CSR and CBT, the compression realm, to introduce non-negative value-based matrix multiplication using the concept of the partial sum to reduce the number of row query operations on the compressed data structure.

We test our algorithm on extremely large matrices in the order of 100s of millions with various levels of sparsity. We show for matrices of order 100 million with 10 million nonzero elements, the space required to store the matrices using the CBT representation is about 6.4MB and requires 13.52s to complete the multiplication using the sequential algorithms provided in this chapter.

8.4 Chapter 6

In this chapter, we show that the given million-scale matrix can be factorized directly on the compressed structure. We also show that the intermediate result obtained in the matrix factorization process can be eliminated using sequential matrix operations. In this chapter, we also introduced element-wise matrix multiplication, division, subtraction, addition, and sequential multiple matrix multiplications on top of the existing work of matrix multiplication. We have also shown that traversing through the matrix in the pattern can avoid an explicit transpose operation during the matrix factorization. We also provide the heuristic relationship between inner rank and the sparsity of the factor matrices, and we have also shown in the results that the lower the rank, the smaller the factors W

and H. In the future, we would expand the computation to the Alternating Least Squares and Gradient Descent approach to factorize matrices. Our compression algorithms mentioned in this paper natively support binary matrices. Hence, we would also expand our work toward Binary Matrix Factorization.

8.5 Chapter 7

In conclusion, the analysis of social networks can provide valuable insights, but as the size of these networks grows, storing them for analysis becomes a challenge. Although various storing mechanisms are available, compressing a graph for storage takes time, and accessing the information directly from a compressed structure is not always straightforward.

In this chapter, we solve the problem by speeding up the compression process on CSR and also increasing the number of queries that can be performed at once. To aid in the process of constructing the CSR in parallel, we provide a parallel prefix sum approach to compute the degree array concurrently. We also propose algorithms to construct the time-evolving differential CSR in parallel using prefix sum.

Overall, the contributions of this chapter provide a valuable foundation for efficient parallel graph processing, which is essential for dealing with the increasingly large and complex graphs that arise in many real-world applications.

Bibliography

https://github.com/sudhigopal/csr_cbt_paper, 02/2021. URL https://github.com/sudhigopal/CSR_CBT_Paper.

http://socialnetworks.mpi-sws.org/data-www2009.html, 03/2020. URL http://socialnetworks.mpi-sws.org/data-www2009.html.

http://konect.uni-koblenz.de/, 03/2020. URL http://konect.uni-koblenz.de/.

http://webscope.sandbox.yahoo.com/catalog.php?datatype=g, 03/2020. URL http://webscope.sandbox.yahoo.com/catalog.php?datatype=g.

Réka Albert, Hawoong Jeong, and Albert-László Barabási. Diameter of the world-wide web. *nature*, 401(6749):130–131, 1999.

S. Álvarez-García, N. R. Brisaboa, G. d. Bernardo, and G. Navarro. Interleaved K2-Tree: Indexing and Navigating Ternary Relations. In *2014 Data Compression Conference*, pages 342–351, March 2014. doi: 10.1109/DCC.2014.56.

Michael A Bender, Erik D Demaine, and Martin Farach-Colton. Cache-oblivious b-trees. In *Proceedings 41st Annual Symposium on Foundations of Computer Science*, pages 399–409. IEEE, 2000.

G. D. Bernardo, N. R. Brisaboa, D. Caro, and M. A. RodrÃguez. Compact data structures for temporal graphs. In *2013 Data Compression Conference*, pages 477–477, March 2013. doi: 10.1109/DCC.2013.59.

Michael W Berry and Murray Browne. Email surveillance using non-negative matrix factorization. *Computational & Mathematical Organization Theory*, 11 (3):249–264, 2005.

Michael W Berry, Murray Browne, Amy N Langville, V Paul Pauca, and Robert J Plemmons. Algorithms and applications for approximate nonnegative matrix factorization. *Computational statistics & data analysis*, 52(1):155–173, 2007.

Guy E Blelloch. Prefix sums and their applications. 1990.

P. Boldi and S. Vigna. The Webgraph Framework I: Compression Techniques. In *Proceedings of the 13th International Conference on World Wide Web*, WWW '04, pages 595–602, New York, NY, USA, 2004. ACM. ISBN 1-58113-844-X. doi: 10.1145/988672.988752. URL http://doi.acm.org/10.1145/988672.988752.

Nieves R. Brisaboa, Diego Caro, Antonio Fariña, and M. Andrea Rodríguez. A compressed suffix-array strategy for temporal-graph indexing. In *SPIRE*, 2014a.

Nieves R Brisaboa, Susana Ladra, and Gonzalo Navarro. Compact representation of web graphs with extended functionality. *Information Systems*, 39:152–174, 2014b.

Binh-Minh Bui-Xuan, Afonso Ferreira, and Aubin Jarry. Computing shortest, fastest, and foremost journeys in dynamic networks. Technical Report RR-4589, INRIA, October 2002. URL https://hal.inria.fr/inria-00071996.

Diego Caro, M. Andrea Rodríguez, and Nieves R. Brisaboa. Data structures for temporal graphs based on compact sequence representations. *Inf. Syst.*, 51(C):1–26, July 2015. ISSN 0306-4379. doi: 10.1016/j.is.2015.02.002. URL http://dx.doi.org/10.1016/j.is.2015.02.002.

Diego Caro, M. Andrea Rodriguez, Nieves R. Brisaboa, and Antonio Farina. Compressed kd-tree for temporal graphs. *Knowl. Inf. Syst.*, 49(2):553–595, November 2016. ISSN 0219-1377. doi: 10.1007/s10115-015-0908-6. URL http://dx.doi.org/10.1007/s10115-015-0908-6.

Flavio Chierichetti, Ravi Kumar, Silvio Lattanzi, Michael Mitzenmacher, Alessandro Panconesi, and Prabhakar Raghavan. On Compressing Social Networks. In *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '09, pages 219–228, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-495-9. doi: 10.1145/1557019.1557049. URL http://doi.acm.org/10.1145/1557019.1557049.

Steven Dalton, Luke Olson, and Nathan Bell. Optimizing sparse matrix-matrix multiplication for the gpu. *ACM Transactions on Mathematical Software (TOMS)*, 41(4):1–20, 2015.

Eduardo F. D'Azevedo, Mark R. Fahey, and Richard T. Mills. Vectorized sparse matrix multiply for compressed row storage format. In Vaidy S. Sunderam, Geert Dick van Albada, Peter M. A. Sloot, and Jack J. Dongarra, editors, *Computational Science – ICCS 2005*, pages 99–106, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg. ISBN 978-3-540-32111-8.

Mehmet Deveci, Christian Robert Trott, and Sivasankaran Rajamanickam. Multi-threaded sparse matrix sparse matrix multiplication for many-core and gpu architectures. Technical report, Sandia National Lab.(SNL-NM), Albuquerque, NM (United States), 2018.

Laxman Dhulipala, Guy E Blelloch, and Julian Shun. Low-latency graph streaming using compressed purely-functional trees. In *Proceedings of the 40th ACM SIGPLAN conference on programming language design and implementation*, pages 918–934, 2019.

P. Elias. Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory*, 21(2):194–203, 1975. doi: 10.1109/TIT.1975.1055349.

Paul Erdős and Alfréd Rényi. On the strength of connectedness of a random graph. *Acta Mathematica Hungarica*, 12(1):261–267, 1961.

Paul Erdos, Alfréd Rényi, et al. On the evolution of random graphs. *Publ. Math. Inst. Hung. Acad. Sci*, 5(1):17–60, 1960.

Afonso Ferreira and Laurent Viennot. A Note on Models, Algorithms, and Data Structures for Dynamic Communication Networks. Research Report RR-4403, INRIA, 2002. URL https://hal.inria.fr/inria-00072185.

Robert W Floyd. Algorithm 97: shortest path. *Communications of the ACM*, 5 (6):345, 1962.

Nicolas Gillis. The why and how of nonnegative matrix factorization. *Connections*, 12(2):257–291, 2014.

Edward F Gonzalez and Yin Zhang. Accelerating the lee-seung algorithm for nonnegative matrix factorization. http://www.caam.rice.edu/tech_reports/2005/TR05-02.ps, 2005.

Sudhindra Gopal Krishna, Michael Nelson, Sridhar Radhakrishnan, Amlan Chatterjee, and Chandra Sekharan. On Compressing Time-Evolving Networks. In *ALLDATA 2021, The Seventh International Conference on Big Data, Small Data, Linked Data and Open Data*, pages 43–48, 2021. ISBN 978-1-61208-842-6. URL https://www.thinkmind.org/index.php?view=article&articleid=alldata_2021_1_70_80024.

Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. High-order Entropy-compressed Text Indexes. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '03, pages 841–850, Philadelphia, PA, USA, 2003. Society for Industrial and Applied Mathematics. ISBN 0-89871-538-5. URL http://dl.acm.org/citation.cfm?id=644108.644250.

Naiyang Guan, Dacheng Tao, Zhigang Luo, and John Shawe-Taylor. Mahnmf: Manhattan non-negative matrix factorization. *arXiv preprint arXiv:1207.3438*, 2012.

David Guillamet and Jordi Vitria. Non-negative matrix factorization for face recognition. In *Catalonian Conference on Artificial Intelligence*, pages 336–344. Springer, 2002.

Fred G Gustavson. Two fast algorithms for sparse matrices: Multiplication and permuted transposition. *ACM Transactions on Mathematical Software (TOMS)*, 4(3):250–269, 1978.

David A Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

Alon Itai, Alan G Konheim, and Michael Rodeh. A sparse table implementation of priority queues. In *Automata, Languages and Programming: Eighth Colloquium Acre (Akko), Israel July 13–17, 1981 8*, pages 417–431. Springer, 1981.

Ramakrishnan Kannan, Grey Ballard, and Haesun Park. Mpi-faun: an mpi-based framework for alternating-updating nonnegative matrix factorization. *IEEE Transactions on Knowledge and Data Engineering*, 30(3):544–558, 2017.

Oguz Kaya, Ramakrishnan Kannan, and Grey Ballard. Partitioning and communication strategies for sparse non-negative matrix factorization. In *Proceedings of the 47th International Conference on Parallel Processing*, pages 1–10, 2018.

Udayan Khurana and Amol Deshpande. Efficient snapshot retrieval over historical graph data. *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 997–1008, 2013.

Hyunsoo Kim and Haesun Park. Nonnegative matrix factorization based on alternating nonnegativity constrained least squares and active set method. *SIAM journal on matrix analysis and applications*, 30(2):713–730, 2008.

Sudhindra Gopal Krishna, Aditya Narasimhan, Sridhar Radhakrishnan, and Richard Veras. On large-scale matrix-matrix multiplication on compressed structures. In *2021 IEEE International Conference on Big Data (Big Data)*, pages 2976–2985, 2021. doi: 10.1109/BigData52589.2021.9671829.

Alan G. Labouseur, Jeremy Birnbaum, Paul W. Olsen, Jr., Sean R. Spillane, Jayadevan Vijayan, Jeong-Hyon Hwang, and Wook-Shin Han. The G* Graph Database: Efficiently Managing Large Distributed Dynamic Graphs. *Distrib. Parallel Databases*, 33(4):479–514, December 2015. ISSN 0926-8782. doi: 10.1007/s10619-014-7140-3. URL http://dx.doi.org/10.1007/s10619-014-7140-3.

Daniel Langr and Pavel Tvrdik. Evaluation criteria for sparse matrix storage formats. *IEEE Transactions on parallel and distributed systems*, 27(2):428–440, 2015.

Daniel D Lee and H Sebastian Seung. Algorithms for non-negative matrix factorization. In *Advances in neural information processing systems*, pages 556–562, 2001.

Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, 11 2021.

Chih-Jen Lin. On the convergence of multiplicative update algorithms for nonnegative matrix factorization. *IEEE Transactions on Neural Networks*, 18(6): 1589–1596, 2007.

Paul Lin, Matthew Bettencourt, Stefan Domino, Travis Fisher, Mark Hoemmen, Jonathan Hu, Eric Phipps, Andrey Prokopenko, Sivasankaran Rajamanickam, Christopher Siefert, et al. Towards extreme-scale simulations for low mach fluids with second-generation trilinos. *Parallel processing letters*, 24(04):1442005, 2014.

Michael Nelson, Sridhar Radhakrishnan, Amlan Chatterjee, and Chandra Sekharan. Queryable Compression on Streaming Social Networks. In *Big Data (Big Data), 2017 IEEE International Conference on*, IEEE BigData '17. IEEE Computer Society, 2017. ISBN 978-1-5386-2715-0. doi: 10.1109/BigData.2017.8258020. URL https://ieeexplore.ieee.org/document/8258020/.

Michael Nelson, Sridhar Radhakrishnan, and Chandra Sekharan. Queryable Compression on Time-Evolving Social Networks with Streaming. In *Big Data (Big Data), 2018 IEEE International Conference on*, IEEE BigData '18. IEEE Computer Society, 2018. doi: 10.1109/BigData.2018.8622386. URL https://ieeexplore.ieee.org/abstract/document/8622386.

Michael Nelson, Sridhar Radhakrishnan, and Chandra N Sekharan. Billion-scale matrix compression and multiplication with implications in data mining. In *2019 IEEE 20th International Conference on Information Reuse and Integration for Data Science (IRI)*, pages 395–402. IEEE, 2019.

Michael Nelson, Sridhar Radhakrishnan, Chandra Sekharan, Amlan Chatterjee, and Sudhindra Gopal Krishna. Queryable compression on time-evolving web and social networks with streaming. *ACM Trans. Web*, 16(2), dec 2021. ISSN 1559-1131. doi: 10.1145/3495012. URL https://doi.org/10.1145/3495012.

Masaaki Nishino, Norihito Yasuda, Shin-ichi Minato, and Masaaki Nagata. Accelerating graph adjacency matrix multiplications with adjacency forest. In

Proceedings of the 2014 SIAM International Conference on Data Mining, pages 1073–1081. SIAM, 2014.

NVIDIA. CUDA C Programming Guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html, 03 2022.

Pentti Paatero and Unto Tapper. Positive matrix factorization: A non-negative factor model with optimal utilization of error estimates of data values. *Environmetrics*, 5(2):111–126, 1994. doi: https://doi.org/10.1002/env.3170050203. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/env.3170050203.

Chenghui Ren, Eric Lo, Ben Kao, Xinjie Zhu, and Reynold Cheng. On querying historical evolving graph sequences. *PVLDB*, 4:726–737, 2011.

A.H. Robinson and C. Cherry. Results of a prototype television bandwidth compression scheme. *Proceedings of the IEEE*, 55(3):356–364, 1967. doi: 10.1109/PROC.1967.5493.

Yaroslav Shitov. The nonnegative rank of a matrix: Hard problems, easy solutions. *SIAM Review*, 59(4):794–800, 2017.

Julian Shun and Guy E Blelloch. Ligra: a lightweight graph processing framework for shared memory. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 135–146, 2013.

Julian Shun, Laxman Dhulipala, and Guy E Blelloch. Smaller and faster: Parallel processing of compressed graphs with ligra+. In *2015 Data Compression Conference*, pages 403–412. IEEE, 2015.

Shaden Smith, Niranjay Ravindran, Nicholas D. Sidiropoulos, and George Karypis. Splatt: Efficient and parallel sparse tensor-matrix multiplication. In *2015 IEEE International Parallel and Distributed Processing Symposium*, pages 61–70, 2015. doi: 10.1109/IPDPS.2015.27.

Richard A Snay. Reducing the profile of sparse symmetric matrices. *Bulletin Géodésique*, 50(4):341–352, 1976a.

Richard A Snay. Reducing the profile of sparse symmetric matrices. *Bulletin Géodésique*, 50(4):341–352, 1976b.

Stanford Network Analysis Project. Stanford Large Network Data Collection. https://snap.stanford.edu/data/index.html, 2011.

Volker Strassen. Gaussian elimination is not optimal. *Numerische mathematik*, 13(4):354–356, 1969.

William F Tinney and John W Walker. Direct solutions of sparse network equations by optimally ordered triangular factorization. *Proceedings of the IEEE*, 55(11):1801–1809, 1967.

Stephen A Vavasis. On the complexity of nonnegative matrix factorization. *SIAM Journal on Optimization*, 20(3):1364–1377, 2010.

Yu-Xiong Wang and Yu-Jin Zhang. Nonnegative matrix factorization: A comprehensive review. *IEEE Transactions on knowledge and data engineering*, 25 (6):1336–1353, 2012.

Brian Wheatman and Helen Xu. A parallel packed memory array to store dynamic graphs. In *2021 Proceedings of the Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 31–45. SIAM, 2021.

Martin Winter, Rhaleb Zayer, and Markus Steinberger. Autonomous, independent management of dynamic graphs on gpus. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7. IEEE, 2017.

Michael M Wolf, Mehmet Deveci, Jonathan W Berry, Simon D Hammond, and Sivasankaran Rajamanickam. Fast linear algebra-based triangle counting with kokkoskernels. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7. IEEE, 2017.

Wei Xu, Xin Liu, and Yihong Gong. Document clustering based on non-negative matrix factorization. In *Proceedings of the 26th annual international ACM SIGIR conference on Research and development in informaion retrieval*, pages 267–273, 2003.

Orestis Zachariadis, Nitin Satpute, Juan Gómez-Luna, and Joaquín Olivares. Accelerating sparse matrix–matrix multiplication with gpu tensor cores. *Computers & Electrical Engineering*, 88:106848, 2020.

Jianting Zhang and Le Gruenwald. Regularizing irregularity: Bitmap-based and portable sparse matrix multiplication for graph data on gpus. GRADES-NDA '18, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450356954. doi: 10.1145/3210259.3210263. URL https://doi.org/10.1145/3210259.3210263.

Sheng Zhang, Weihong Wang, James Ford, and Fillia Makedon. Learning from incomplete ratings using non-negative matrix factorization. In *Proceedings of the 2006 SIAM international conference on data mining*, pages 549–553. SIAM, 2006.

Jacob Ziv and Abraham Lempel. Compression of individual sequences via variable-rate coding. *IEEE transactions on Information Theory*, 24(5):530–536, 1978.