HoMLN-SD: Substructure Discovery In Homogeneous Multilayer Networks

by

ARSHDEEP SINGH

Presented to the Faculty of the Graduate School of
The University of Texas at Arlington in Partial Fulfillment
of the Requirements
for the Degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE AND ENGINEERING

THE UNIVERSITY OF TEXAS AT ARLINGTON
Dec 2023

ACKNOWLEDGEMENTS

I would like to extend my heartfelt gratitude to Dr. Sharma Chakravarthy for giving me the chance to be involved in his research and working on this project. I am truly thankful for the unwavering support and guidance that he has provided me throughout this endeavor.

I am extremely grateful to Dr. Abhishek Santra for providing me with guidance and support during all the ups and downs of my academic journey. His expertise and insightful feedback have been invaluable throughout the entire process. Thank you for sharing your time and knowledge with me.

I would like to express my gratitude to Dr. Ashraf Aboulnaga for dedicating his time to serve on my committee and for providing valuable comments, suggestions, and guidance.

Additionally, I want to express my sincere appreciation to my friends at ITLab - Kiran, Hafsa and Anamitra, for all their continuous help. Thank you all once again for your contributions, and I am honored to have had the privilege of working with such talented individuals.

I would also like to express my heartfelt thanks to my family and friends for their unwavering support and encouragement. Their love and belief in me have been a driving force behind my successes. They have always been there for me and their constant support has kept me motivated and inspired. I am forever grateful for their presence in my life.

Dec 1, 2023

ABSTRACT

HoMLN-SD: Substructure Discovery In Homogeneous Multilayer Networks

Arshdeep Singh, M.S.

The University of Texas at Arlington, 2023

Supervising Professor: Dr. Sharma Chakravarthy

Substructure discovery is a process in data analysis and data mining that involves identifying and extracting meaningful patterns, structures, or components within a larger dataset. These substructures can be of various types, such as frequent patterns, motifs, or any other relevant features within the data. The growth of the internet and the proliferation of mobile devices have led to the generation of enormous amounts of data. Companies like Facebook and Twitter can generate large datasets from user interactions on their websites, such as connections between users and user generated content. Moreover, advances in processing power and storage capacity have made it possible to collect and store these large amounts of data, known as big data.

The present research in big data focuses on developing new techniques and technologies for modeling, processing, and analyzing large and complex datasets. MLNs (Multilayer Networks) are more effective at modeling complex, diverse data, as compared to traditional data modeling techniques such as transactions or graphs (simple or attributed). MLNs consist of multiple layers, each layer representing a different feature of the dataset. For example, in a social network, each layer can

represent a different type of connection between users, such as friendship, work, or family. We can also model the same set of users across different social networks, with each network as its own layer making the data easier to understand and comprehend.

Substructure discovery in Multilayer Networks typically requires aggregation and integration of information from multiple layers, as substructures can extend beyond individual layers. Existing algorithms designed for single and attributed graphs and do not work natively on MLNs. As MLNs are a relatively new representation model, there is a need to develop algorithms that are specifically designed for the analysis of MLNs or to develop frameworks that can apply existing algorithms to MLNs while preserving the structure and semantics of the data. Currently available algorithms for graph mining are for single graphs or forests. They are memory based, disk based or use a database system approach. These algorithms have been extended to work on the MapReduce framework to improve scalability and have the capacity to process increasingly large datasets. But most of these need to aggregate the multilayer network to a single graph using projection or other types of aggregation (e.g., union).

This thesis presents two algorithms for computing substructures in homogeneous MLNs (where each layer has the same set of nodes but different connectivity) without layer aggregation, utilizing the decoupling approach. The first approach involves applying composition *after each iteration* of the substructure discovery algorithm. In this method, composition is carried out frequently, with the aim of attaining the same accuracy as ground truth. In the alternative approach, composition is only applied once at the end of the substructure discovery process. This approach is more computationally efficient, as it reduces the overhead associated with frequent composition. This approach is a trade-off between accuracy and efficiency. One of the goals for this dual approach is to analyze and understand this trade-off. In both approaches, the iterative substructure discovery process remains the same. We begin with individual layers and identify substructures within each layer. Subsequently, the generated results from each layer are used to compose substructures that span across multiple layers.

We use a decoupling approach (divide and conquer paradigm) for MLN substructure discovery to accommodate layers of arbitrary size. We employ the MapReduce framework to leverage distributed data processing for substructure discovery. This allows us to achieve better scalability and response time. The decoupling approach can be applied to handle multiple layers. Mining substructure in this way posed several challenges which we had to address: i) finding layer-wise substructures, ii) composing the layer-wise results to find substructures across layers, iii) maximizing performance by exploiting parallelism as much as possible, and iv) verifying the correctness of results. For verification, the proposed algorithms were applied to synthetic (Subgen) and real-world datasets like Amazon and DBLP.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

Figure Page

LIST OF TABLES

Table Page

CHAPTER 1

INTRODUCTION

The ability to analyze large, diverse datasets is becoming increasingly important in many areas of research from diverse application domains. Big data analytics can provide valuable insights and help businesses make data-driven decisions. Big data analysis is extremely valuable for organizations as it provides knowledge that would be difficult to discover using traditional methods. They may reveal interesting patterns, trends, and invaluable business insights which can then be used for improving business strategies, developing new products or services, and making better decisions. Moreover, big data analytics is not limited to a specific industry or sector. It is being used in healthcare to develop personalized treatments, in finance to detect fraud, in transportation to optimize routes, and in retail to gain insights into customer behavior and improve product design and delivery. With the massive amount of data generated by sources such as social media, e-commerce websites, scientific experiments, and financial transactions, there is a need for effective methods and frameworks to model this data and extract insights. This includes the ability to identify important relationships that constitute repeating patterns in the data.

Therefore, data mining has always been a crucial component of big data analysis, helping organizations understand large and complex data sets. The purpose of data mining is to extract interesting patterns that are frequent or possess interesting properties, such as good compression of the original graph in data that may not be apparent without mining. Because graph databases are widely used to represent data in which entities have one or more relationships, graph mining is useful for analyzing data that has many structural relationships. Graphs are used to model data by representing objects or entities as nodes, and the relationships between them as edges. For example, in a social network, nodes might represent users with username as node label and edges might represent connections between those users like friends, followers, etc. as edge labels. Similarly, in a chemical compound, nodes might represent atoms, and edges might represent the bonds between those atoms. Graph based database management systems like Neo4J [1] are becoming increasingly popular due to their ability to model interconnected and complex data sets as graphs and analyze them. They offer optimized performance and scalability, making them well-suited for applications such as social networking, recommendation engines, and network management.

Graph-based data mining has proven to be a powerful approach for analyzing data that can be naturally represented as a graph. The key to this approach is the ability to extract and analyze the numerous components (in the form of substructures which are connected subgraphs) that are present within the data modeled by a graph. We can then use these to extract interesting patterns present in the dataset. By leveraging graph mining techniques, we can discover these interesting patterns that may be difficult or impossible to identify using other methods.

1.1 Substructure Discovery in Graphs

Substructure discovery is an unsupervised approach to data mining to infer new knowledge and is typically applied on graph representations. Frequent patterns are a common concept in data mining that refers to item sets, subsequences, or substructures that occur frequently in a given dataset, meeting or exceeding a user-defined threshold. For example, in a transactional dataset, a frequent itemset could be a set of items, such as bread and butter, that are often bought together. Such an insight can be used to identify sets of items that are frequently purchased together and help a business stock and group their items appropriately. Similarly, in the context of graph theory, frequent structural patterns can be recurring patterns of subgraphs, subtrees, or sublattices that occur frequently in the data.

Suppose we have a graph database, and we want to identify frequently occurring subgraphs. Frequent occurrence indicates the importance of the substructure in the larger data set and hence can be of interest. For example, number of occurrences of a substructure in a protein indicates the contribution of the substructure to the overall characteristic. This may differ from protein to protein. We could define a frequent subgraph as a substructure that appears in a minimum number of graphs in the database, as specified by a user-specified threshold. By identifying frequent subgraphs, we can gain insights into the underlying structure of the data and identify recurring patterns or motifs.

Frequent substructures have numerous applications across various domains, and their identification can help uncover essential insights and knowledge that can be used to advance research and development in these fields. For example, in the field of bioinformatics, substructure discovery can be used to identify common patterns in DNA sequences, which may be indicative of important biological functions or relationships. As graphs are often used to model complex systems, such as social networks, transportation networks, and biological networks, identifying substructures within these graphs can help us understand the underlying patterns and relationships that govern these systems.

In the biological domain, frequent structural patterns can identify common substructures or motifs in biological networks, such as protein-protein interaction networks or gene regulatory networks. These may correspond to functional units or biological pathways, which can provide valuable insights into the underlying biology.

In social network analysis, frequent structural patterns can identify common substructures in social networks, such as groups of individuals that frequently interact or clusters of individuals with similar interests. This information can be useful in identifying groups, understanding the dynamics of social networks, and predicting behaviors or outcomes. For example, number of subgraphs (frequency) of a particular size indicates the number of people with that many relationships (e.g., friends, connections etc).

In chemistry, frequent subgraphs or structural patterns in chemical compounds can provide insights into the properties of the molecules, such as their reactivity or biological activity. For example, identifying frequent subgraphs in a database of compounds with known biological activities can help identify structural features associated with specific biological functions, which can be useful in drug discovery and design.

In computer science, these substructures can be used for a variety of applications such as network analysis, data mining, and machine learning. They can be used to simplify and analyze complex graphs. By identifying and isolating substructures, we can reduce the complexity of a large or dense graph, making it easier to analyze or visualize.

1.2 Why MultiLayer Networks (MLNs) instead of traditional graphs

Most complex systems are made up of a variety of interacting components that work together to create novel emergent behavior. A promising method for studying such systems is to analyze the networks that encapsulate these interactions among the system's components within its model. Simple networks like single graphs (also called simple graphs or a monoplex) have provided valuable insights into the functioning of different natural, social, biological, and technological systems in recent decades. However, real-world systems are often interconnected, and they may have multiple interdependencies that single-layer networks can represent but are difficult to understand (structure as well as semantics) or analyze. A more comprehensive framework that allows for different networks to develop or interact with one another is required to manage this added complexity.

For example, the edges within networks can possess diverse characteristics, as observed in modern social networks. They frequently comprise multiple forms of connectivity information, edges can either be classified based on the nature of the relationships between people or the interactions between them. Moreover, social networks often include diverse types of nodes, such as males and females, or social structures, such as individuals being a part of a club or an organization. Representing a social system as a network where people are connected by only one type of relationship often leads to an oversimplification of reality and information loss. Therefore, sociologists have long acknowledged the significance of studying multiple social networks using distinct types of connectivity among the same group of individuals [2]. Such systems are better studied through multilayer networks. Multilayer networks are a more general framework that allows for the analysis of complex systems with multiple interacting units and inter-dependencies that are not properly captured by single layer networks. In a multilayer network, each layer represents a different type of interaction or relationship between the system's constituents. For example, in a social network, one layer may represent friendships between individuals, while another layer may represent professional LinkedIn connections. By analyzing the interactions between layers, it is possible to gain insights into the emergent behavior of the system as a whole.

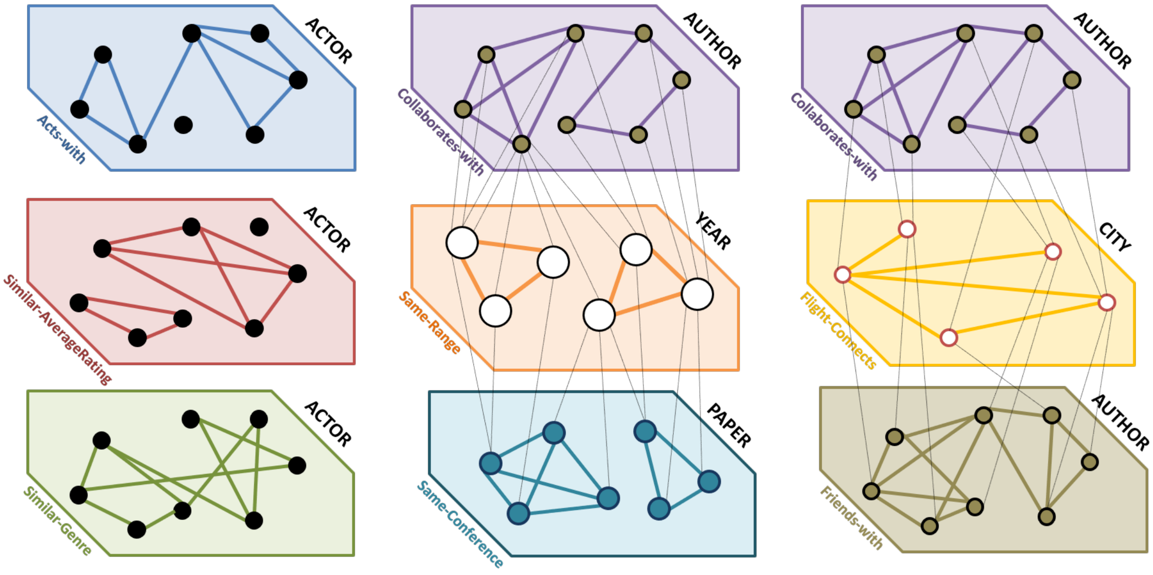
Multilayer networks are a powerful tool for modeling complex systems because they allow for the analysis of several types (or subsets of several types) of interactions and relationships simultaneously. This can reveal patterns and structures that would be difficult to detect using single layer network models or using the aggregated approach. Multilayer networks have been used to study a wide range of systems, from biological networks to transportation systems to social networks.

Multilayer networks provide a complete and more nuanced picture of the interactions and inter-dependencies that give rise to emergent behavior in complex systems, making them an essential tool for network science and other fields of research. Although there has been a lot of research for analyzing conventional graphs and a number of main-memory algorithms have been developed for various analysis, extending these to the multilayer networks can be challenging. Traditional network analysis techniques have limitations when applied to multilayer networks, and therefore, there is a need to develop new algorithms and methods to explore these networks and their characteristics. Traditionally, although multilayer networks are used for modeling, they are converted into a single graph using aggregation or projection for analysis.

The benefits of modeling data using multilayer networks are explored in detail in [3,4]. To illustrate the benefits of multilayer networks in real world applications, Zitnik et al. [5] constructed a multilayer network comprising molecular interactions, where each layer represented a different human tissue. Their results demonstrated that incorporating the tissue hierarchy into the model enhanced its predictive capability as compared to simple graphs.

Multilayer networks can be of several types, including homogeneous, heterogeneous, and hybrid, depending on the type of layers and connections between them. Figure 1.1 shows these types of multilayer networks.

1. **Homogeneous Multilayer Networks (HoMLNs):** In this type of MLN, each layer has the same or common set of nodes which are interconnected within the layer and termed as intra-layer edges. However, each layer has its own set of



(a) IMDb Set 1
(Homogeneous MLN)

(b) DBLP Set 2
(Heterogeneous MLN)

(c) Author-City Set 3
(Hybrid MLN)

Figure 1.1: Different Types of Graph Models

inter-layer edges based on the relationship being modeled. Each node or edge can have a labels. Labels of nodes and edges are not unique. For example, in a social network, we could have different layers representing different types of interactions (e.g., online messaging, face-to-face interactions), but the set of individuals remains the same in each layer.

2. **Heterogeneous Multilayer Networks(HeMLNs):** Here, the layers have different sets of node types, and the edges in each layer differ in terms of their meaning or interpretation. The presence of inter-layer edges that connect different types of nodes or entities across layers distinguish HeMLNs from HoMLNs. This makes them suitable for modeling and analyzing complex, heterogeneous data. For instance, we could have a multilayer network representing a transportation system, where each layer corresponds to a different mode of transport (e.g., roads, trains, flights), and the nodes in each layer are different (e.g., stations, airports, highways).

3. **Hybrid Multilayer Networks(HyMLNs):** This type of network combines elements of both homogeneous and heterogeneous multilayer networks, where some layers have the same set of nodes and edges, while others have different sets of nodes and edges. This type of network is often used to represent complex systems where there are both uniform and diverse interactions between the nodes, such as a social network where some individuals have multiple roles (e.g., student and teacher).

Over the past few years, there has been a surge of interest in mining and analysis of multilayer networks. Several algorithms have been developed for substructure discovery on single graphs and even attributed graphs, but to the best of our knowledge not for MLNs. Many existing algorithms for mining complex networks are main-memory algorithms and hence cannot deal with large graphs (depending on the available memory). However, massive-scale datasets from many applications areas (social networks being one of them) cannot be analyzed using these algorithms. Online social networks such as Facebook [6] and LinkedIn [7] have 3 billion and 950 million monthly active users respectively. Internet Movie Database (IMDb) [8] information on 513,598,803 data items with 14,631,612 titles and 12,418,036 individuals (actors, directors, writers, etc.) involved in the entertainment industry. These dataset sizes are typical of the databases that we see today.

Computing substructures on such large datasets is no longer viable on individual machines. Consequently, distributed approaches for storing and processing such data have become necessary. Moreover, the present approach of aggregating the MLN into a single graph becomes infeasible. This has led to a growing demand for the development of algorithms that can run directly on MLNs without aggregating them. Such algorithms for graph mining in multilayer networks can handle large-scale data and provide effective solutions to the challenges posed by multilayer networks.

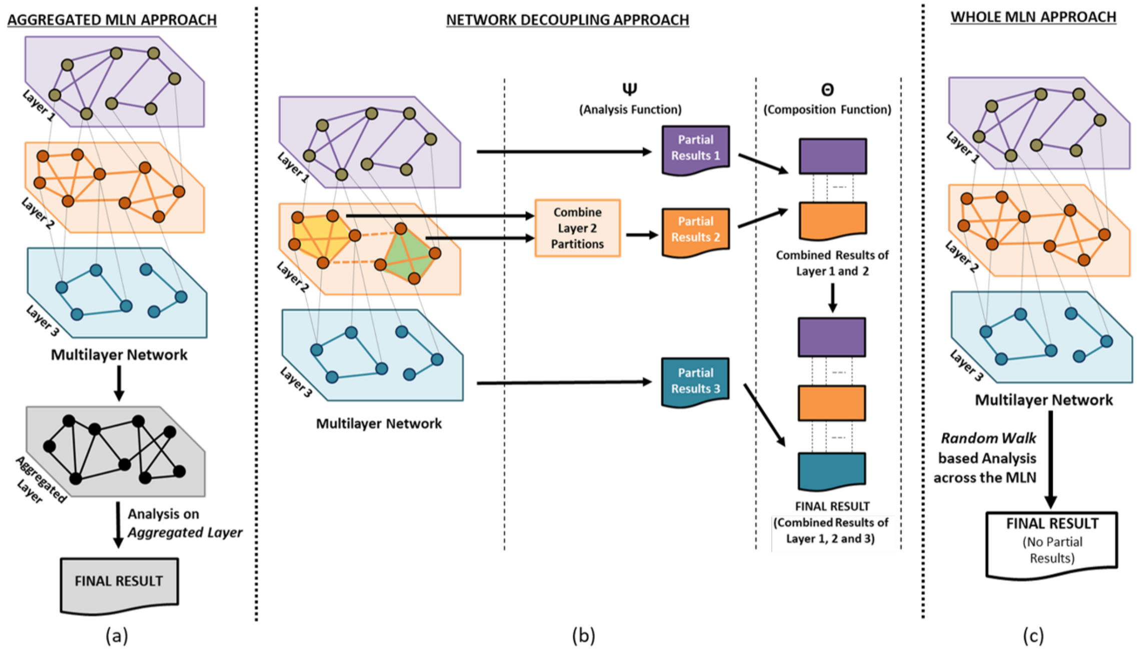


Figure 1.2: Approaches to Analyze MLNs

Among some of the work already done on MLNs, [9] proposed a novel general-purpose approach that can be used for any analysis by leveraging the layers as decompositions of MLN and computing analysis using a composition of results from individual layers.

1.3 Current Approaches to Analyze MLNs

Current approaches to analyze Multilayer Networks (MLNs) typically involve mapping the networks to an equivalent single graph through various methods. In case of homogeneous MLNs this involves aggregating the edges of the multilayer network into a single-layer network. However, this mapping process can lead to the loss of valuable information present in the multilayer graphs.

A network decoupling is presented as a method for analyzing MLNs without transforming them into another form. This is a novel method proposed in [10–12] for finding substructures in multilayer networks that approaches the problem by decoupling the network into individual layers of the MLN. The decoupling approach retains the structure and semantics of the layers in the result while leveraging existing algorithms. It is akin to a *divide and conquer* strategy for MLNs, as illustrated in Figure 1.2(b). It is applied as follows:

1. Utilize the analysis function to analyze each layer individually, considering aspects such as frequent subgraphs, community structure, centrality metrics, etc.
2. Apply a composition function to compose the partial results from each layer for any two chosen layers, generating intermediate results.
3. Repeat the composition process until the desired expression is computed.

This approach contrasts with current methods, as depicted in Figure 1.2(a), where aggregation-based approaches lead to the loss of both structure and semantics. Additionally, Figure 1.2(c) illustrates MLN approaches where only inter-layer edges are considered instead of all edges.

The decoupling-based approach has demonstrated effectiveness, particularly for centrality [12–14] and community [11, 15] analysis. However, its application to substructure discovery has been minimal [16]. A comprehensive investigation into the efficacy of the decoupling-based approach for substructure discovery has not been undertaken.

1.4 Substructure Discovery in a Multilayer Networks

There are two major approaches to find substructures in multilayer networks:

1. **Layer Aggregation into a single graph**

   This method aggregates the information from different layers of the network into a single network. We can then apply traditional graph mining techniques to identify substructures in the single or aggregated network. But there are drawbacks to this approach:

   - **Loss of information**: Aggregating the layers to a single layer results in the loss of information that is specific to each layer. Processing each layer independently and in parallel is not possible due to the aggregation process, which limits the parallelization opportunities.
   - **Complexity**: Aggregating the layers and processing can become computationally expensive as the size of the single graph increases due to inclusion of edges from each layer. This can limit the scalability of the approach. In the case of very large multilayer networks, the combined graph is too large to fit into memory and we need to use other approaches like partitioning.
   - **Lack of flexibility**: Aggregation is not appropriate when we need to work on a subset of the layers as each subset would need to be aggregated and separate single graphs generated for analysis.

2. **Decoupling based approach**

   In this approach, the substructures are identified in each layer independently. After that, the substructures are composed using a new composition function across two layers to obtain the final substructures in the Multilayer Network. This binary composition can be repeated for more than 2 layers.

   Decoupling approach offers a many of advantages over aggregating multiple layers into a single graph:

   (a) Preservation of MLN Structure: The underlying structure and modeling of the Multilayer Network (MLN) are retained. There is no loss of information [17] as the MLN structure is preserved along with the semantics (labels in each layer). In contrast, combining all layers into a single graph may hide the origin of generated substructures, making it challenging to discern which layers they belong to.

(b) Use of Existing Algorithms: It allows for the use of existing single-layer algorithms for identifying substructures. Any existing algorithm can be used for substructure discovery in each layer and parallel processing of each layer. This approach leverages the natural decomposition of a MLN into layers, which are likely to be smaller. The composition uses the output of these algorithms.

(c) Flexibility for Subset Analysis: Decoupling provides the flexibility to analyze specific subsets of layers within the MLN. This allows tailored analysis on selected layers without requiring the entire combined MLN.

(d) Better Parallel Processing: By handling each layer individually and in parallel,it enables more efficient use of resources. Each layer can be processed independently, taking advantage of parallel processing capabilities for faster results.

1.5 Problem Statement

This thesis addresses the problem of finding "interesting" substructures in a given Homogeneous Multilayer Network (MLN) using the decoupling approach while retaining the MLN structure and semantics. The crucial part is to compute the inter-layer substructures only after the identification of all the intra-layer substructures, by only using the results from individual layers and no other information. We also achieve this without aggregating the MLN layers into a single graph.

We use a decoupling approach to find substructures present in the layers [18] and then compute the inter-layer substructures (not generated when processing each layer individually) that exist across layers. When finding substructures in MLNs, the best substructures may exist across layers, which makes it challenging to compute the best substructures in the MLN correctly. As substructure discovery algorithms are iterative by nature, composition can be done in multiple ways: i) after each iteration, ii) after all iterations, or iii) somewhere in between.

The algorithms available in the literature [18–20] are not applicable to the multilayer setting as they are designed to identify substructures in a single graph. However, these substructures can be detected by aggregating the MLN into a single graph, which is the approach we have employed to generate our ground truth. Nonetheless, as the size of the MLN expands, analysis on the MLN using the aggregated approach becomes computationally costly and inefficient.

Subdue [19] is one of the early graph mining algorithms that detects the best substructure using the minimum description length principle. Subdue constructs the whole graph and stores it in the form of an adjacency matrix in main memory and then mines by iteratively expanding vertex of a substructure of size k to a substructure of size k+1 in iteration k.

HDB-Subdue [20] implements subdue using a relational database approach using SQL. The goal was to remove main-memory dependency as RDBMS has no size limitation on a relation. A graph is represented using a relation to represent an edge with label information as a tuple. It can handle multiple edges, cycles, and hierarchical reduction to deal with a general graph. HDB-Subdue uses unconstrained substructure expansion with duplicate elimination, to explore all possible expansions including multiple edges. SQL-based analytic functions were used to implement the beam. HDB-Subdue was able to scale to a million nodes and 1.6 million edges. But due to the self-joins on relations, the response time increased with the size of the graph. This approach could not be used for graphs of arbitrary size. Partition and distributed processing were essential to handle them.

MapReduce (or M/R) Subdue [18] casts the subdue into a distributed framework using the MapReduce framework. The basics of graph mining such as systematic expansion and computing graph similarity have been implemented in the MapReduce paradigm. This approach enables horizontal scalability of substructure discovery using partitioning strategies. This allowed the MapReduce based substructure discovery to scale to even larger graphs over traditional methods. We will discuss these approaches in more detail in Chapter 2.

The algorithm proposed in [16] followed the decoupling approach to find substructures in an MLN. The approach was to find increasing size substructures present in the layers of the MLN and compose them to compute inter-layer substructures after each iteration. This process was repeated till the desired size of substructure was reached or when there were no more substructures to generate.

As substructure discovery is an iterative process the composition function can be employed in each iteration or after the conclusion of the substructure discovery process for each layer, presenting a trade-off between response time and accuracy. In this thesis, we propose two approaches to substructure discovery in Homogeneous Multilayer Networks (HoMLNs): i) employing the composition function after each iteration, generating inter-layer substructures multiple times, and ii) applying the composition function only once at the end, generating inter-layer substructures only once. We are analyzing the trade-off between response time and accuracy to understand the nuances for these two approaches. We also employ a range-based partitioning scheme to handle arbitrarily large layers in the MLN.

1.6 MapReduce: A Distributed Framework

MapReduce is a widely used framework for processing large-scale data in a distributed environment. The ability of MapReduce to process data in parallel across distributed computing resources has made it a popular choice for large-scale data processing tasks, including graph computations. Many companies, such as Google, Facebook, and Twitter, have used the MapReduce framework for graph processing tasks, such as page ranking, community detection, and recommendation systems. Apache Hadoop is an open-source framework that implements the MapReduce programming model and has been widely used for large-scale distributed processing of data that can run on a large cluster of commodity machines and is highly scalable.

The MapReduce model involves three stages:

- **Map stage**: In this stage, the input data is split into smaller parts, and each split is processed independently using the same code (data parallelism) by a different worker node in the cluster. The worker nodes apply the user defined map function which takes an input data in the form of key-value pairs one at a time and converts it into a set of key- value pairs which are the intermediate outputs. Before the data is sent to the reducer nodes, an optional combiner can be applied to each group of key-value pairs at the mapper nodes. The combiner aggregates and combines values for the same key locally on the mapper node. This step reduces the amount of data that needs to be transferred to the reducers.
- **Shuffle stage**: In this stage, the intermediate outputs produced in the map stage are partitioned on the key emitted by the mapper (number of partitions determined by the number of reducers given or default which is 1) and shuffled based on partitions. All key value pairs within the same partition are sent to the same reduce worker node.
- **Reduce stage**: Partitions from each mapper are merged based on key value to obtain a single value list for each key in the partition. In this stage, the worker nodes process each key-value-list in the partition. Each key-value list is processed by the user code to generate the desired output. Each reducer writes it own output to HDFS.

The framework automatically handles the distribution of data by generating splits as indicated or using default block size, task scheduling, and fault tolerance across a cluster of commodity machines. The runtime system handles the parallelization and execution of the program, machine failures that may occur, and the required inter-machine communication.

The MapReduce model is designed to be user-friendly, even for programmers without prior experience with parallel and distributed systems. The model abstracts away the complexities of parallelization, fault-tolerance, locality optimization, and load balancing. As a result, users can focus on developing algorithms and logic specific to their applications without having to worry about the low-level details of distributed computing. This is why we have chosen the MapReduce paradigm to address our problem of substructure discovery in multilayer networks. MapReduce framework provides the architecture to process each of the layers independently while achieving maximum parallelization. It is also possible to split each layer and process the layer in a distributed manner if the layer size increases.

We have adapted the substructure discovery approach proposed for large single graphs in MapReduce [18] to work in the case of multilayer networks. The Mapper allows us to process the layers in parallel to find substructures in each layer. The shuffle and reduce stages are used to bring substructures that exist across layers together for composition. After composition, we rank the discovered substructure using graph isomorphism and a specified metric (MDL or frequency). Using the algorithms proposed in [18] enables the partitioning of layers using a range-based partitioning scheme. This approach allows us to handle input layers of arbitrary that are part of MLN.

Using this framework, our goal to preserve the MLN structure and process the layers independently is also accomplished. In addition, the MapReduce framework can automatically adjust to the available resources. The framework can be scaled up easily by requisitioning as many nodes as needed which can help to improve the response time for larger datasets. Our approach can use an arbitrary number of processors to exploit parallelism.

1.7 Thesis Contributions

The contributions of this thesis are:

- Proposed two composition algorithms for Substructure Discovery in a Homogeneous Multilayer Network.
- Incorporating the algorithms within the MapReduce framework
- Ensured correctness of both the approaches
- Evaluated the impact of various parameters (layer distribution, beam size, substructure size) on the accuracy.
- Extensive experimental analysis using large scale real-world and synthetic datasets to test accuracy and performance against the ground truth.

1.8 Thesis Organization

Rest of the thesis is organized as follows -

- Chapter 2 discusses the background and related work in graph mining and substructure discovery using MapReduce and other approaches.
- Chapter 3 elaborates on the preliminaries for graph mining such as input graph representation, sub graph expansion, duplicate elimination, graph isomorphism, partition management in the MapReduce framework.

- Chapter 4 presents our decoupling-based design of the composition algorithms for homogeneous multilayer networks – one for iterative (HoICA algorithm) and one for at-the-end single composition (HoSCA algorithm)
- Chapter 5 discusses the implementation details for all the components used for substructure discovery for Multilayer Networks in the MapReduce environment.
- Chapter 6 provides extensive experimental analysis of several data sets used to validate our approach.
- Chapter 7 concludes the thesis and discusses the avenues for future work.

CHAPTER 2

RELATED WORK

In this chapter, we shall present a comprehensive review of the significant work that has been done in the field of substructure discovery. We will examine the existing studies and methodologies undertaken for finding substructures within complex datasets. We will investigate the diverse approaches employed, ranging from memory-based approaches to disk-based approaches to database-oriented approaches. By critically analyzing the strengths and limitations of each approach, we aim to achieve a complete understanding of the current literature. This will allow us to compare our proposed algorithms and their advantages as compared to the work in the literature.

2.1 Existing Graph Mining techniques

The main challenge of counting all instances of a specific substructure that are exact or similar, in a graph is to develop an algorithm for detecting identical or similar subgraphs. To ensure the generation of all substructures, a systematic approach is required. This involves generating substructures of increasing sizes in each iteration. To contain the exponential size of that space, a heuristic is needed. This is done by using frequency or Minimum Description Length (MDL) [21] and choosing top-k values (or beam size). Higher frequency or MDL can indicate an interesting property of the graph. As discussed earlier, graph mining techniques in the context of substructure discovery can be main memory based, disk based or database-oriented. In this thesis, our focus is on mining recurring patterns and "interesting substructures" in a Homogeneous Multilayer Network. We will discuss research done in both single layer graphs (monoplexes) and multilayered graphs as this would enable us to observe the inherent evolution of substructure mining as the data modelling became more complex.

2.1.1 Main Memory Approaches

Most of the earlier approaches developed for graph mining utilized main memory algorithms. These algorithms employed a complete representation of the graph, typically an adjacency list or matrix, which needed to be loaded into memory for processing. In this approach, the algorithms could effectively access and process the entire graph, enabling the extraction of meaningful patterns and insights from the data. This approach was feasible as the size of the graphs was not too large and was able to fit into memory.

2.1.1.1 Subdue

Among the very first main memory algorithms was Subdue [19]. It iteratively generates substructures of increasing sizes and evaluates them using the MDL principle. The subdue algorithm uses a constrained beam search approach. The algorithm is initiated with a substructure of size one, which represents a single edge. In each iteration, the algorithm selects the best substructures and expands their instances by one edge. This expansion is unconstrained and all possible expansions are undertaken. The best substructures from the newly generated instances are selected for the next iteration. The algorithm keeps track of the best substructures, which would be returned when either all potential substructures have been evaluated or the overall computation surpasses a predefined threshold.

An intriguing aspect of the Subdue algorithm involves its utilization of background knowledge, which is used to direct the search towards more suitable substructures. This background knowledge is formulated as rules for assessing substructures, and it can have domain dependent or independent rules. When evaluating a substructure, these rules influence the value given to the substructure by the algorithm. Each rule is attributed with a positive or negative weight, which steers the search towards the desired type of substructure. Consequently, the assessment of each substructure is influenced by the insights provided by the user's background knowledge and the MDL principle.

2.1.1.2 Apriori-based approach

The other main memory approaches to graph mining are AGM [22] and FSG [23], with both following the Apriori-based methodology. The Apriori-based approach to searching for repetitive substructures follows a bottom-up approach, beginning with small-sized graphs and progressively expanding the search. During each iteration, existing substructures are expanded by merging two similar substructures that have slight variations. It uses frequent k-subgraphs to generate frequent (k+1) subgraphs. For instance, a substructure involving 3 edges could be identified by merging two substructures containing 2 edges each, which differ by only 1 edge.

The Apriori property, initially introduced in [22], forms the foundation of the Apriori Graph Mining (AGM) technique. AGM utilizes the apriori property to streamline the search scope, enhancing the efficiency and scalability of the repetitive substructure exploration process.

The FSG algorithm (Frequent Subgraphs) (FSG) [23] is another apriori-based approach which aims at discovering interesting substructures that appear frequently over an entire set of graphs. This differs from Subdue, where interesting substructures are discovered within a single graph (or a forest). It is designed along the lines of Apriori association rule mining algorithm but utilizes the concept of canonical labeling. By leveraging the fact that identical graphs exhibit the same canonical labeling, we can exploit this property to detect occurrences of frequent substructures. FSG employes a flattened representation of the graph's adjacency matrix for determining the canonical labels. This reduces the V! complexity for canonical labeling to V1!+V2!.. where V1 and V2 are smaller than V. Hence it avoids V! complexity.

However, main memory algorithms possess certain inherent limitations, such as their inability to manage large and extensive patterns, the generation of vast candidate sets, and the need for multiple database scans. These limitations, combined with the growing size of today's datasets, render these approaches infeasible within the current landscape of big data.

2.1.2 Disk-Based Approaches

The advent of increasingly comprehensive methodologies for data acquisition has led to the incorporation of even more structural information. This gives rise to the issue of graph sizes becoming so large that the data can no longer be stored in main memory. Disk-based graph mining methods [24–26] emerged to resolve this issue. A portion of the graph data is maintained in memory, while the remaining data is stored on the disk. Given the challenges and excessive costs associated with random access to disk-based graphs, indexing the graph appeared to be the most suitable approach. Moreover, frequent substructures are particularly well-suited for indexing due to their relative stability in database updates. This enables cost-effective incremental maintenance of the index. This is particularly beneficial in scenarios involving large graphs, where indexing can be achieved through the frequent substructures inherent within the graph.

The gIndex methodology [27] capitalizes on frequent substructures as indexing units. A substructures is deemed frequent if its frequency surpasses a defined minimum support threshold. This enables frequent substructures to be directly retrieved due to indexing. Additionally, leveraging the attribute of incremental updating, gIndex can be created through a sole scan of a given database.

However, many graph indexing techniques entail a computationally intensive index construction phase. It is essential to minimize the count of indexing features to maintain a compact index structure that can fit within the main memory for fast and efficient access. In cases where the graph size is large, the index size also enlarges proportionally. This leads to an expensive and inefficient index construction process and lead to an index structure that may be too large to fit into memory. Considering these challenges, devising more efficient techniques becomes of paramount importance.

2.1.3 Database-oriented Approaches

Disk-based algorithms tackle the challenge of handling portions of a graph within available memory for processing. However, these algorithms require explicit marshalling of data between the disk and the main memory. This requires careful integration into the algorithm's design and implementation. The effectiveness of disk-based methods can be greatly influenced by the rate of data transfer between disk and memory, buffer size, buffer management and the replacement policies, and hit rates. An alternative approach involves leveraging efficient buffer management and query optimization techniques that are already available in Database Management Systems (DBMSs). This involves mapping graph mining algorithms to SQL queries and utilizing a DBMS to store the data, taking advantage of the mature optimizations available in the DBMS. [20,28] have indeed employed this approach and we shall take a deeper look into their methodologies.

2.1.3.1 HDB-Subdue

HDB-Subdue [20] is a modification on top of both DB-Subdue and EDB-Subdue [29]. It can accommodate a more comprehensive array of graph structures, including cycles and multiple edges between vertices. This is achieved by extending the graph representation through the introduction of connectivity attributes. In addressing certain limitations of EDB-Subdue's constrained substructure expansion, HDB-Subdue allows for unconstrained expansion of substructures. However, this unconstrained expansion can give rise to generation of pseudo duplicate instances of the same substructure. This can happen when the same instance is expanded or enumerated but in a different order. To counter this, HDB-Subdue employs an approach of identifying these pseudo-duplicates by sorting substructure instances based on vertex numbers and connectivity maps, subsequently eliminating them. Similarly, for counting the frequency of substructure instances, HDB-Subdue arranges instances by vertex labels and connectivity attributes. Notably, HDB-Subdue also introduces hierarchical reduction of graphs. After the best substructure is determined for a given iteration, HDB-Subdue compresses the graph by replacing all instances of the best substructures with a single vertex.

2.1.3.2 DB-FSG

DB-FSG [28] shifts the paradigm of frequent subgraph (FSG) mining from a main memory-based approach to a database-oriented strategy while still operating across a set of graphs. It expands upon the substructure representation proposed by HDB-Subdue. A more efficient algorithm for the elimination of pseudo duplicates is presented, surpassing the efficiency of the approach used in HDB-Subdue. It also introduces a streamlined method for deducing structural relationships from relational data, aimed at facilitating graph mining, whether for identifying the best subgraph or for extracting frequent subgraphs. This approach involves the inference of the entity-relationship model from the database, utilizing instances of tables along with primary and foreign key constraints to generate instances of graphs based on populated instances of these relations. A key emphasis of this approach is its adaptability towards various relational database platforms, coupled with efforts to minimize memory and space requirements.

Although these database-oriented approaches were able to achieve scalability with graphs containing well over a million nodes and edges, they also had some notable challenges. The use of joins for substructure expansion is expensive and independent expansion (unlike main-memory approaches) leads to the generation of duplicate substructures. Removing these duplicate substructures necessitated the sorting of columns. But in row-based Database Management Systems (DBMSs), this leads to incurring significant costs in performance. As these DBMSs are optimized for sorting rows, several joins had to be employed for column sorting, contributing to the overall cost. Secondly, the performance of operations in DBMSs is constrained by the number of columns present in a relation, imposing an upper limit on the size of the largest substructure that could be represented using database-oriented approaches.

2.2 Distributed Approach to Graph Mining

As previously discussed, graph sizes can reach significant magnitudes, rendering traditional main memory-based systems ineffective. In response, alternative approaches store the graph on disk and load it into memory when required. Disk-based strategies are suitable for managing graphs larger than what can be accommodated in memory [30], but they introduce challenges related to customizing buffer management and introducing I/O latency. With the proliferation of big data, conventional systems become inadequate, prompting the need for a shift to distributed approaches for data management and processing. Several distributed graph databases are available for large-scale processing, including Neo4J [1], ArangoDB [31], Dgraph [32]. However, these databases primarily focus on enhancing the storage aspect of graph data and do not prioritize optimizing graph mining within the database.

Certain graph mining algorithms [33–35] have demonstrated effectiveness when deployed within a cloud architecture. Moreover, research has been done on finding interesting patterns within large graphs using the MapReduce framework [33]. However, the pattern finding algorithm proposed in [33] needs a predefined pattern to search for all instances of that specific pattern within the graph. But if we need to identify a pattern that exhibits specific characteristics, like having the most compressibility, we cannot provide the pattern beforehand. There is also ongoing research into partitioning extensive graphs into manageable segments to enhance distributed processing across computational resources [36].

2.3 Substructure discovery using MapReduce

M/R Subdue proposed in [18] implements the subdue algorithm using the MapReduce Paradigm. It casts main memory approach (Subdue), along with its nuances into a MapReduce distributed framework. This adaptation allows it to leverage the distributed processing capabilities provided by the MapReduce framework. Conventional main memory-based techniques have been extended to operate on graph partitions, enabling parallel processing. By leveraging the MapReduce distributed framework, the processing capacity is expanded across a cluster of commodity machines. This approach enhances the scalability of graph mining and enables more efficient handling of large-scale graph data.

The first step of substructure discovery is the partitioning of the input graph into smaller segments, followed by the aggregation of results across these partitions. It is an iterative algorithm that generates substructures of increasing sizes, starting from a single edge (substructure of size 1). Duplicates are removed as necessary and a metric (like MDL) is applied to rank the substructures. The expansion of substructures by one edge is performed in each iteration using adjacency lists for each partition. The algorithm continues until a specified substructure size is attained or no further substructures can be generated. After each iteration, a subset of the best substructures is selected for further consideration, updating substructure partitions accordingly. All substructures existing within the constrained beam search are passed on to the subsequent iteration for further expansion. This enables the efficient and distributed exploration of meaningful substructures within large graphs.

During the MapReduce process, each input record is handled individually by the Mapper. The input graph is represented as a sequence of graph edges, with each edge represented by an edge label, source vertex ID and label, and destination vertex ID and label. In the initial stages, substructure partitions are disjoint, ensuring that the same edge is not duplicated across multiple partitions. The adjacency list is partitioned according to the substructure partition as well. Following the initial iteration, new edges are incorporated into the existing substructure partitions. This necessitates the updating of the adjacency list for each partition that has gained a new edge. Therefore, it is essential to carefully consider the approach used for generating partitions and subsequently updating them.

2.4 Graph Partitioning using MapReduce

METIS [37] is one of the earliest and most effective graph partitioning schemes available. METIS can produce partitions of high quality while minimizing the edge crossover (cut-set) between partitions. However, it does consume a notable amount of time during the partition generation process. Another limitation of METIS is its incompatibility with temporal graphs which possess the ability to add and remove of vertices and edges over time. Therefore, M/R Subdue introduces two algorithms, dynamic AL-SD and static AL-SD, based on the graph partitioning technique used - Arbitrary Partitioning and Range Based Partitioning, respectively.

2.4.1 Arbitrary Partitioning

Here, the initial substructure partitions are generated through random grouping of graph edges. These initial partitions are disjoint, but adjacency list partitions are not. Each Mapper expands a given substructure using the corresponding adjacency list partition, in parallel. In the reducer, counting is performed by grouping together isomorphic substructures. Newly generated substructures are written to the same partition and sent forwarded for subsequent iterations. Considering the possibility of new substructure partitions containing edges that span multiple partitions, the adjacency list is updated to contain all newly introduced vertices along with their respective adjacency lists. The effectiveness of this type of partitioning is highly reliant on having a small edge cut set.

2.4.2 Range Partitioning

This approach partitions the global adjacency list by evenly distributing vertex IDs into distinct ranges, thereby establishing adjacency partitions. In contrast to the prior method, these adjacency partitions are disjoint and share no vertices between adjacency list partitions. The process employs two consecutive MapReduce jobs: the first job handles substructure partitioning and expansion, while the second one is responsible for substructure calculations. The adjacency list partitions remain the same across iterations, avoiding the expensive updates needed for arbitrary partitioning. This requires the routing of substructures to the correct partition which has the corresponding adjacency list for expansion. The range information allows for quick identification of the partitions to which a substructure belongs (a substructure can belong to multiple partitions). However, the same substructure may have to be expanded in multiple partitions due to disjoint nature of adjacency lists in the range partitioning approach.

The experimental analysis of both partitioning methods showed that despite Range partitioning incurring slightly higher shuffle costs, it outperforms Arbitrary partitioning, which involves high I/O costs due to the updating of the adjacency lists after each iteration. While traditional partitioning strategies like METIS aim to reduce edge crossover (cut-set) between partitions, they require multiple passes over the graph to create the initial partitions. This renders them unsuitable for larger graphs. Additionally, these partitioning approaches lack support for adjacency list partitions. Furthermore, their emphasis on creating equally sized connected partitions (or partitions that are a power of 2) can lead to varying workloads across heterogeneous machines, thereby impeding speedup. It is worth highlighting that the initial partitioning cost for the two aforementioned schemes is lower compared to traditional alternatives, as they only perform a single pass over the graph. Finally, Range partitioning is particularly well-suited for heterogeneous clusters due to its ability to align the range divisions with the capacities of individual machines.

2.5 Substructure discovery in Multilayer Networks

The approaches discussed earlier have primarily focused on single graphs, but real-world scenarios often involve enriched graphs with complex relationships. This complexity is exemplified by vertices displaying multiple relationships, which can be viewed as distinct layers within an underlying graph. These layers can be represented in a Multilayer Network (MLN), where distinct relationships are depicted as various layers within the MLN.

MLN-Subdue [16] employs a decoupling-based strategy specifically designed for Homogeneous Multilayer Networks (HoMLN). Inspired by substructure discovery techniques for single graphs like M/R Subdue [18], this approach aims at uncovering substructures within HoMLN wherein each layer is independently processed without aggregating the layers into a single graph. This helps preserve the integrity of the Multilayer modeling and its inherent structure. Additionally, the Decoupling approach affords the flexibility to analyze specific subsets of layers within the Multilayer Network. This enables tailored analysis on selected layers without the need to consider the entire combined structure. The algorithm is implemented using the MapReduce framework and can handle any arbitrary number of layers in the Homogeneous Multilayer Network. Each layer is independently processed in parallel, while the substructures generated for each layer are composed after each iteration to identify substructures spanning within the layers of the MLN. The core elements of graph mining, including substructure generation, the composing of substructures across layers to generate structures spanning multiple layers, removal of duplicates, and counting of isomorphic substructures, have all been integrated into the MapReduce paradigm.

The MLN-Subdue approach employs a ”divide and conquer” methodology to generate substructures within individual layers independently. The substructures generated for each layer are composed after each iteration to identify those spanning the entire MLN. The process involves independent discovery of substructures of size k within each layer, starting with k equal to 1. Subsequently, a composition function is applied to determine substructures of size k present across all layers. The composition function is applied after each iteration and not when substructures of all sizes within each layer have been identified. The composition algorithm is based on the principle that within a Homogeneous Multilayer Network, every connected substructure is associated with a shared vertex or node that acts as a link for edges originating from various layers.

However, this approach had its limitations. There is no provision for partitioning the layers of the MLN in case they are very large. Also the composition is applied in each iteration, the input to the composition function comes from the unconstrained expansion of intra-layer edges. The composition function has no means to limit this input. This can reduce performance when the expansion generates a large number of substructures.

Drawing inspiration from previous research on graph mining, we aim to present a two substructure discovery algorithm for a Homogeneous Multilayer Network (HoMLN) tailored for a MapReduce distributed environment. We use range partitioning for the creation of initial partitions. We propose two composition algorithms: i) after each iteration (HoICA), and ii) after all iterations (HoSCA). Then we can compare the trade-offs of these two approaches at the opposite ends of the composition spectrum.

CHAPTER 3

PRELIMINARIES

In this chapter, we will introduce the definitions and concepts used in the rest of the thesis. We will then discuss the prerequisites for substructure discovery and the various aspects that are involved in this process.

3.1 The Graph Model

A graph is a data structure made up of two sets - nodes (or vertices) and edges. The nodes represent entities or objects, and the edges represent the relationships or connections between the nodes. Graphs are used to represent several types of networks and model complex aspects of real-world data where relationships among entities are presumed to be important. For example, in a social network each user (entity) is represented by a node and the connections between users (relationships) are represented by edges.

By modeling data as a graph, we can analyze and extract insights from the relationships between the entities. Nodes and edges can have labels which are used to assign attributes to the nodes and edges. The use of labels in graphs allows for more complex and nuanced representations of data, enabling more sophisticated analysis and insights to be derived from the graph.

3.1.1 Types of Graph Models

1. **Simple graph:** A simple graph (also called a single graph or monoplex) is a type of graph that represents entities as nodes and relationships as edges.

Generally, nodes have unique numbers, but labels of nodes and edges do not need to be unique. This model of a graph does not allow loops or multiple edges between nodes, making it easy to understand and analyze. Despite its simplicity, this model is widely used as it is adequate for many purposes. But by using this model, we cannot have multiple edges and labels to represent multiple entity and relationship types. These limitations can be mitigated by using an attributed graph.

2. **Attributed graph:** An attributed graph (also called a multigraph) can capture multiple entity types and multiple relationships between entities as it allows multiple edges between the same pair of nodes, as well as loops (same start and end node). Multiple labels can also be associated with the same nodes and edges. As a model it is more expressive than a simple graph but is also more complex. This additional complexity can make it more difficult to analyze and interpret the graph. Additionally, algorithms that work well on simple graphs may not be applicable or efficient on multigraphs. This brings us to multilayer networks

3. **MultiLayer Network:** A MultiLayer Network (or MLN) is a network of simple graphs that consists of multiple layers, each representing a different relationship between entities based on a specific feature. The layers may include entities of the same or different types and can also be related to each other. The purpose of using an MLN is to simplify the representation from an attributed multigraph with multiple types of entities and relationships by separating them into distinct layers with each layer being a simple graph. This approach can enhance clarity in understanding and processing the data. Additionally, the algorithms that already exist for simple graphs can be utilized for MLNs as well.

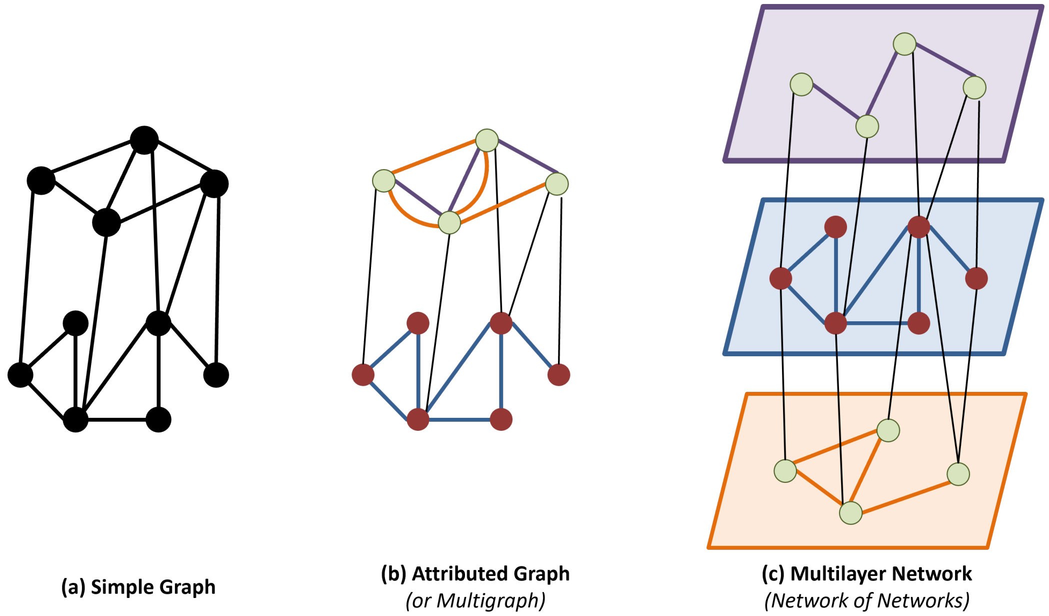


Figure 3.1: Different Types of Graph Models

Figure 3.1 shows the different types of graph models discussed above. Figure 3.1(a) shows a simple graph with only one type of nodes and edges without any labels. Figure 3.1(b) shows an attributed graph with multiple node and edge types, with different colors representing distinct types. It also has multiple edges between two nodes (colored light green). Lastly, Figure 3.1(c) demonstrates a multilayer network where the attributed graph in Figure 3.1(b) is separated into different layers. Each layer becomes a simple graph and has information about a single type of entity and relationship. As there were three types of relationships (represented through orange, blue, and purple colored edges), the resulting MLN has three separate layers. Additionally, note that a Hybrid MLN is generated as the first and third layers have the same node types but different relationships.

3.2 Input Graph Representation

The choice of graph representation can have a significant impact on performance of a substructure discovery algorithm, both in terms of memory usage and execution time. The characteristics of the graph such as its size, density of the edges, and the types of attributes associated with the nodes and edges being processed need to be considered before deciding the representation to be used.

Widely used graph representations in mining algorithms include adjacency matrix, adjacency list and an edge list. The adjacency matrix representation, which is a square matrix that depicts the connections between the vertices, offers a very efficient way for checking whether two vertices are adjacent in constant time. However, when dealing with sparse graphs, it can result in a significant waste of memory as most entries in the matrix will be 0. On the other hand, the adjacency list representation is more space-efficient compared to the adjacency matrix [38], making it a better option for sparse graphs. Edge list representation represents the graph as a list of edges and is easy to construct and manipulate. As most real-world datasets tend to be sparse [39], we have used the adjacency list representation along with an edge list.

In this thesis, we will focus on labeled graphs with vertex and edge labels. Each vertex has a vertex label and a unique vertex identifier (vertex id). Each edge has an edge label and is directed. We can use either undirected or directed edges in our approach, but we will only consider directed edges in this thesis. This is because directed graphs represent relationships explicitly using directions. Undirected edges can be accounted for by assuming them to have bidirectional directionality, meaning they have a direction in both ways. This leads to the creation of multiple edges between nodes. Vertex and edge labels are not assumed to be unique.

Figure 3.2 shows an example input graph relevant to this thesis. The 5 vertices have 5 unique vertex id's (1-5) and 3 non-unique vertex labels (A, B and C). The 6 edges have a direction (from source vertex to destination vertex) as denoted by the arrow and 4 non-unique edge labels (i, x, y and z).

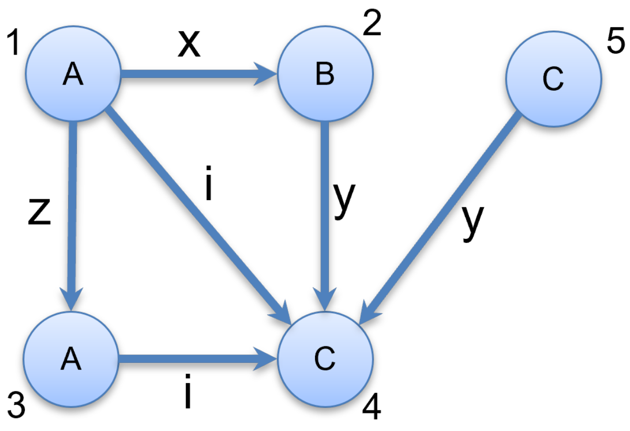


Figure 3.2: Input Graph

3.2.1 Edge list

In our approach, the input graph is represented as an unordered list of edges or 1-edge substructures. Each edge is represented by a 5-element tuple ⟨edge label, source vertex id, source vertex label, destination vertex id, destination vertex label⟩ which captures the direction of the edge as well. Table 3.1 displays the 1-edge substructures of the input graph shown in Figure 3.2.

Our representation is versatile and can be expanded to accommodate multiple edges by including an edge identifier in the edge representation to differentiate between multiple edges connecting the same two nodes. In this representation, a k-edge substructure, is represented as a list of k 1-edge substructures. The graph is stored as a text (ASCII) file where each line corresponds to a 1-edge substructure and serves as the input to our algorithm.

3.2.2 Adjacency List

An adjacency list is a way of representing a graph in which each vertex is associated with a list of edges that contain that vertex. Given a vertex, its adjacency list is the set of edges that connect that vertex to its neighbors. For each vertex, the number of edges is equal to the sum of the in and out degrees of that vertex. The adjacency list representation allows us to lookup all the edges incident on a chosen vertex and expand the substructure from that vertex by one edge. Table 3.2 demonstrates the adjacency list associated with all the vertices present in the input graph shown in Figure 3.2.

Edge List
(x,1,A,2,B)
(z,1,A,3,A)
(i,1,A,4,C)
(y,2,B,4,C)
(i,3,A,4,C)
(y,5,c,4,c)

Table 3.1: Input Graph Representation

Vertex ID	**Adjacency List**
1	(x,1,A,2,B), (z,1,A,3,A), (i,1,A,4,C)
2	(x,1,A,2,B), (y,2,B,4,C)
3	(z,1,A,3,A), (i,3,A,4,C)
4	(i,1,A,4,C), (y,2,B,4,C), (i,3,A,4,C), (y,5,c,4,c)
5	(y,5,c,4,c)

Table 3.2: Adjacency List

3.3 Graph Partitions

We may encounter a situation when the layer in a given MLN is too large to fit into the main memory of a single machine. If need be, we can partition the layer L_i into p partitions (L_i^1, L_i^2,.., L_i^p) such that each partition can fit within the main memory.

The partitions are generated by using range-based partitioning [18]. Range info is used to divide the adjacency list into partitions using a contiguous range of vertex ids (range, by definition, has to be contiguous, but all vertex ids in the range may not be present). The number of partitions and their size can be customized. With range partitioning, the adjacency lists for partitions are disjoint, meaning that each vertex id and its adjacency list belong to only one partition. Edges that connect two partitions are replicated in all the adjacency list partitions. During expansion, the same substructure can be expanded by multiple adjacency partitions. However, since adjacency list of partitions are disjoint on vertex ids, each substructure is only expanded once on a given vertex id, using a single adjacency list. Figure 3.3 illustrates adjacency list of partitions generated using range partitioning scheme on the input graph shown in Figure 3.2. Here, vertex ids 1 to 3 are assigned to partition L_i^1 and vertex ids 4 and 5 are assigned to partition L_i^2. The red edges connect both partitions and these edges are repeated in multiple adjacency lists of partitions. Adjacency list of partitions are shown in Table 3.3. Edges (i,1,A,4,C), (y,2,B,4,C) and (i,3,A,4,C) are present in both the adjacency lists of partitions.

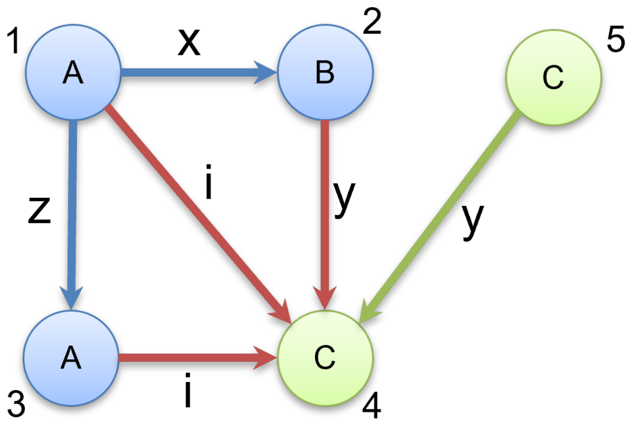


Figure 3.3: Partitioned Input Graph

Vertex ID	Adjacency List
1	(x,1,A,2,B), (z,1,A,3,A), (i,1,A,4,C)
2	(x,1,A,2,B), (y,2,B,4,C)
3	(z,1,A,3,A), (i,3,A,4,C)

Vertex ID	Adjacency List
4	(i,1,A,4,C), (y,2,B,4,C), (i,3,A,4,C), (y,5,c,4,c)
5	(y,5,c,4,c)

Table 3.3: Adjacency List of Partitions

3.4 Graph Expansion

The process of discovering the best substructures for compressing a graph involves generating increasing-sized substructures and counting the isomorphic substructures. We start from a 1-edge substructure and expand it by adding an edge on each vertex in that substructure in an iteration. Each expansion becomes a separate substructure. The above expansion is done without any constraints, so that each substructure can grow into every possible larger substructures, independent of other substructures. This ensures completeness of substructure generation process. However, this unconstrained expansion results in the generation of duplicate substructure instances, which must be removed to prevent incorrect counting. Figure 3.4 shows how expansion leads to duplicates. If two substructures have the same set of vertices and edges and the exact connectivity, they are duplicates. To ensure accurate identification of duplicates and differentiate them from isomorphic substructures, we introduce the concept of a canonical substructure and a canonical instance.

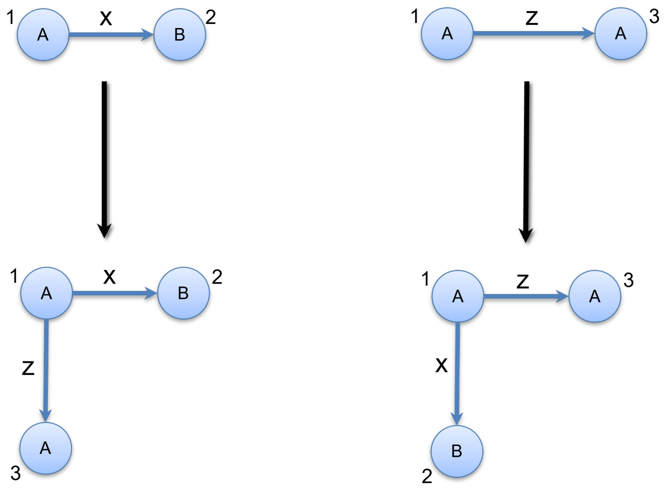


Figure 3.4: Duplicate Generation

3.4.1 Canonical Representation

We use lexicographic ordering on labels. Firstly, we order the edges on the edge label. In case of the edges have the same edge labels, we order them based on the source vertex label. If source vertex labels are also the same, then we further order them based on the destination vertex label. If edge labels and vertex labels are all identical, then we order them based on the source vertex ids. Finally, if source vertex ids are also the same, then we order them based on the destination vertex ids. Using this lexicographic order for each 1-edge, a substructure can be uniquely represented, which is called a canonical k-edge instance. If two k-edge substructures are duplicates, they must have the same ordering of their vertex ids and therefore, will have the same canonical k-edge instance. If we convert the duplicates generated in Figure 3.4 to canonical instances, they will be the same as illustrated in Figure 3.5.

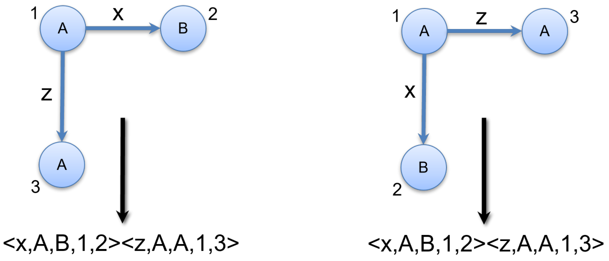


Figure 3.5: Example of Canonical Instances

The canonical representation of instances is helpful in identifying and removing duplicate instances. However, counting the frequency of isomorphic substructures requires a canonical form that does not rely on vertex IDs. Therefore, we use a canonical form of the substructure that excludes vertex IDs, derived from the canonical instances.

3.4.2 Graph Isomorphism

Graph isomorphism refers to the concept of two graphs being structurally identical, with a one-to-one mapping of their vertices, preserving edge connections and edge labels, and edge orientations. Exact matching of substructures is needed to compute frequent substructures accurately. Isomorphs have the same vertex labels and edge labels but different vertex ids (as they are different instances). To identify them, we need to create a canonical substructure from the canonical instance. For this vertex ids are changed from absolute to a relative ordering. Any two exact substructures will have the same relative ordering of vertex ids [18]. This is used to identify and compute the frequency of a substructure.

As the canonical instance already follows a lexicographic ordering, we can construct a canonical k-edge substructure by ordering the unique vertex ids in the order of their appearance in the canonical instance. Thus, the canonical substructure can be derived by replacing every vertex id with their relative position in the instance. The resulting canonical substructure allows us to identify isomorphs easily. Figure 3.6 provides an example of how a canonical substructure is created from the canonical instance. Note that the isomorphs have different canonical instances, but their relative positions following the canonical instance are the same. The relative positioning of vertex id (1, 4, 2) for the first instance and (3, 4, 2) for the second instance reduces to (1, 2, 3). This leads to both having the exact same canonical substructure.

The process of canonical label generation is crucial in our approach, as it enables us to identify duplicates and isomorphic instances. In this thesis, we rely on the concept of a canonical instance and a canonical substructure to differentiate between duplicates and isomorphic substructures, respectively.

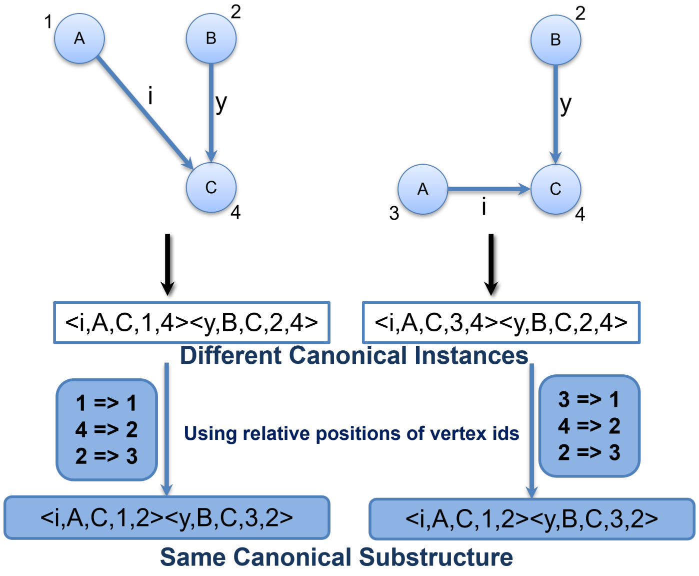


Figure 3.6: Example of Graph Isomorphism

3.5 Metrics for Ranking

In the context of substructure discovery, the definition of an interesting substructure may vary based on the context and the goal of the analysis. The choice of metric depends on how we define an interesting substructure. The goal may be to find the most frequent substructure or the ability of a substructure to best compress a given graph. Commonly used evaluation metrics to rank substructure and measure their importance are compression and frequency. The Minimum Description Length (MDL) [40] and frequency are popular ranking metrics for graph mining, with the former emphasizing the substructure's ability to compress the entire graph and the latter focusing solely on the number of occurrences of its instances. Counting the number of instances is required for both.

3.5.1 Minimum Description Length

Minimum Description Length (MDL) is a metric used in the context of graph mining for ranking substructures. The MDL principle evaluates the importance of a substructure based on how well it can compress the entire graph. The description length of a graph G is the number of bits needed to encode the graph. For a substructure S that occurs in graph G, MDL is computed using the formula:

$$MDL = \frac{D_L(G)}{D_L(S) + D_L(G|S)}$$

where $D_L(S)$ = description length of the substructure being evaluated

$D_L(G|S)$ = description length of the graph where every substructure S has been replaced by a node in the graph

$D_L(G)$ = description length of the original graph

The graph is compressed by replacing each instance of the substructure by a node. The substructure that minimizes $D_L(S) + D_L(G|S)$ has the most ability to compresses the graph and is considered the best substructure in the graph. Both the size of the substructure and the number of instances have a bearing on compression. Therefore, MDL can highlight the importance of a substructure based on its compression abilities. Instead of bits, we compute MDL using the number of vertices and edges called DMDL (Database Minimum Description Length) proposed in [20].

3.5.2 Overlap-independent Frequency

In this metric, all overlapping instances of substructures are considered as independent occurrences and are taken into account for counting. For example in Figure 3.7, the frequency would be computed as 4 even though there is a significant amount of overlap in the instances. In the top two substructures, vertex ids 1 and 4 overlap, while in the bottom two substructures, vertex ids 3 and 4 overlap. Due to such overlaps, the frequency count does not provide a true representation of frequency when compared to disjoint substructures that have identical vertex and edge labels. Therefore, this frequency measure is not typically used.

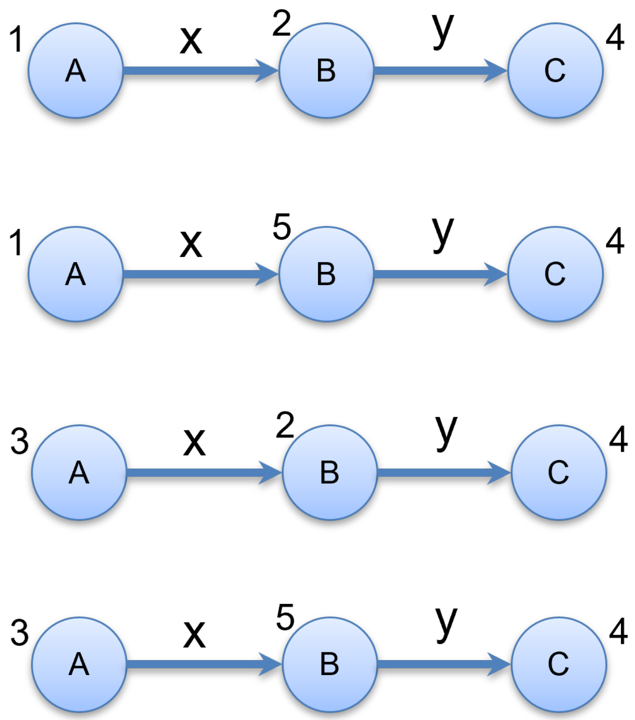


Figure 3.7: Example of Overlapping Instances

3.5.3 Overlap-cognizant Frequency

In this metric, overlapping instances of substructures are not considered as independent occurrences and thus are not counted. To count non-overlapping instances, we adopt the Most Restrictive Node (MRN) metric proposed in the literature [41,42].

The MRN is the node in the substructure that has the least number of occurrences in the graph. The number of instances of a substructure is then computed as the product of the number of occurrences of the MRN in the graph and the number of occurrences of the other nodes in the substructure that are not adjacent to the MRN. Overlap-cognizant frequency is then computed using the maximal number of non-overlapping instances for a canonical substructure. For example, in Figure 3.7, we have F(A) = [1,3], F(C) = [4] and F(B) = [2,5]. As min(2,1,2)=1, the MRN is node C and frequency of the substructure is 1.

However, overlapping instances may be important in certain contexts. Thus, we also keep track of overlapping instances and utilize both sets of information to compute the frequency and Minimum Description Length (MDL) for a given substructure.

We have comprehensively reviewed the existing literature relevant to this thesis. We discussed key concepts, methodologies, and findings from prior studies that contribute to our understanding of the process of substructure discovery. In the next chapter, we shall discuss the design of our composition algorithms.

CHAPTER 4

DESIGN

Substructure discovery algorithms identify recurring patterns within larger, more complex structures. In the context of a graph database, substructure discovery involves searching for frequently occurring subgraphs or patterns within the dataset. It should be able to find structures of a given size in an arbitrarily large graph. In this chapter, we discuss the design of our substructure discovery algorithm for multilayer networks focusing on our approach and the challenges faced.

4.1 Overview of Design

Our approach employs an iterative Decoupling-Based algorithm to identify the substructures within a homogeneous multilayer network. We systematically generate substructures of increasing sizes within each layer (henceforth called intra-layer). Since we are dealing with MLNs which can have multiple layers, we also need to find substructures which exist across layers, called inter-layer substructures. Thus, we need to use intra-layer substructures independently generated for each layer to generate inter-layer substructures that exist across the layers. Throughout this process, we eliminate any generated duplicates, count the number of isomorphic substructures, and apply a ranking metric. We then apply the beam (top-k) to restrict our search space and only use these beam substructures as candidates for expansion in the next iteration. This entire process iterates until either a designated substructure size is achieved or no more substructures can be generated.

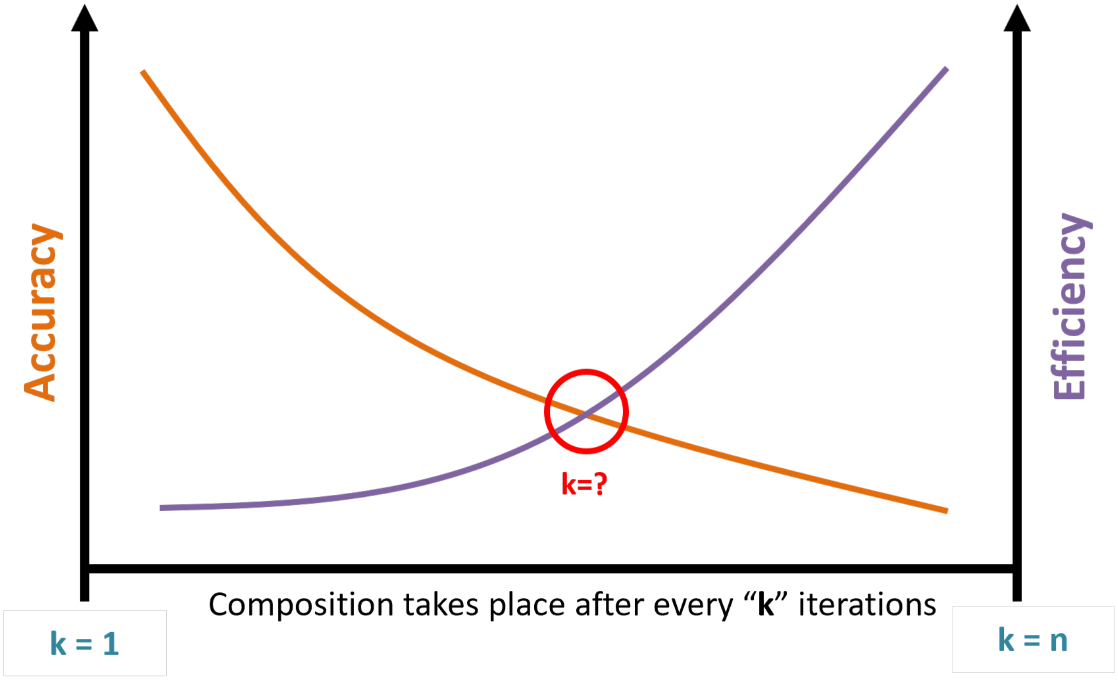


Figure 4.1: Accuracy and Efficiency Trade-off using 'k'

The primary emphasis of this thesis is on evaluating two extreme approaches for substructure discovery in HoMLNs: i) applying composition function after each iteration (multiple times) to generate inter-layer substructures and ii) applying the composition function at the end (only once) with respect to accuracy and efficiency trade-off. The substructure discovery algorithm follows an iterative process and generates substructures of increasing size in each iteration. The composition function can thus be applied in each iteration or at the end of the substructure discovery process for each layer. The decoupling approach also allows for the possibility of conducting this composition at various iterations of the algorithm, offering a trade-off between efficiency and accuracy. This trade-off is depicted in Figure 4.1, where the X-axis represents iterations of the algorithm where we can apply the composition. Y axis shows the expected accuracy and efficiency.

For this thesis, we apply composition at the two extremes, composing in each iteration(k= 1) and composing at end (k= n). We must discern and understand any potential differences in the composition algorithm between these two approaches.

The first approach, called the Iterative Composition algorithm (Ho-ICA), involves performing composition in each iteration. Here inter-layer substructures are composed in every iteration. We believe that this approach should not have any accuracy drop as all substructures (intra and inter) are evaluated using the metric and carried to the next iteration. We will validate this conjecture experimentally. But this comes at the cost of increased response time and computational resources due to the frequent composition.

The second approach, called the Single Composition Algorithm (Ho-SCA), focuses on generating final intra-layer substructures for each layer. After completing all layer-wise iterations for each layer, the results are used by the composition algorithm for generating the inter-layer substructures. This approach only involves doing composition once, providing better response time and computational efficiency. However, this efficiency comes at the cost of loss in accuracy, as certain information from previous iterations is missed. One of the challenges is to correlate layer characteristics with accuracy drop.

The important aspects of our design are:

Correctness: The composition algorithm achieves completeness by generating all possible inter-layer substructures. Simultaneously, for soundness of result, it focuses on identifying only correct inter-layer substructures to prevent the introduction of inaccuracies. The accuracy of the algorithm is then evaluated in relation to a ground truth, which serves as a reference point for the correctness of the algorithm.

Efficiency: Composition is done as efficiently as possible. The emphasis on efficiency aims to optimize the computational resources and reduce processing time while maintaining the accuracy and completeness of the generated inter-layer substructures.

Scalability: It is an outcome of the efficiency requirements in addition to using the MapReduce framework for each layer. To achieve scalability, we utilize the "Divide and Conquer" strategy to partition large layers that may exceed the memory capacity of a single machine. For an input Multilayer Network with n layers, each layer (L^n) is partitioned into p smaller partitions ($L^n_0, L^n_1, ..., L^n_P$).

Resource utilization: To harness parallel processing on layer partitions, we employ k processors to handle them simultaneously. The value of k can range from 1 to np, and it can be adjusted based on the availability of resources. For instance, if sufficient resources are at hand, we can scale up k as needed, extending it up to np (number of layers times number of partitions).

4.2 Challenges for Composing

The process of identifying substructures within a Multilayer Network (MLN) differs significantly from finding substructures within a single graph. Substructures in a Multilayer Network (MLN) can exist in several different ways, and it's important to consider various factors to accurately discover them.

1. Substructures can exist solely in one layer (intra-layer)
2. Substructures can exist across multiple layers (inter-layer)

In both of these scenarios, substructures can also extend across multiple layer-wise partitions (intra-partition), and this aspect must be effectively managed.

The algorithms should be capable of handling all these scenarios effectively, while also managing the elimination of duplicates during both the expansion and composition phases.

4.3 The Two Approaches

We propose two different approaches, both employing iterative Decoupling-Based approaches to substructure discovery. The primary objective is to process each layer of the Multilayer Network independently and in parallel. We use the decoupling-based approach discussed in Chapter 1 for substructure discovery. Since our focus is on large graphs, it is assumed that all the required information (layer data and its adjacency list) cannot be stored in the memory of a single machine. Therefore, partitioning the graph among multiple processors is essential. Our inspiration for this partitioning strategy comes from earlier work on MapReduce-based substructure discovery [18].

Now, we shall delve into the details of the two approaches.

4.3.1 HoICA: HoMLN Iterative Composition Algorithm

This approach generates substructures for layer in each iteration. Composition algorithm is applied in each iteration to generate missing inter-layer substructures. After each iteration, all substructures from the individual layers and composed substructures spanning across layers are combined. A ranking metric (like MDL) and beam are applied to generate the substructures for the next iteration. Each iteration corresponds to a particular size of substructures. In each iteration, the substructures are expanded by a single edge in their own layer, starting with size one substructures for first iteration. This generates all the intra-layer substructures for that size. Also in the first iteration, the composition function generates inter-layer substructures by expanding each layer into every other layer. In the subsequent iterations, composition function will generate composed substructures of size k+1, using the composed substructures from the previous iteration (size k).

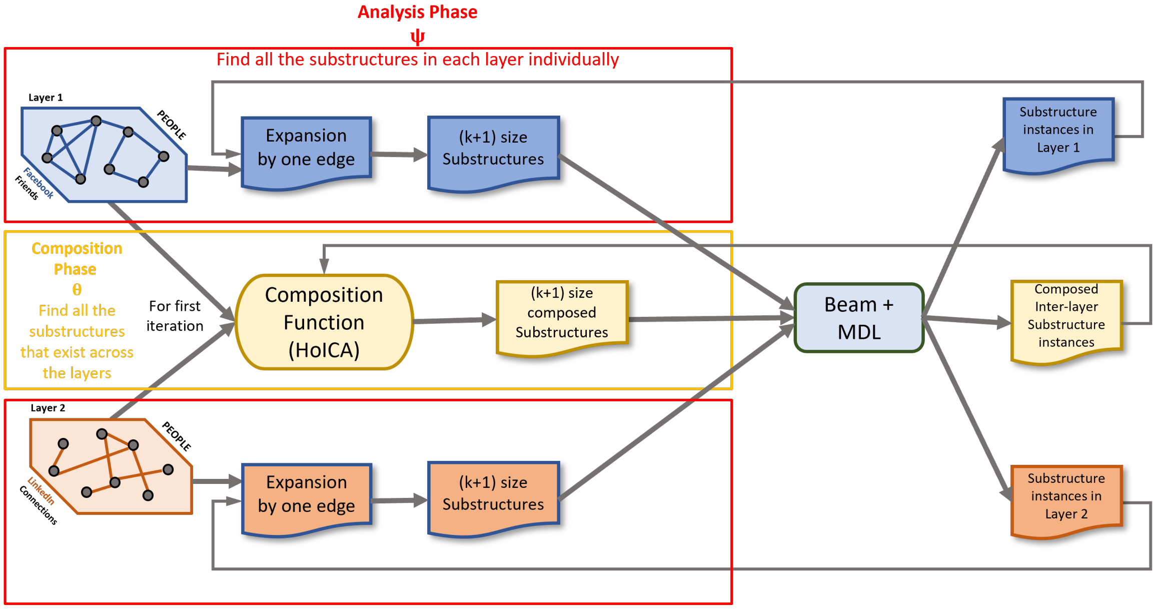

Analysis Phase
ψ
Find all the substructures in each layer individually
Layer 1
PEOPLE
Facebook
Friends
Expansion by one edge
(k+1) size Substructures
Substructure instances in Layer 1
Composition Phase
θ
Find all the substructures that exist across the layers
For first iteration
Composition Function (HoICA)
(k+1) size composed Substructures
Beam + MDL
Composed Inter-layer Substructure instances
Layer 2
PEOPLE
Linkedin
Connections
Expansion by one edge
(k+1) size Substructures
Substructure instances in Layer 2

Figure 4.2: Overview of the Decoupling Approach for Iterative Composition

Following these two steps, all the substructure instances from both are counted utilizing graph isomorphism and canonical labeling. Duplicates are also eliminated in this step. These instances are subsequently ranked based on a selected metric, such as Minimum Description Length (MDL) or frequency. To restrict the search space, a beam metric is applied, delimiting the exploration to the top beam substructures, which are used as candidates for expansion in the next iteration.

At the end of each iteration, the substructures from each layer and the inter-layer substructures are separated for the next iteration. This is critical and is consistent with the decoupling principle which stipulates that only layers are processed independently and without using the knowledge of the other layers. In the next iteration, substructures from each layer is expanded by an edge and in the next composition stage, inter-layer substructures are expanded with one edge and the process repeats. Note that composition can make use of information from both layers.

To illustrate this, consider we start with a k-edge substructure in each layer, with k = 1 for the first iteration. In this iteration, we systematically expand all vertices within the substructure independently (due to which duplicate instances are generated) by adding a single edge to each. Also the composition function is called, expanding the previously composed inter-layer instances using edges from each participating layer. The purpose here is to generate substructures that span multiple layers. Following these steps, we identify the beam substructures, which are then segregated into layer-wise and inter-layer substructure instances. These substructures are then directed to their respective destinations for the next iteration. The intra-layer substructures become inputs for our analysis function (Psi being the substructure analysis in this case), while the inter-layer substructures are passed on to the composition function (theta for substructure composition). This iterative process continues until a termination condition is met. Figure 4.2 provides a comprehensive visual overview of HoICA for substructure discovery for a Homogeneous Multilayer Network.

4.3.2 HoSCA: HoMLN Single Composition Algorithm

In this approach, our initial focus is to understand what portion of the correct inter-layer substructures can be generated as we are not doing it after each iteration. For each layer during a particular iteration, we expand these substructures by one edge, removing any duplicates that might be generated. Subsequently, we count the substructure instances through the use of graph isomorphism and canonical labeling. These substructures are then ranked based on a selected metric such as MDL or frequency, followed by the application of a beam to restrict the search space.

This iterative process is executed for all layers independently, producing layer-wise outputs of size k from each layer. Upon the completion of all layer-wise iterations,

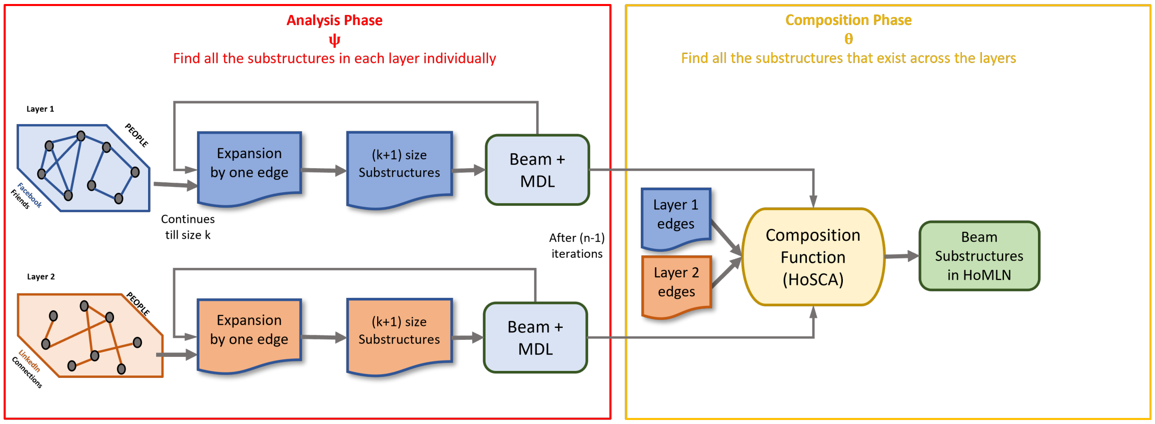


Figure 4.3: Overview of the Decoupling Approach for Single Composition

these outputs are directed to our composition algorithm. Its purpose is to identify substructures that extend across multiple layers. This composition algorithm specifically deals with substructures of size k and is responsible for generating inter-layer substructures of the same size (k), which is elaborated upon as the Single Composition algorithm in the following section.

The subsequent inter-layer substructures are then ranked using the same metric and evaluated in comparison to the intra-layer substructures, with elimination of any duplicates that may be generated. The key advantage of this approach lies in the fact that the composition is only performed once, rather than in each iteration. This leads to an efficient substructure discovery, although it does involve some trade-off in accuracy due to information loss from previous iterations. Figure 4.3 provides a comprehensive visual overview of HoSCA for substructure discovery within a Homogeneous Multilayer Network

Of the two approaches presented and analyzed in this thesis, it is important to understand their difference. In the iterative approach , composition is applied after each iteration to generate missing inter-layer substructures. As this is done after each iteration, our hypothesis is that the accuracy is preserved and should be the same as the ground truth. However, composition cost is incurred in each iteration increasing the overall cost of this approach as shown in Figure 4.1 red line. In contrast, the other approach applies the composition function only once at the end incurring less composition cost. This is illustrated in Figure 4.1 by the blue line. Again, our conjecture is that the accuracy will suffer in this approach due to missed inter-layer substructures at different iterations. This has better computation cost as composition is applied only once. As can be seen from Figure 4.1, there is an efficiency-accuracy trade-off between these two approaches.

The purpose of this analysis is to understand the accuracy/efficiency trade-off based on layer characteristics and graph characteristics within each layer. Once this is understood, it may be possible to determine which approach to use (and several alternatives for the iterative approach as well) based on the accuracy needed for the application. Our experimental results will validate the above points for a large number of data sets with diverse characteristics.

4.4 Discussion of Composing Approaches

As previously emphasized, for MLNs, focusing solely on substructures within layers is insufficient, as substructures spanning multiple layers will not be generated. To ensure accuracy and completeness of our approach, we use the decoupling approach by customizing a composition function that compensates for the missing (or not generated) substructures and inserting that into the substructure discovery process. This composition function needs to combine substructures from each layer to find those that exist across different layers. In a homogeneous multilayer network, all layers have the same set of nodes but different node connectivity. If there is a common node present in substructures from different layers, it suggests that these substructures can be connected to form substructures that exist in multiple layers. This insight forms the foundation of our combining algorithm. Consequently, by grouping all substructures generated from each layer according to their vertex IDs (nodes), we can effectively include all edges that traverse across different layers, allowing us to compose edges that share a common vertex ID across different layers.

Now, lets take a detailed look at the two algorithms used for substructure discovery in HoMLNs.

4.4.1 Single Composition Algorithm (HoSCA)

The primary objective of this algorithm is to understand whether all missing substructures of size k that satisfy the metric can be generated at the end using substructures of size k generated so far. As we have indicated, our conjecture is that waiting until the last iteration may hinder the generation of all substructures of size k that satisfy the metric and beam. of course, this composition can also be applied after each iteration as discussed earlier incurring additional computation. The key motivation for this approach is to study the accuracy-computation trade-off.

The underlying rationale for this composition is that when we join two connected subgraphs, one each from two (or more) layers, that share a common vertex id, the resulting subgraph would be connected and further will be a valid inter-layer substructure instance. Figure 4.4 shows how two connected subgraphs from different substructures joined on a common vertex id will lead to a connected substructure instance. Thus, when we combine pairs of subgraph instances for two layers (and more for greater than two layers), with their combined size adding up to k, we would generate composed inter-layer substructures of size k.

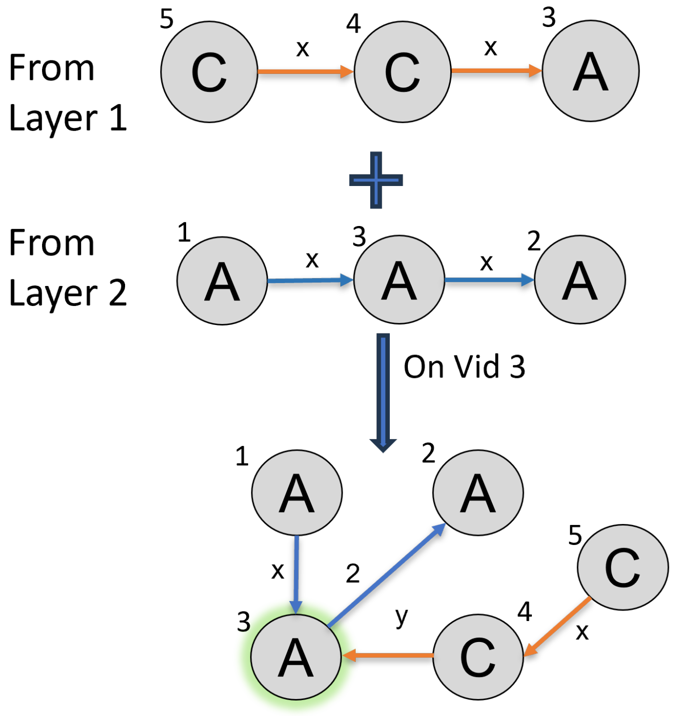


Figure 4.4: Joining subgraphs on a common vertex id

The input of the algorithm is a set of intra-layer substructures, each of size k, where k can be any positive integer. The initial step involves generating connected subgraphs of sizes ranging from 1 to *k-1* from each of the participating intra-layer substructures. This process is accomplished by iteratively breaking down each substructure instance into different sized substructure instances as outlined in Algorithm 1. To achieve this, we begin by creating substructures of size 1 and subsequently expand them up to a size of *k-1*, adding a single edge in each iteration. The resulting output includes all possible connected subgraphs of the specified size range. This approach allows us to circumvent the generation of disconnected subgraphs, avoiding connectivity check adding additional computation.

Algorithm 1 Algorithm to split connected subgraphs into smaller connected subgraphs

```
Input: Substructures of size k from N layers ($S_k^N$) represented as a sequence of edges
    $< E_1; \ldots ; E_k >$, where $E_i$ is $< E_L, V_{id}^s, V_L^s, V_{id}^d, V_L^d >$
Output: resultSet containing Connected subgraphs of size 1 to $k - 1$
Initialize $tempSet$ and $resultSet$ to empty sets
for each substructure ($S_k^N$) do
    Generate $Size1$ set by breaking the substructure ($S_k^L$) into single edges
    $SameSizeSubs \leftarrow Size1$
    for $i = 1$ to $k - 1$ do
        for each edge $E_x$ in $Size1$ do
            for each substructure $S_y$ in $SameSizeSubs$ do
                if $E_x \in S_y$ then                          ▷ Duplicate Edge
                    continue
                end if
                if $V_{id}^x \cap V_{id}^y \neq \varnothing$ then        ▷ $V_{id}$ is a set containing vertex ids
                    $S_{new} \leftarrow E_x + S_y$          ▷ Append edge $E_x$ to substructure $S_y$
                    Tag $S_{new}$ with its layerID $L_{ID}^n$
                    $tempSet \leftarrow S_{new}$
                end if
            end for
        end for
        $resultSet \leftarrow tempSet$
        $SameSizeSubs \leftarrow tempSet$
        clear $tempSet$
    end for
end for
```

Furthermore, this strategy significantly restricts the number of generated subgraphs, as they typically amount to less than the total count of all possible subgraphs of size 1 to k. In the case of a substructure of size$k$, the number of potential subgraphs is calculated as the sum of combinations, specifically: $C_1^k + C_2^k + ... + C_{k-1}^k$. A substantial portion of these potential subgraphs would be disconnected. Thus generating only connected subgraphs would greatly reduce the number of subgraphs we generate.

The next phase of the algorithm involves combining the generated instances based on their vertex ids by applying the *Combine* function for each vertex id. This recursive function examines all possible combinations of the connected subgraphs originating from different layers that combine to a k-edge inter-layer substructure. The composition process requires the grouping of subgraphs generated in the previous step based on vertex IDs. This grouping ensures that the composition algorithm can efficiently identify and combine subgraphs that will generate inter-layer composed substructure instances. Thus the setup of input parameters for the *Combine* function becomes essential. We utilize the MapReduce paradigm to bring subgraphs that share a common vertex id together. The exact process is discussed in detail in the next chapter.

The inputs to the combine function are:

1. Subgraphs: A collection comprising of sets where each set contains substructures from one layer. In each set, the substructure size ranges from 1 to *k-1*.
2. Accumulator String: This is a string used as an accumulator to gradually build the inter-layer substructure until it reaches the desired size *k*.
3. Combinations Set: This is a set used to store the composed substructures.
4. Size: This parameter denotes the target size of the substructures to be generated, which is k in this context.

As the input subgraphs are connected, the accumulator string, which generates and stores the inter-layer substructure instances, will consistently maintain a valid connected substructure. Furthermore, since the sets in the *Subgraphs* contain substructures from different layers, we can be certain that the substructure generated is inter-layer substructure, meaning it includes edges from different layers.

Algorithm 2 describes the recursive combine function.

Algorithm 2 Combine Function for generating inter-layer substructures of size k

```
Input: Subgraphs: Collection of sets containing layer-wise substructures of $size_1^{K-1}$;
    accumulator: Initially an empty string, accumulates edges in each recursive call;
    combinations: Set to store composed substructures; size: Size of substructures
    to generate
Output: Composed substructures added to combinations set
Initialize combinations $\leftarrow \varnothing$
for each vertexid in currentSet do
    Generate Subgraphs List for vertex id
    accumulator $\leftarrow$ empty string
    Call combine (Subgraphs, accumulator, combinations, size)        $\triangleright$ size = k
end for
Recursive Combine Function
islast $\leftarrow$ Boolean(Subgraphs contains only one set)
currentSet $\leftarrow$ first set in Subgraphs
for each substructure in currentSet do
    newSubstructure = accumulator + substructure
    if islast is true then
        if size of newSubstructure = size then
            Add to combinations set
        end if
    else
        newSubgraphs = Subgraphs $-$ currentSet
        combine (newSubgraphs, newSubstructure, combinations, size)
    end if
end for
```

4.4.1.1 Duplicate Substructure Generation

A consequence of composition done independently on each vertex id of a substructure is that it results in the generation of duplicates. We've discussed the process of duplicate generation due to expansion, along with duplicate elimination in Chapter 3. Thus we shall now focus on duplicates generated during composition.

Duplicates Generated during Composition: Since the composition process operates on all vertex ids, there's a possibility that the same substructures may be generated during composition multiple times if they share more than one common vertex identifier. Figure 4.5 shows how the same substructure instance is generated by composing on vertex id 1 and on vertex id 3. Unfortunately, avoiding these duplicates is not possible since we apply composition for each vertex id independently. Hence, we eliminate them to ensure correctness of the results. Failure to do so can result in incorrect counting. We do this elimination before the substructure counting step utilizing the same approach discussed in Chapter 3.

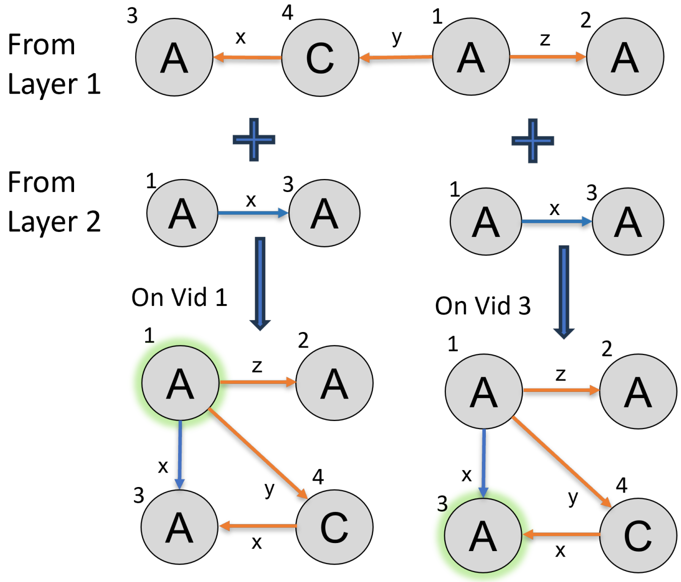


Figure 4.5: Duplicates Generated During Composition

4.4.2 Iterative Composition Algorithm (HoICA)

The main drawback with the Single Combination Algorithm (SCA) lies in the loss of information between iterations. This is especially true for larger size substructures (greater values of k), as we lose increasingly more information as more iterations pass. Although we generate k-size substructures using all possible sizes of substructures from each layer, it is not the same as doing it after each iteration. This is because some substructures of a specific size that should have come out at some iteration may not come out in this process and we cannot regenerate them at the end.

An interesting question that comes up is whether we can use the single composition algorithm after each iteration. We cannot as there is a subtle difference in the way composition is applied. For the SCA approach, two (or more) k-size substructure are used to create a k-size inter-layer substructure. In contrast, in the iterative composition, (k-1)-size substructures from participating layers are used to compose a k-size inter-layer substructure. The former one is expansion whereas the latter is combining. Combining takes more effort as we need to first split the substructure and then combine whereas expansion does not require splitting. The splitting was necessary in single composition as we were composing at the end of all iterations and no longer had the iteration-wise results. However, that is not the case for iterative composition. Hence, combining is more computationally expensive than expanding and should not be used for iterative composition. However, both generate duplicates which need to be eliminated.

Composing after each iteration provides us access to the results from the previous iteration. This enables us to expand the size *k-1* substructures from the previous iteration, using adjacency lists from other layers to create composed substructures in the current iteration. Thus, the iterative algorithm mimics the substructure discovery algorithm by generating size *k* composed inter-layer substructure instances using size *k-1* composed instances from the previous iteration.

Moreover, having composed instances at the end of each iteration offers another advantage. These composed instances can now be incorporated into the set of beam substructures. This enhances our ability to prune the search space more effectively without sacrificing any relevant substructures, enhancing the overall accuracy of the process.

The overall process for substructure discovery remains largely the same. We start by expanding each layer to generate intra-layer substructures. However, instead of proceeding to the next iteration, we've moved the composition phase inside each iteration. As previously discussed, we will expand the composed substructures from the previous iteration into other layers to generate size k intra-layer substructures. In the first iteration we expand single-edge substructures from each layer into the other layers. This provides us with the initial set of composed substructures. It is worth noting that when expanding a substructure from layer N into layer M, the result is the same as expanding a substructure from layer M into layer N. This insight can be (and is being) used in preventing the generation of duplicate composed instances. This operation only needs to be performed once, and the choice of which layer to initiate the expansion from is arbitrary, as the composed instances are identical.

The input of the algorithm is a set of composed substructures from previous iteration, each of size *k-1*, where k is at least 2. We then expand these substructures using adjacency lists of other layers, to get inter-layer composed substructures of size k. The inter-layer substructures, in addition to the intra-layer substructures generated in the current iteration, are forwarded to the metric evaluation process followed by beam application. Given that the beam substructures now consist of both intra-layer and inter-layer substructures, they are appropriately seperated and directed to their respective tasks in the subsequent iteration.

Algorithm 3 describes the composition function used to compose substructure after each iteration.

4.5 Dealing with Large Layer Graphs

In both of our approaches, we employ range-based partitioning to divide a single layer into multiple partitions. This partitioning strategy is essential because

Algorithm 3 Iterative Approach Composition Function

```
Input: ($S_{k-1}$): Set of composed substructures of size $k-1$ from previous iteration,
    where $k$ starts with 2, and each substructure is represented as a sequence of
    edges $< E_1; \ldots ; E_k >$, where $E_i$ is $< E_{Label}, V^s_{id}, V^s_{Label}, V^d_{id}, V^d_{Label} >$; $AL^m_p \ldots AL^n_p$:
    Adjacency List partitions for N layers
Output: Composed inter-layer substructures of size $k$
for each $(k-1)$edge instance $I_{k-1} \in (S_{k-1})$ do
    for each vertex-id $v \in I_{k-1}$ do
        $EL_v \leftarrow \{v \in AL^m_p \bigcup \ldots \bigcup AL^n_p\}$        ▷ edge list of $v$ from every layer
        for each edge $e \in EL_v$ do
            if  $e \notin I_{k-1}$ then
                $I^{composed}_k \leftarrow I_{k-1} + e$        ▷ Append edge $e$ in lexicographical order
                $S^{composed}_k \leftarrow I^{composed}_k$        ▷ Add to composed set $S^{composed}_k$
            end if
        end for
    end for
end for
```

the adjacency list of a single layer may be too large to fit entirely into the main memory. This is especially true for very large homogeneous MLNs. Consequently, each layer is divided into p partitions, and the user has the flexibility to choose p based on the layer size and the available computational resources.

It's important to note that the number of partitions is based on the value of p, which can be chosen by the user. For maximum parallelism, all p partitions can be loaded and processed by individual processors simultaneously. However, when resources are limited, processing all p partitions in parallel may not be feasible. In such cases, multiple partitions may go to a single processor. Ideally, our goal is to have the number of partitions equal to the number of available processors to maximally optimize parallel processing. Another scenario favoring more partitions arises in the case of uneven partitions. Range-based partitioning, with fixed ranges, may result in some partitions being larger than others due to graph characteristics. Uneven partitions can lead to load imbalance, where certain processors have more work to do than others, resulting in inefficient resource utilization. In this case, having more partitions than the number of processors can prove beneficial for achieving better load balancing.

4.6 Correctness

As previously discussed, we anticipate HoICA to have complete accuracy. To prove its correctness, we shall employ induction to demonstrate that we consistently generate identical substructure instances as the ground truth in each iteration. This ensures that we do not overlook any instances. Given that the beam remains consistent with the ground truth for every iteration, we can confidently assert that HoICA attains full accuracy in substructure discovery.

In the case of HoSCA, where composition does not occur between iterations, we lose some substructures between iterations that cannot be generated. Since composition takes place after all iterations, certain substructures may go unnoticed in their individual layers. Notably, a subgraph may form part of a frequent substructure in the Multilayer Network (MLN) but may be infrequent within its specific layer. A counterexample is presented to show how a substructure may be overlooked in HoSCA.

Additionally, we will verify these observations through experimental analysis in Chapter 6, comparing our results with the ground truth to validate the accuracy and completeness of our approach.

4.6.1 HoICA

Lemma Statement: For any substructure s in the input data, the iterative algorithm *HoICA* generates s as an element of S, where S represents the set of substructures generated by *HoICA*.

We will use proof by induction. We can show that for any iteration k, HoICA correctly generates all results for the next iteration *(k + 1)*. Following is the :

Base Case:

For iteration 1, HoICA used the size 1 substructures (i.e. the edge list) of each individual layer and generates size 2 composed inter-layer substructure instances using the adjacency lists of the other layers. As it utilizes all edges in the MLN to generate these instances, it will generate every size 2 composed inter-layer instance possible. For two layers L_1 and L_2,

1. It generates all L_1 intra-layer substructures by expanding L_1 substructures using L_1 adjacency list.
2. It generates all L_2 intra-layer substructures by expanding L_2 substructures using L_2 adjacency list.
3. It generates inter-layer substructures using L_1 substructures and expanding them using L_2 adjacency list.
4. It generates inter-layer substructures using L_2 substructures and expanding them using L_1 adjacency list.

Thus HoICA correctly generates all size 2 substructure instances in the first iteration *(k = 1)*.

Inductive Step:

Lets assume that for some arbitrary iteration k, the algorithm correctly generates all instances.

HoICA will expand inter-layer substructure instances from the previous iteration (iteration k) to generate expanded inter-layer substructure instances for iteration $k+1$. Specifically, it employs the adjacency list for each layer to which the composed instance belongs. Through an unconstrained expansion on every vertex ID of the instance using these adjacency lists, HoICA ensures the comprehensive generation of all possible expanded inter-layer instances. For two layers L_1 and L_2,

1. It generates all intra-layer substructures by expanding L_1 substructures using L_1 adjacency list and L_2 substructures using L_2 adjacency list.
2. It generates inter-layer substructures using composed substructures from previous iteration and expanding them using L_1 adjacency list.
3. It generates inter-layer substructures using composed substructures from previous iteration and expanding them using L_2 adjacency list.

The unconstrained nature of this expansion process guarantees that no instances are overlooked. Thus, HoICA accurately generates all instances for the subsequent iteration $k + 1$, given all instances from the previous iteration k.

Conclusion:

Based on the base case and the inductive step, we can confidently assert that HoICA consistently generates substructures of all sizes in the input data for a given iteration.

4.6.2 HoSCA

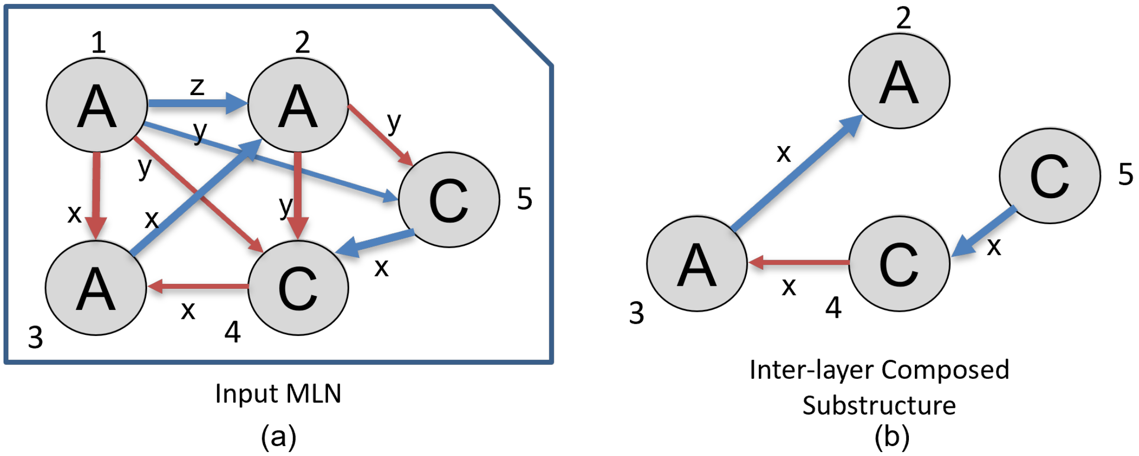


Figure 4.6: Input MLN and Composed Substructure

We shall present a counterexample to illustrate how substructure instances are missed in HoSCA. Figure 4.6(a) depicts an input Multilayer Network (MLN) for the algorithm, comprising of two layers - Layer 1 (red) and Layer 2 (blue). To generate a size 3 substructure, HoSCA requires size 3 substructures from both layers. Figure 4.6(b) shows the inter-layer instance we need to generate. Figures 4.7(a) and 4.7(b) showcase all the substructure instances of size 3 which possess the required edges to generate the instance shown in Figure 4.6(b).

To create the indicated instance, a single edge from Layer 1 and two edges from Layer 2 are needed. However, the two edges required from Layer 2 are not simultaneously present in any of the 3 edge substructure instances present in Layer 2. Instance 4.7(b)[i] contains one edge ⟨x,A,3,A,2⟩ , and instance 4.7(b)[ii] contains the other edge ⟨x,C,5,C,4⟩ , but these two edges are never concurrently present in an instance. Consequently, we would need to compose these two instances from Layer 2

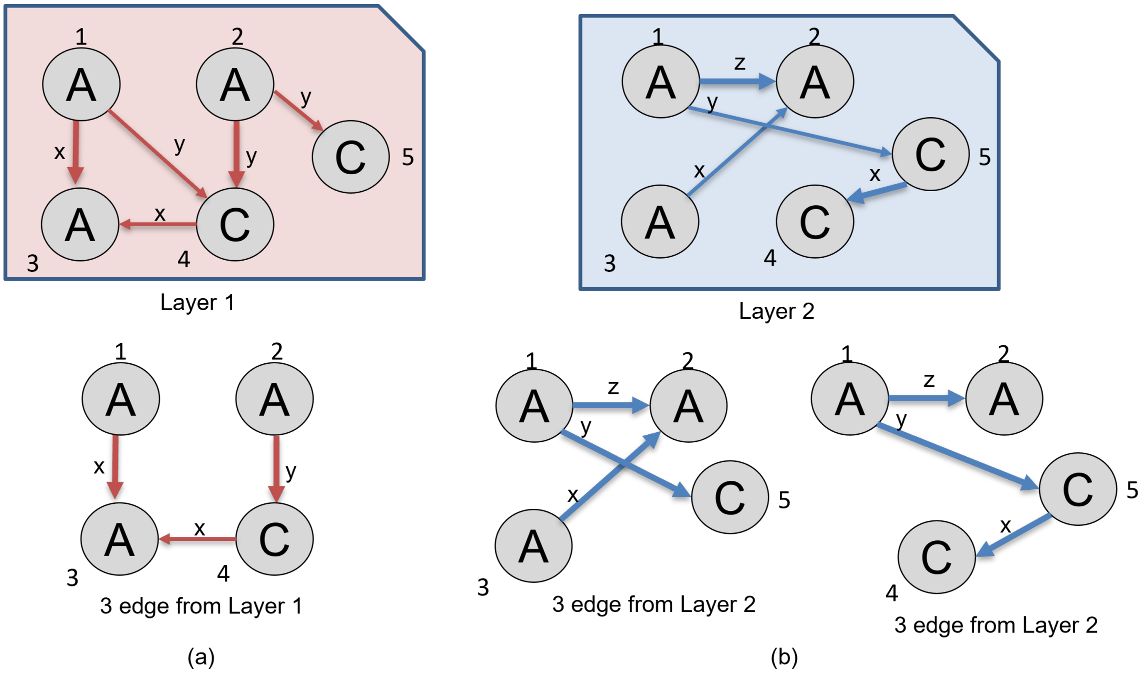


Figure 4.7: Size 3 Instances generated from MLN layers

first, which is not possible in the composition phase. Hence, the composed instance shown is never generated by HoSCA.

Figure 4.8 illustrates the generation of the same missed substructure by HoICA.

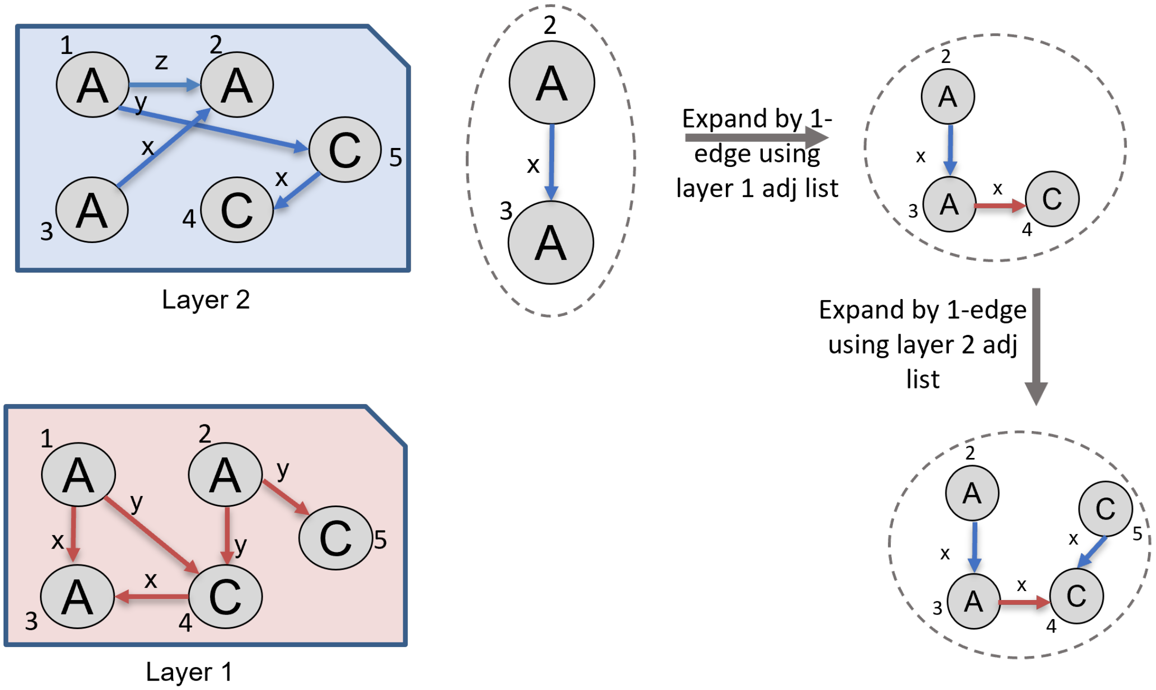

1
A
z
2
A
y
x
C
5
3
A
4
C
x
Layer 2
2
A
x
3
A
Expand by 1-edge using layer 1 adj list
2
A
x
3
A
x
C
4
Expand by 1-edge using layer 2 adj list
1
A
2
A
y
y
x
C
5
y
A
3
x
4
C
Layer 1
2
A
C
x
x
5
3
A
x
C
4

Figure 4.8: Missed Substructure generated by HoICA

In this chapter, we have explored the two approaches to composition and their respective algorithms in HoMLNs. We have shown how the algorithms work and formal proofs were constructed to demonstrate the correctness of the algorithm. In the next chapter, we will focus on the implementation details of these algorithms within the MapReduce framework.

CHAPTER 5

IMPLEMENTATION

In this chapter, we will discuss the implementation of the two decoupling-based substructure discovery algorithms discussed in Chapter 4. We use the MapReduce framework to facilitate distributed and parallel processing. Both the range partitioning and substructure discovery are done inside the MapReduce framework. The driver program which runs MapReduce jobs is implemented in java. The configuration file passed to the driver program specifying the parameters is discussed at the end.

Input Graph Representation: Our input graph is represented as a sequence of unordered edges (or 1-edge substructure). We have discussed the representation in detail in Chapter 3. To recap, each edge is represented by a 5 element tuple ⟨edge label, source vertex id, source vertex label, destination vertex id, destination vertex label ⟩. A substructure instance is a sequence of canonical edge instances. A substructure is a sequence of canonical edge instances represented using a relative vertex id to identify exact subgraphs.

5.1 HoSCA using MapReduce

The single composition algorithm for substructure discovery in HoMLNs (HoSCA) is divided into two distinct phases of the decoupling approach, Analysis phase and Composition phase. Figure 5.1 shows the overall flow of HoSCA using the MapReduce paradigm.

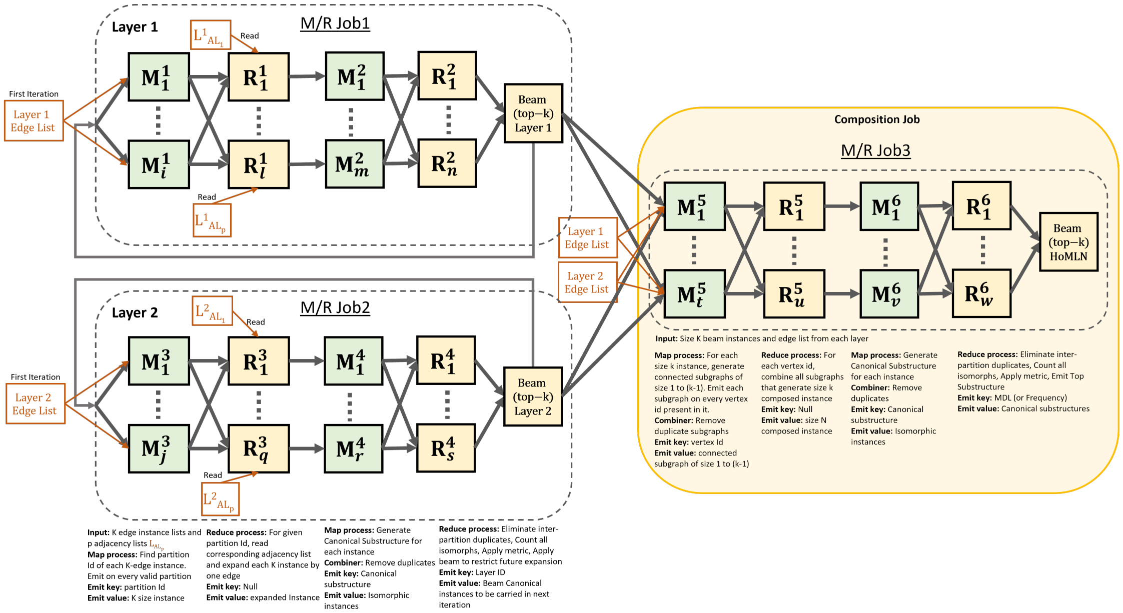

Layer 1
M/R Job1
Read
First Iteration
Layer 1 Edge List
M_1^1
M_i^1
R_1^1
R_l^1
M_1^2
M_m^2
R_1^2
R_n^2
Beam (top−k) Layer 1
$L^1_{AL_1}$
$L^1_{AL_p}$
Layer 2
M/R Job2
First Iteration
Layer 2 Edge List
$L^2_{AL_1}$
$L^2_{AL_p}$
M_1^3
M_j^3
R_1^3
R_q^3
M_1^4
M_r^4
R_1^4
R_s^4
Beam (top−k) Layer 2
Input: K edge instance lists and p adjacency lists L_{AL_p}
Map process: Find partition Id of each K-edge instance. Emit on every valid partition
Emit key: partition Id
Emit value: K size instance
Reduce process: For given partition Id, read corresponding adjacency list and expand each K instance by one edge
Emit key: Null
Emit value: expanded Instance
Map process: Generate Canonical Substructure for each instance
Combiner: Remove duplicates
Emit key: Canonical substructure
Emit value: Isomorphic instances
Reduce process: Eliminate inter-partition duplicates, Count all isomorphs, Apply metric, Apply beam to restrict future expansion
Emit key: Layer ID
Emit value: Beam Canonical instances to be carried in next iteration
Composition Job
M/R Job3
Layer 1 Edge List
Layer 2 Edge List
M_1^5
M_t^5
R_1^5
R_u^5
M_1^6
M_v^6
R_1^6
R_w^6
Beam (top−k) HoMLN
Input: Size K beam instances and edge list from each layer
Map process: For each size k instance, generate connected subgraphs of size 1 to (k-1). Emit each subgraph on every vertex id present in it.
Combiner: Remove duplicate subgraphs
Emit key: vertex Id
Emit value: connected subgraph of size 1 to (k-1)
Reduce process: For each vertex id, combine all subgraphs that generate size k composed instance
Emit key: Null
Emit value: size N composed instance
Map process: Generate Canonical Substructure for each instance
Combiner: Remove duplicates
Emit key: Canonical substructure
Emit value: Isomorphic instances
Reduce process: Eliminate inter-partition duplicates, Count all isomorphs, Apply metric, Emit Top Substructure
Emit key: MDL (or Frequency)
Emit value: Canonical substructures

Figure 5.1: HoSCA Architecture using MapReduce

5.1.1 Analysis Phase

The analysis phase (M/R job 1 and M/R job 2) will operate independently and concurrently on all layers. Within each layer, it will identify all substructures of size k. It also follows the range-based partition strategy discussed in Chapter 3.

Using a range based partitioning strategy, we can identify two distinct grouping criteria that must be followed when handling substructures:

1. Routing Instances Across Processors: Instances across different processors need to be routed to the appropriate adjacency partition for expansion.
2. Grouping Expanded Instances for Ranking: Once expanded, the instances need to be grouped based on their canonical substructure for ranking purposes.

It's crucial to note that these two processes are sequential, such that expansion can only occur before ranking. Consequently, each iteration of the analysis phase needs two consecutive MapReduce pairs. The first MapReduce pair facilitates the routing of substructures to their correct partitions and expansion of substructures in those partitions. The second MapReduce pair eliminates duplicates and identifies the beam substructures with the best rank using metrics such as frequency or Minimum Description Length (MDL). Algorithms 4 and 5 further expand on the mapper and reducer classes in detail. It's important to note that both pairs constitute the analysis phase of the decoupling approach, which is used to find intra-layer substructures. They will be executed in parallel as separate jobs on each layer.

Algorithm 4 Analysis Phase: First MapReduce pair for Partitioning and Expansion

INPUT: Graph Layer edge list (or substructures of size 1), Layer Adjacency List Partitions, Partition Range Information
OUTPUT: Substructures of Size k in Layer

```
Class Mapper
function SETUP
    RangeInfo = load range information of adj partitions
end function
function MAP(key = Null, value = substructure instance)
    PIds = Set for unique partition ids for an instance
    get the source vertex id($sVid$) and destination vertex id($dVid$) from each edge
    for each vertex–id v in value do
        PIds.addPartitionId(v,RangeInfo)
    end for
    for each partition–id p in PIds do
        emit(Key = p, value = substructure instance)
    end for
end function
Class Reducer
function REDUCE(key = partition id, valueList = list of instances in partiton)
    Load partition key from HDFS
    for each k-edge canonical instance ks in values do
        expand ks to (k+1)-edge canonical instance
        emit(Key = partition id, value = expanded substructure instance)
    end for
end function
```

Algorithm 5 Analysis Phase: Second MapReduce pair for Substructure Evaluation

```
INPUT: Substructure Instances, beam size, metric (MDL in our case)
OUTPUT: Beam Substructures
Class Mapper
function MAP(key = Null, value = substructure instances)
    vMap = hashtable to hold unique vertex info for an instance
    get the source vertex id($sVid$) and destination vertex id($dVid$) from each edge
    for each vertex–id v in value do
        vMap.put(v,null)
    end for
    update vMap positions from the value
    new key = generate Canonical substructures from value
    emit(Key = Canonical substructure, value = substructure instance)
end function
Class Reducer
function SETUP
    Beam Map = Null // To store best substructures
end function
function REDUCE(key = Canonical substructure, valueList = list of Isomorphic Instances)
    C = Count(Instances in Values)
    MDL = Calculate mdl of each key using count of Instances
    Update Beam Map with MDL
end function
function CLEANUP
    for each k-edge canonical instance ks in values do
        emit (null, each instance in beamMap)
    end for
end function
```

5.1.2 Composition Phase

After the completion of the analysis phase for each layer, the composition phase is initiated. The output generated by each layer during the analysis phase serves as the input for the composition phase. Similar to the analysis phase, the composition phase comprises two consecutive MapReduce pairs, which are necessary for the same reasons as in the analysis phase. However, unlike the analysis phase, the composition phase is no longer iterative. Both MapReduce pairs are executed exactly once. The second MapReduce pair remains the same as algorithm 5. The first pair is discussed in algorithm 6. The *combine* function called in line 31 is a recursive function, and it was comprehensively explained in Algorithm 2 in Chapter 4.

Algorithm 6 Composition Phase: First MapReduce pair to Generate Inter-Layer Substructures

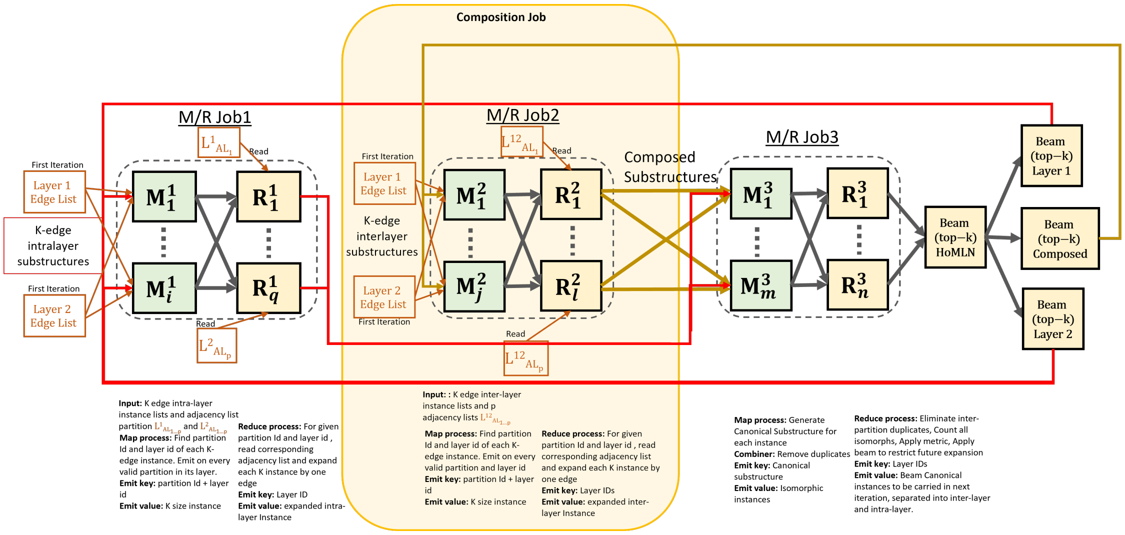
```
INPUT: Intra-layer Substructure Instances of Size k from each Layer
OUTPUT: Composed Inter-layer Substructure Instances of Size k
Class Mapper
 1: function MAP(key = layerID, value = Intra-layer Substructure Instances)
 2:     for each substructure instance ($S_k^N$) do
 3:         Generate connected instances of size 1 to k-1
 4:         Store all in instanceSet
 5:     end for
 6:     for each instance $i$ in instanceSet do
 7:         for each edge $e$ in $i$ do
 8:             get source and destination vertex ids ($sVid$ and $dVid$) in $e$
 9:             VIds.add($sVid$,$dVid$)
10:         end for
11:     end for
12:     for each vertex–id v in VIds do
13:         emit(Key = v, value = layerID + ":" + instance)
14:     end for
15: end function
Class Combiner
16: function COMBINER(key = vertex id, value = instance list)
17:     add each instance in instance list to a set
18:     for each instance $I$ in the set do
19:         emit(Key = vertex id, value = instance)
20:     end for
21: end function
```

```
Class Reducer
function REDUCE(key = vertex id, valueList = list of instances with layerID)
    Initialize a hashmap $layerMap$ ▷ to store layerwise substructures on this $V_{id}$
    for each instance $S_x^N$ in instance list $V_{id}$ do
        Extract $LayerId$ from tagged $instance$
        Add $instance$ to $layerMap$    ▷ key ← $layerID$; value ← $setofinstances$
        if $layerMap$ has at least 2 layers then    ▷ can generate inter-layer subs
            Create an empty list called $Subgraphs$
            Initialize an empty set called $combinations$
            for each  key-value pair in $layerMap$  do
                Add $valueset$ to $Subgraphs$ list
            end for
            Call $combine$ ($Subgraphs$, $accumulator$, $combinations$, $size$) ▷ size = k
        end if
    end for
    for each composed instance $c$ in combinations set do
        emit(Key = Null, value = c)
    end for
end function
```

5.2 HoICA using MapReduce

The iterative composition algorithm for substructure discovery in HoMLNs (HoICA) follows the same two distinct phases of the decoupling approach, Analysis phase and Composition phase. But in this approach, the composition is done as aprt of each iteration. Figure 5.2 shows the overall flow of HoICA using the MapReduce paradigm.

The algorithm now comprises of three MapReduce pairs for each iteration. The first and third pair are the same as first pair and second pair of analysis phase of HoSCA (Algorithms 4 and 5). The composition function and its MapReduce pair undergoes a change. The composition function will now expand inter-layer composed instances by using the adjacency list of each layer to which it belongs. The expansion



Input: K edge intra-layer instance lists and adjacency list partition $L^1_{AL_{1...p}}$ and $L^2_{AL_{1...p}}$
Map process: Find partition Id and layer id of each K-edge instance. Emit on every valid partition in its layer.
Emit key: partition Id + layer id
Emit value: K size instance

Reduce process: For given partition Id and layer id , read corresponding adjacency list and expand each K instance by one edge
Emit key: Layer ID
Emit value: expanded intra-layer Instance

Input: : K edge inter-layer instance lists and p adjacency lists $L^{12}_{AL_{1...p}}$

Map process: Find partition Id and layer id of each K-edge instance. Emit on every valid partition and layer id
Emit key: partition Id + layer id
Emit value: K size instance

Reduce process: For given partition Id and layer id , read corresponding adjacency list and expand each K instance by one edge
Emit key: Layer IDs
Emit value: expanded inter-layer Instance

Map process: Generate Canonical Substructure for each instance
Combiner: Remove duplicates
Emit key: Canonical substructure
Emit value: Isomorphic instances

Reduce process: Eliminate inter-partition duplicates, Count all isomorphs, Apply metric, Apply beam to restrict future expansion
Emit key: Layer IDs
Emit value: Beam Canonical instances to be carried in next iteration, separated into inter-layer and intra-layer.

Figure 5.2: HoICA Architecture using MapReduce

logic and implementation remain similar to the intra-layer substructure expansion (Algorithm 4).

The reducer code remains unchanged. The modification takes place in the mapper, where the input now includes inter-layer substructures with multiple layer IDs. Consequently, the mapper emits the substructure for each layer and the corresponding partition of the adjacency list. During the first iteration, there won't be any inter-layer substructures. Therefore, it is necessary to generate the initial inter-layer substructures using the edge list for each graph layer (size 1 instances). In the subsequent iterations, we will use the composed substructures from previous iterations. Algorithm 7 shows the mapper and reducer classes in detail.

5.3 HoSCA and HoICA Implementation Differences

A key difference between the two implementations is that in HoSCA each layer is processed independently in its own MapReduce Job whereas in HoICA every layer is processed together inside the same MapReduce Job. Each edge is tagged with its layer

Algorithm 7 Composition Phase: Second MapReduce pair to Generate Inter-Layer Substructures for HoICA

INPUT: Inter-Layer Substructures, Graph Layers edge list, Layer Adjacency List Partitions, Partition Range Information
OUTPUT: Inter-Layer Expanded Substructures

Class Mapper

```
function SETUP
    RangeInfo = load range information of adj partitions
end function
function MAP(key = Null, value = inter-layer substructure instance OR Graph Layers edge list)
    LIds = Set for unique layer ids for an instance
    PIds = Set for unique partition ids for an instance
    if First Iteration then
        value = layer-wise edge list
        LIds = layers to which the instance does not belong
    else
        value = inter-layer substructure instance
        LIds = each layerID from the substructure instance
    end if
    get every layerID from the substructure instance
    get the source vertex id($sVid$) and destination vertex id($dVid$) from each edge
    for each vertex–id v in value do
        Find the PartitionId that v belongs to using Range Info
        Add PartitionId to PIds
    end for
    for each partition–id pid in PIds do
        for each layer–id lid in LIds do
            emit(Key = lid+pid, value = substructure instance)
        end for
    end for
end function
```

Class Reducer

```
function REDUCE(key = layer id + partition id, valueList = list of instances in partition)
    Load partition adj list from HDFS
    for each k-edge canonical instance ks in values do
        expand ks to (k+1)-edge canonical instance
        emit(Key = Null, value = expanded substructure instance)
    end for
end function
```

ID to differentiate between inter-layer and intra-layer instances. This tagging wasn't needed in HoSCA as each layer operated independently and in isolation from others. The independence allowed smaller layers to progress through their own iterations without waiting for larger layers. Thus a smaller layer could finish its analysis phase and have all of its intra-layer substructures results ready even before the iterations of larger layers are finished. This imbalance could be remedied by allocating different number of partitions for each layer and different number of mappers.

However in HoICA, the composition function runs inside every iteration. Since composition relies on the results from every layer, it acts like a blocking function. Each layer has to wait on the completion of the composition function before they can progress to the next iteration. Thus each layer has a dependence on the iteration wise results of every other layer. Hence, there is no need to retain the independence of layers in the analysis phase. Even the smaller layers need to wait for the substructures from the larger layers in each iteration. Consequently, metric evaluation stage also becomes a bottleneck, as only after its completion for a given iteration will a given layer know which instances from that layer are present in the beam. To address this, HoICA processes layers together. This decision was made because even if layers were processed independently, smaller layers would still be blocked, leading to their processors sitting idle. By implementing HoICA in this way, we can distribute the load uniformly and achieve better load balancing.

It is important to note that the correctness is not affected by this implementation choice. Load balancing can be done in other ways by using appropriate number of partitions and mapper/reducer nodes. Since we are using the same number of partitions and mapper/reducer numbers for experimentation, the above implementation was chosen.

5.4 Configuration Parameters

We configure various tasks by specifying parameters in a configuration file. The configuration file includes different parameters, such as graph size, beam size, choice of metric, range partitioning information, size of substructures to generate, among others. Below is the list of configuration file options for various parameters.

1. USER: The username for the account running the jobs
2. BASE DIRECTORY: Base HDFS directory
3. BEAM SIZE: Number of substructures to store, ties are handled by keeping all ties
4. SUPPORT: A substructure needs to have number of instances greater than the support value to be considered for metric evaluation. Default value is 1. This helps prevent the beam from being flooded with substructures with frequency of 1.
5. GRAPH INPUT DIRECTORY: Directory in HDFS where the input MLN is loaded.
6. FIXED ADJACENCY DIRECTORY: Directory in HDFS with the partitions of the adjacency lists for each layer.
7. METRIC: Metric to be used for substructure evaluation. Values can be "MDL" or "FREQUENCY".
8. GRAPH VERTEX: Number of vertices in the graph.
9. GRAPH EDGE: Number of Edges in the graph.
10. NO OF LAYERS: Number of layers in the MLN
11. DISTRIBUTION: Distribution of edges among the layers
12. NO OF PARTITIONS: Number of partitions of the adjacency list. Used to calculate the range information as Range = GRAPH VERTEX/NO OF PARTITIONS

13. NO OF MAPPERS: Number of Mappers to use for the MapReduce job.
14. NO OF REDUCERS: Number of Reducers to use for the MapReduce job.
15. MaxSize: Maximum size of substructures to be generated. Used to calculate number of iterations to run. Number of iterations = MaxSize-1
16. MinSize: Minimum size of substructures to be generated.
17. nSubsOutput = Number of best substructures to print. This best substructure list is updated every iteration. For example, if the parameter is set to ”5”, we will write the top 5 substructures with the best MDL/Frequency value into a file during the first iteration. In each subsequent iteration, if any substructure has a better MDL/Frequency value compared to the current substructures, it will overwrite the existing entries. This ensures that, irrespective of the number of iterations executed, the top nSubsOutput file will contain the best substructure offering best compression for the MLN.

Here is an example of the input parameter file used.

```
USER = "axs9120"

BASE_DIRECTORY = "/user/"

BEAM_SIZE = 4

SUPPORT = 1

GRAPH_INPUT_DIRECTORY = "input/"

FIXED_ADJACENCY_DIRECTORY = "fixed_test_adj/"

METRIC = "MDL"      // "FREQUENCY", "MDL"

GRAPH_VERTEX = 800000

GRAPH_EDGE = 1600000

NO_OF_LAYERS = 2

DISTRIBUTION = 50,50

NO_OF_PARTITIONS = 4
```

```
NO_OF_MAPPERS= 4
NO_OF_REDUCERS= 4
MaxSize = 5    // Max size of substructures generated
MinSize = 5    // Min size of substructures generated
nSubs_output = 5   // Best substructures to print
```

5.5 Implementing Job Counters

For evaluating the performance of our MapReduce jobs, we utilized both built-in counters and implemented our own user-defined counters within our Mapper and Reducer programs. Counters in Hadoop MapReduce serve as valuable tools for collecting statistics related to the job, aiding in quality control and enabling analysis of cost and space utilization. The following is a comprehensive list of all the job counters employed. These counters are written to a file after the completion of each MapReduce job.

- Setup time (Mapper and Reducer): Creating a counter for setup time proves beneficial in analyzing the setup cost associated with a Mapper/Reducer. In our MapReduce job, we use the setup to set the values of parameters that would be used in the main map or reduce functions. As we do not do any computation here, these times are always negligible.
- Map time: The mapper invokes the map method for each key/value pair. The map time for any mapper is the time it takes to process all key/value pairs assigned to it.

  The map time in HoICA algorithm consists of routing the substructures (inter-layer and intra-layer) to the correct partitions and the creation of canonical substructures from all instances.

The map time in HoSCA algorithm consists of routing the substructures (intra-layer), creation of connected subgraphs in composition and the creating canonical substructures from all instances.

The map time is computed by finding the maximum among the map times of all individual mappers. This allows us to capture the overall time spent on the mapping phase. In both algorithms, when increasing the number of mappers for the same graph size, we see a reduction in map time as each mapper processes less data.

- Combiner time: We use the combiner to find and eliminate duplicates in the mapper node during the metric evaluation job. This way, we reduce the amount of data that needs to be sent to the reducers. As each mapper gets the instances from one partition, the combiner will eliminate all intra-partition duplicated.
- Reduce time: The Reduce method is invoked for each ¡ key, (list of values) ¿ pair. This counter provides insight into the overall time taken by our Reduce method.

  The reducer time in HoICA involves expansion of substructures(inter-layer and intra-layer) in each layer, counting isomorphic instances, and applying metrics to constrain future expansion.

  In HoSCA, the reducer time involves expansion of substructures(intra-layer), combining substructures across layers to generate inter-layer, counting isomorphic instances, and applying metrics to constrain future expansion.

  Utilizing a counter here is crucial since the majority of both the algorithm's work is done in the reduce phase. This counter allows us to track and analyze the total time spent on these critical reduce phase operations.
- Cleanup time (Mapper and Reducer): The cleanup function is not called in the mapper in any of the jobs. Main cost of cleanup is incurred in reducer

of the metric evalution job. The cleanup function is used to write the beam substructures to the disk in hdfs. The cleanup time is calculated for each reduce task and we store the maximum among all tasks in this counter to determine the overall cleanup time.

- Duplicates Removed by combiner: We have implemented a counter to see how many duplicates were removed by the combiner.
- Duplicates Removed by Reducer: We have implemented a counter to see how many duplicates were removed by the reducer. These are mostly inter-partition duplicates as those could not be removed by the combiner.
- Number of Substructures written: This counter stores the number of substructures that were written out to the disk. This provides valuable insights into the number of substructure instances present in the beam for that iteration. This counter helps in determining the size of the beam substructure set for each iteration.

In the next chapter, we will explore the experimental analysis of our algorithms, evaluating their correctness and scalability across different types of graph sizes and distributions. Additionally, we will examine the impact of parameters such as beam size and the number of partitions on each algorithm's performance.

CHAPTER 6

EXPERIMENTS

In this chapter, we will highlight the results and delve into an in-depth analysis of various experiments conducted on a diverse range of synthetic and real-world graphs. We aim to compare the effects of various MLN characteristics, including layer sizes, layer distributions, the impact of partitions, and the scalability of the two algorithms.

6.1 Experimental Environment:

All experiments were conducted on the Expanse cluster at SDSC (San Diego Supercomputer Center). The Expanse cluster is structured into 13 SDSC Scalable Compute Units (SSCUs), consisting of 728 standard nodes, 54 GPU nodes, and 4 large-memory nodes. We have used the standard nodes for running java with Hadoop MapReduce. Expanse's standard compute nodes are each equipped with two 64-core AMD EPYC 7742 processors and have 256 GB of DDR4 memory. Each compute node has access to a 12 PB parallel file system and 1 TB SSD for local scratch space. Expanse uses the SLURM workload manager for job scheduling. Table 6.1 shows the configuration for each compute node.

6.2 Dataset Generation:

In the context of our substructure discovery algorithm, when we model a dataset as a Homogeneous Multilayer Network (HoMLN), each layer is considered as an independent graph. Consequently, we can transform a large single-attribute graph

Compute Node Component	**Configuration**
Node Count	728
Cores/Node	128 built on 2 processors (64 cores each)
Processor	AMD EPYC 7742
Memory	256 GB DDR4 DRAM
Storage	1TB Intel P4510 NVMe PCIe SSD

Table 6.1: Expanse System Details

into MLNs by partitioning them into layers. This approach provides the flexibility to assess the algorithm's performance across MLNs of varying sizes while maintaining control over other graph characteristics, such as the distribution of edges among layers. Furthermore, for empirical validation, we can incorporate substructures with known frequencies to ensure consistent identification whether the dataset is processed as a single graph or as layers. Hence, our methodology involves generating substantial single graphs with embedded substructures, followed by the creation of homogeneous layers for the Multilayer Network (HoMLN).

6.2.1 Graph Generation:

The initial step involves generating a large single-attribute graph. We use a synthetic graph generator -Subgen. Subgen enables us to create graphs of varying sizes, with multiple embedded substructures of different sizes and frequencies. The input parameters to Subgen include:

1. Graph output filename
2. Number of vertices in the graph
3. Number of edges in the graph
4. Number of unique vertex labels
5. Number of unique edge labels
6. Number of substructures to embed in the graph

7. For each substructure
    - (a) Number of instances
    - (b) Number of vertices
    - (c) For each substructure vertex
        - i. The vertex label. It must be of the form v0, v1 etc..
    - (d) Number of edges
    - (e) For each substructure edge
        - i. The edge label. It must of the form e0, e1, etc.. .
        - ii. The first vertex to which this edge is attached.
        - iii. An integer ranging from 0 to (number of substructure vertices - 1)
        - iv. The second vertex to which this edge is attached.
        - v. An integer ranging from 0 to (number of substructure vertices - 1)

Here is an example input for the graph generator.

```
1MV4ME_embed_5E_100000_1000vl_4000el.g
1000000
4000000
1000
4000
1
100000
6
y1
y2
y3
y4
y5
y6
5
c1
0
1
c2
```

```
1
2
c3
2
3
c4
2
4
c5
1
5
```

Each parameter is specified on a separate line. Here we are creating a graph with 1 million vertices and 4 million edges. There are 1000 unique vertex labels and 4000 unique edge labels. A substructure of size 5 is embedded in the graph with a frequency of 100000. It has 6 vertices (labeled y1, y2, y3, y4, y5, y6) and 5 edges (labeled c1, c2, c3, c4, c5). The generated graph file is called 1MV4ME_embed_5E_100000_1000vl_4000el.g, signifying its size (1MV4ME), embedded substructure size and its frequency (5E and 100000) and number of distinct vertex labels (1000vl) and edge labels (4000el). The graph file generated has an extension of *.g*.

6.2.2 Layer Generation:

The output format of the graph generated by Subgen does not align with the input format used by our algorithm (MR format). Thus, we need to convert the graphs into MR format and also split it into layers. We have written a Python program called LayerGen to achieve both these goals.

The first step is to create layers from the input graph. The script processes each edge from the input (single graph) and distributes it into various files, with the number of files determined by the required number of layers. Multiple layers are created based on the specific needs of the experiment. Each edge is written to only one file, ensuring that the same edge does not appear in multiple layers. We also use different distributions to split the edges between layers. We assign the edges randomly to each layer such that we have the layer distribution we desire.

LayerGen also provides an option to generate connected layers. To establish connections between various disconnected components within a layer, we utilize the Python package NetworkX [43] to identify all the connected components. Subsequently, each connected component is connected to another connected component by introducing an edge between them, distinguished by a unique edge label. This process is iterated as necessary to create a fully connected layer. The newly added edges are then reintegrated into the single aggregated graph, ensuring that our Multilayer Network (MLN) remains consistent with the ground truth. Note that each connecting edge is assigned a distinct label, preventing it from forming a frequent substructure, as any substructure with a connecting edge will always have only one instance.

After the layers are generated, we convert the layers from the subgen format to MR format which is a sequence of one-edge substructures as shown in Chapter 3. We then proceed to generating the adjacency list and its range based partitions according to the number of partitions needed. Generating the adjacency list for each layer is essential since the expansion process in each layer operates independently. We create the adjacency list in memory, although the MapReduce framework can be employed if the graph size is too large.

The parameters passed to our LayerGen program are as follows:

```
[params]
# valid graph path in subgen or non-subgen format
INPUT_GRAPH_PATH = 1000KV4000KE.g
# number of layers to generate
NO_OF_LAYERS = 2
# Distribution ratio for layer generation
# (you can enter multiple distributions as colon separated )
```

```
DISTRIBUTION_RATIOS = 50,50;70,30;90,10
# edgebased or random layer generation
LAYER_GENERATION = random
# if the layers generated should be connected or disconnected or both
CONNECTIVITY_OPTION = disconnected
# number of partitions to generate for each layer
# (you can enter multiple partitions as comma separated)
ALL_PARTITIONS = 8,16,32,64
# Used to calculate range information (number of vertices/number of partitions)
NO_OF_VERTICES = 1000000
# true if input is already in mr format, else false
MR_FORMAT = false
MLN_OPTION = homln  # hemln or homln
```

6.2.3 Dataset Description:

Dataset	Used For	Beam Value	M/R configs (per layer)
Synthetic	Accuracy, Response Time	4,8,12	2M/2R, 4M/4R, 8M/8R
Synthetic Large	Response Time, Scalability	4	4M/4R, 8M/8R, 16M/16R, 32M/32R
Amazon	Response Time, Scalability	4	8M/8R, 16M/16R, 32M/32R, 64M/64R
LiveJournal	Response Time, Scalability	4	8M/8R, 16M/16R, 32M/32R, 64M/64R

Table 6.2: Dataset description

We conducted experiments on various real-world and synthetic datasets to evaluate the effectiveness, correctness, speedup concerning different configurations of mappers and reducers, and scalability of our approach. The datasets used for our experiments are outlined in Table 6.2. We have utilized synthetic graphs of various sizes with multiple embedded substructures with user-defined frequency. This helps

GID	base graph #nodes, #edges	Edge	L1 edges	L2 edges
	(both HoICA and HoSCA)	Distribution		
1	50KV, 100KE	50/50	49797	50203
2		70/30	69862	30138
3		90/10	89899	10101
4	100KV, 500KE	50/50	249953	250047
5		70/30	349722	150278
6		90/10	449691	50309
7	400KV, 1000KE	50/50	500006	499994
8		70/30	700077	299923
9		90/10	899994	1000006
10	1000KV, 4000KE	50/50	1998995	2001005
11		70/30	2799577	1200423
12		90/10	3599822	400178
13	Amazon (0.74MV2.6ME)	50/50	1305431	1305160
14		70/30	1827821	782770
15		90/10	2349423	261168
16	LiveJournal (4.9MV69ME)	50/50	34489570	34504203
17		70/30	48295818	20697955
18		90/10	62096307	6897466

Table 6.3: Dataset Distributions

us to verify our results with the known frequency and substructure. Additionally, real-world datasets from Amazon and LiveJournal were also used for our experiments.

6.3 Empirical Correctness

While formal proofs offer a strong theoretical foundation, empirical validation is essential to ensure that the algorithm's behavior aligns with real-world expectations.

Extensive testing was conducted using diverse datasets and operational scenarios, covering a spectrum of possible inputs and conditions. To verify the correctness of our algorithms, we conducted tests on small synthetic graphs generated by Subgen. Subgen was employed to generate small graphs with predefined embedded substructures ranging from size 50KV/100KE to 1MV/4ME. Subdue [19], HoICA,

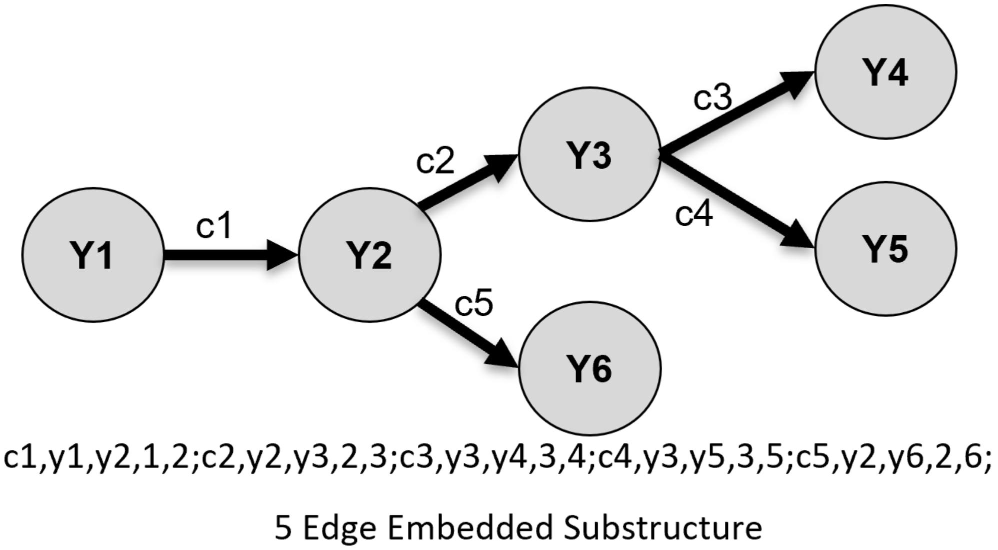

Y1
c1
Y2
c2
Y3
c3
Y4
c4
Y5
c5
Y6
c1,y1,y2,1,2;c2,y2,y3,2,3;c3,y3,y4,3,4;c4,y3,y5,3,5;c5,y2,y6,2,6;
5 Edge Embedded Substructure

Figure 6.1: Embedded substructure

and HoSCA, when run on these graphs, consistently discovered the same substructures. The frequency of substructures found by HoSCA was less than the embedded frequency. In case, of HoICA

Furthermore, the correctness and efficiency of the algorithm were assessed by comparing the results of SUBDUE and our algorithms on the same dataset, converted into layers to generate the MLN. However, SUBDUE, being a main-memory approach, encountered difficulties with increasing graph sizes beyond 100KV/500KE.

To verify correctness on large graphs, we embedded substructures with a user-defined frequency, aiming to find the same substructures. Figure 6.1 illustrates the 5-edge substructure that we have embedded in the synthetic datasets.

6.4 Accuracy

As demonstrated earlier, HoICA consistently exhibits full accuracy, a validation further supported by our experimental analyses. Across various synthetic datasets, HoICA consistently identified all instances of the embedded substructure with exact frequencies. Figure 6.2 shows the algorithm's accuracy on a 50KV100KE graph with an embedded substructure of size 10 and frequency 5000.

In the case of HoSCA, the suspected loss of accuracy is evident, as depicted in Figure 6.2, which illustrates the diminishing accuracy with increasing substructure size. For this experiment we have used a dataset of 50K vertices and 100K edges with an embedded substructure of size 10 with a frequency of 5000. The edges were evenly distributed, with a 50/50 split between the layers. Additionally, the beam parameter was set to a value of 4. Experiments were conducted from iterations 1 to 9, where the result for size k was obtained by composing the output of size k from each layer. In other words, to generate a size 4 substructure, the size 4 beam output from each layer was utilized (which is the output of the *k-1* iteration).

The accuracy significantly drops as the size of the substructure increases. This decline is due to the extended delay before performing composition, resulting in the loss of more information and substructure instances. For instance, if composition is done at size 4 (i.e. using iteration 3 output), instances from 2 iterations are missed. However, if composition is delayed until size 10, instances from 9 iterations are missed. For each iteration that passes without applying composition, we lose increasingly more information. This leads to an increased loss of substructure instances, as composition was not done to regenerate the lost substructures. Consequently, the missed instances accumulate, and by iteration 9, there is minimal information left to regenerate the overlooked substructures.

6.5 Effect of various parameters on Accuracy

In the previous experiment, the beam parameter and layer distribution were fixed at 4 and 50/50, respectively. Now, let's examine how varying these parameters affects our accuracy.

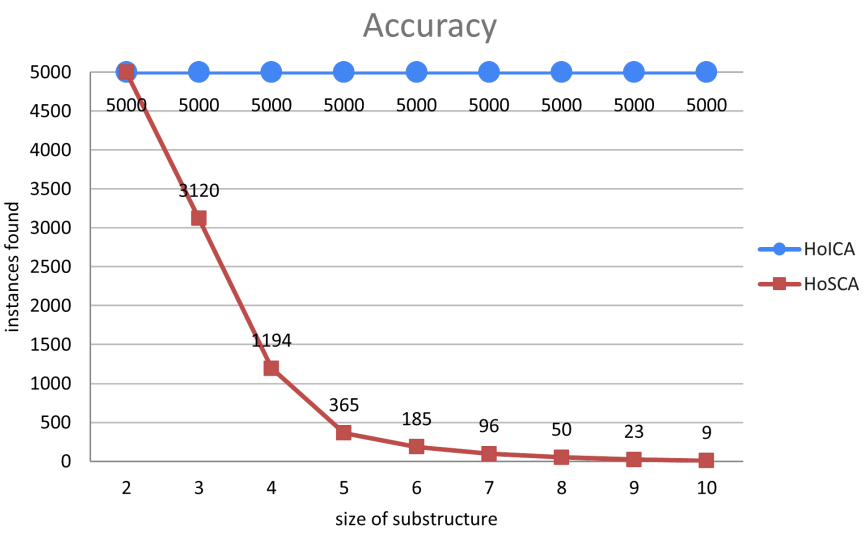


Figure 6.2: Accuracy and size of substructure, 50KV100KE

6.5.1 Effect of Layer Distribution on Accuracy

HoICA consistently attains full accuracy regardless of the layer distribution, displaying resilience to variations in the distribution ratio — be it 50/50, 70/30, or 90/10.

Conversely, for HoSCA, diverse layer distributions yield distinct baseline accuracies. However, these accuracies can be further increased by higher beam sizes, as elaborated in the next section.

Notably, skewed layer distributions appear to yield higher accuracy. This can be attributed to the larger size of a single layer, thus containing a greater number of substructure instances. As these instances are intra-layer and not subjected to loss between iterations, a skewed distribution mitigates the risk of losing instances due to infrequent composition. In contrast, uniform distributions are more likely to split instances across layers, generating higher number of inter-layer instances, contributing to a lower baseline accuracy. Thus, the observed variation in accuracy

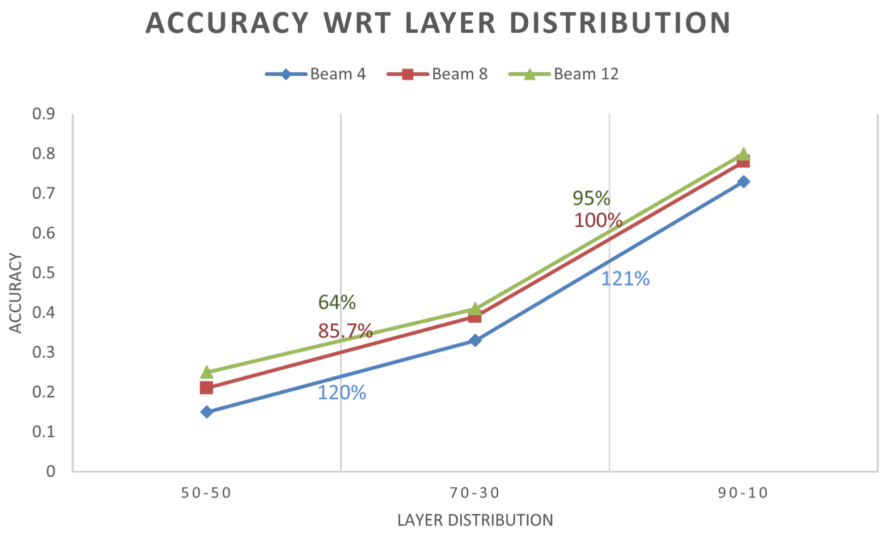


Figure 6.3: Accuracy and Distribution, 50KV100KE

is more indicative of layer characteristics than the algorithm itself. Figure 6.3 shows the increase in accuracy with respect to the layer distribution.

6.5.2 Effect of Beam on Accuracy

HoICA is able to consistently achieve full accuracy at a beam size of 4. Thus, further examination of the beam size parameter in HoICA will not yield any substantial insights due to the already attained optimal accuracy.

However, for HoSCA, which demonstrated a loss of accuracy with increasing substructure size, an exploration of the beam size parameter becomes important. As depicted in Figure 6.4, increasing the beam size contributes to an increase in accuracy. The observed trend suggests that increasing the beam size increases the number substructure instances as input to the composition. Notably, the rise in accuracy from beam size 4 to 8 is more than the subsequent increase from beam size 8 to 12. This pattern implies a limitation on the achievable accuracy associated with increasing the beam size. The diminishing returns in accuracy suggest that there is a constraint on

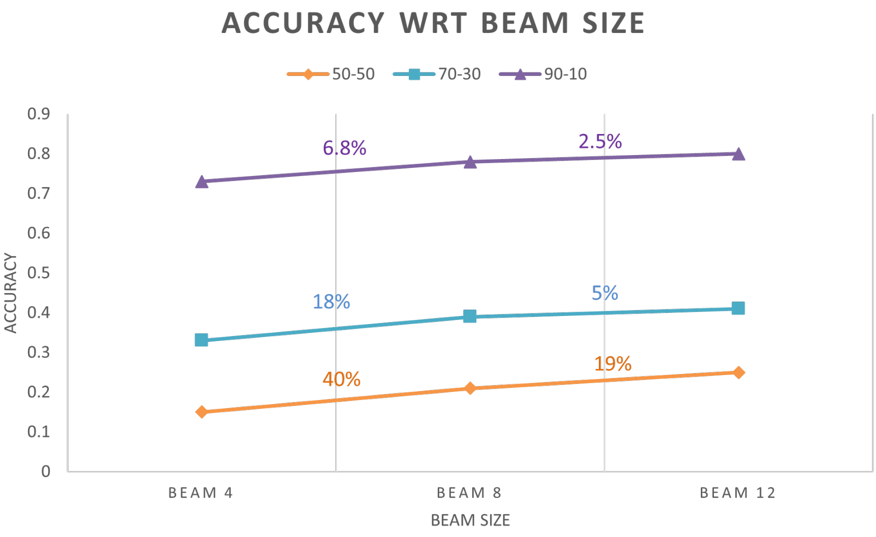


Figure 6.4: Accuracy and Beam Size, 50KV100KE

the algorithm's capacity to regenerate missed instances. This constraint is attributed to the permanent loss of certain substructure instances between the multiple iterations that had passed before the composition was applied. Consequently, the algorithm is unable to recover this missed instances, even with an increased beam size.

6.5.3 Effect of Layer Connectivity on Accuracy

The experiment was replicated with both connected and disconnected layers to explore whether the connectivity of layers affects HoSCA's accuracy. Surprisingly, there was no discernible difference in accuracies between connected and disconnected layers. This lack of discrepancy can be explained by considering the unique connecting edge added, which does not form a frequent substructure. Therefore, it is improbable for this connecting edge and the instance it generates to become part of the beam substructures. This is because they have a frequency of one and thus possess the lowest possible MDL value. In fact, the only scenario in which these instances could be in the beam is when the entire graph is present in the beam. This is undesirable, as having the entire graph in the beam is to be avoided. Hence, introducing a connected layer does not yield a notable impact. Furthermore, as shown in the correctness section of Chapter 4, HoSCA can lose instances even when the layer itself is connected. Moving forward, we will refrain from explicitly generating connected layers to maintain a more generic approach.

6.6 Response Time and Scalability

To gauge our response time, we will employ varying numbers of partitions along with an equivalent number of mappers and reducers to maximize parallelization. The objective is to observe a reduction in the time required to complete our experiments as we increase resources. We aim to understand the speedup we can achieve, whether it follows a linear trend or exhibits diminishing returns over time. Large synthetic and real-world datasets were utilized to verify the speedup and scalability of our approach.

6.6.1 Synthetic Graphs

For the analysis of speedup achieved on synthetic datasets, we explored various dataset sizes, including 50K vertices and 100K edges (50KV100KV), 100K vertices and 500K edges (100KV500KE), 400K vertices and 1 million edges (400KV1ME), and 1 million vertices and 4 million edges (1MV4ME). Regardless of the layer size, each layer was partitioned into the same number of partitions. We experimented with 8, 16, 32, and 64 partitions, maintaining an equivalent number of mappers and reducers. We saw a similar trend in each dataset. Here, we have focused on the largest dataset, 1MV4ME. The results for all the datasets can be seen in Figure 6.11.

For HoICA, Figure 6.5 reveals an average speedup of 48.77% as we increased the number of partitions and the count of mappers/reducers from 8 to 16. Subsequently, a speedup of 43.7% was observed when further increasing them from 16 to 32. Finally, a

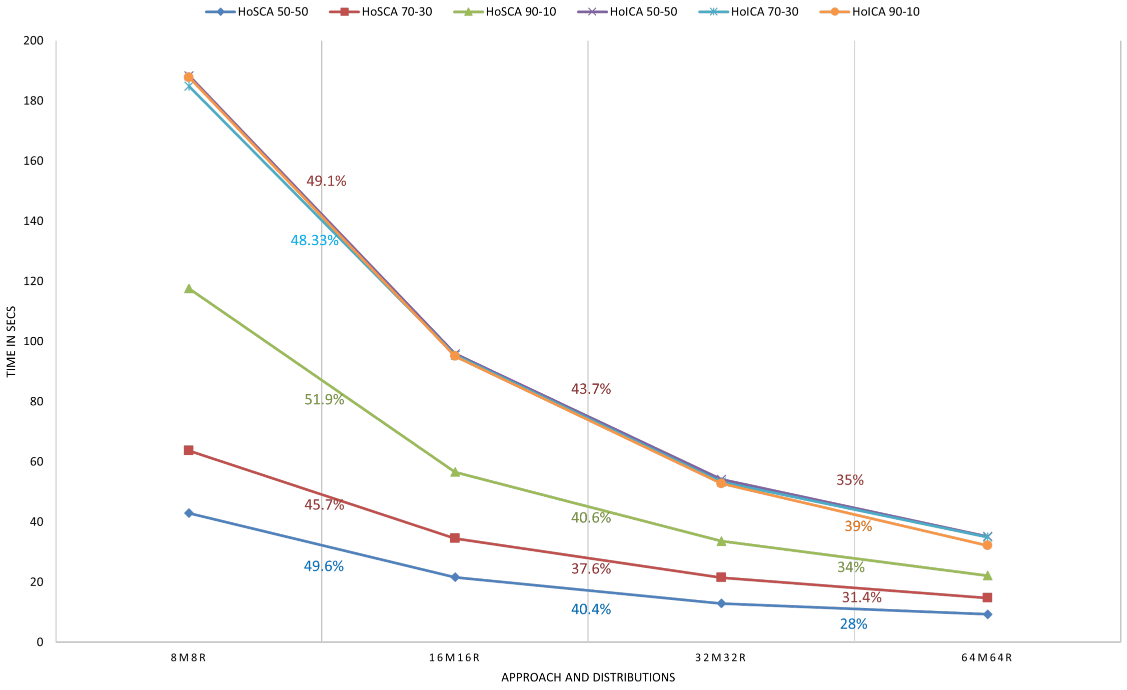


Figure 6.5: Speedup: HoSCA vs HoICA (1MV4ME)

speedup of 36.33% was noticed when further increasing them from 32 to 64. However, the speedup achieved was not linear; doubling the mappers and reducers does not halve the time taken. As the number of partitions increases, the speedup diminishes.

In case of HoSCA, Figure 6.5 illustrates a significant fluctuation in time taken based on layer distribution. There is an average speedup of 49.7% when going from 8 mappers and reducers to 16. Subsequently, a further average speedup of 39.5% is observed when increasing them from 16 to 32. Finally, a speedup of 31.3% is noticed when further increasing them from 32 to 64. Similar to HoICA, the speedup achieved in this scenario is nonlinear and begins to taper off with continued increases in the number of mappers and reducers.

This observed speedup can be attributed to the partitioning of the same dataset into more partitions, processed by an increased number of processors in parallel.

Smaller-sized partitions contribute to a reduced computational load on each processor, leading to the higher speedup. However, the diminishing speedup suggests that the increase in the number of partitions does not endlessly improve performance. Higher number of partitions allows more instances to belong to multiple partitions, increases the potential for inter-partition duplication. This leads to redundant work in different processors. Moreover, excessively small input splits can negatively impact Hadoop performance, introducing overhead related to task setup, communication, and coordination. Conversely, overly large input splits can result in an uneven distribution of processing workload among nodes, leading to inefficient resource utilization. Hence, determining the optimal number of input splits is crucial for optimizing Hadoop job performance. If the objective is to maximize efficiency, employing 16 Mappers/Reducers appears to be the most effective choice as it provides the best speedup. However, in case there is surplus processing power available, 64 Mappers/Reducers still yields a respectable speedup of around 30% while taking the least amount of time.

Effect of Layer Distribution:

As evident from Figure 6.5, layer distribution does not significantly impact the processing time in the case of HoICA. This behavior stems from the algorithm's implementation, where all layers are processed within the same MapReduce job, enabling effective load balancing and ensuring uniform load distribution among processors.

However, in HoSCA, each layer is processed in a separate MapReduce job. In case of skewed layer distributions, where a smaller layer finishes processing earlier than the larger one, its processors remaining idle while waiting for the larger job to complete. Consequently, skewed distributions lead to increased processing time compared to uniform distributions, where load distribution among processors is more balanced. To mitigate this, alternative partitioning strategies based on layer size could be explored for HoSCA.

6.6.2 Real World Graphs

We have employed real-world graphs to demonstrate scalability and speedup. Two real-world datasets, namely Amazon and LiveJournal, were utilized, featuring sizes of 0.74MV2.6ME and 4.9MV69ME, respectively. These datasets were partitioned into 8, 16, 32, and 64 partitions, ensuring equivalent number of mappers and reducers for each partition configuration.

Amazon(0.74MV2.6ME): The notable change is the minimal difference between the two approaches for this dataset. This is unexpected as we expect HoSCA to be faster compared to HoICA due to less number of compositions applied. The consistent trend in speedup as the number of mappers/reducers increases remains unchanged for both approaches. Figure 6.6 reveals the time taken and the average speedup achieve for different number of partitions and the count of mappers/reducers from 8 to 16 to 32 to 64.

The reason for the similarity in execution times between both approaches can be explained knowing the fact that the most frequent substructures in the dataset were of size 2. Given that real-world datasets may or may not contain frequent substructures, and we have not embedded larger substructures in these datasets, both approaches exhibit similar performance in capturing substructures of size 2. As composition was minimal in this dataset, the disparity between the two approaches diminishes. As shown in Figure 6.7, composition job for HoICA does much less work after iteration 1. Thus in this case, HoICA effectively acts like a single composition, eroding HoSCA advantage. Interestingly, HoSCA has slightly higher timings than HoICA, which can be attributed to skewed distributions having uneven workload in the HoSCA implementation.

We can contrast this with the composition job in the synthetic dataset (1MV4ME). Figure 6.8 shows the composition job for the synthetic dataset. In this case, as there

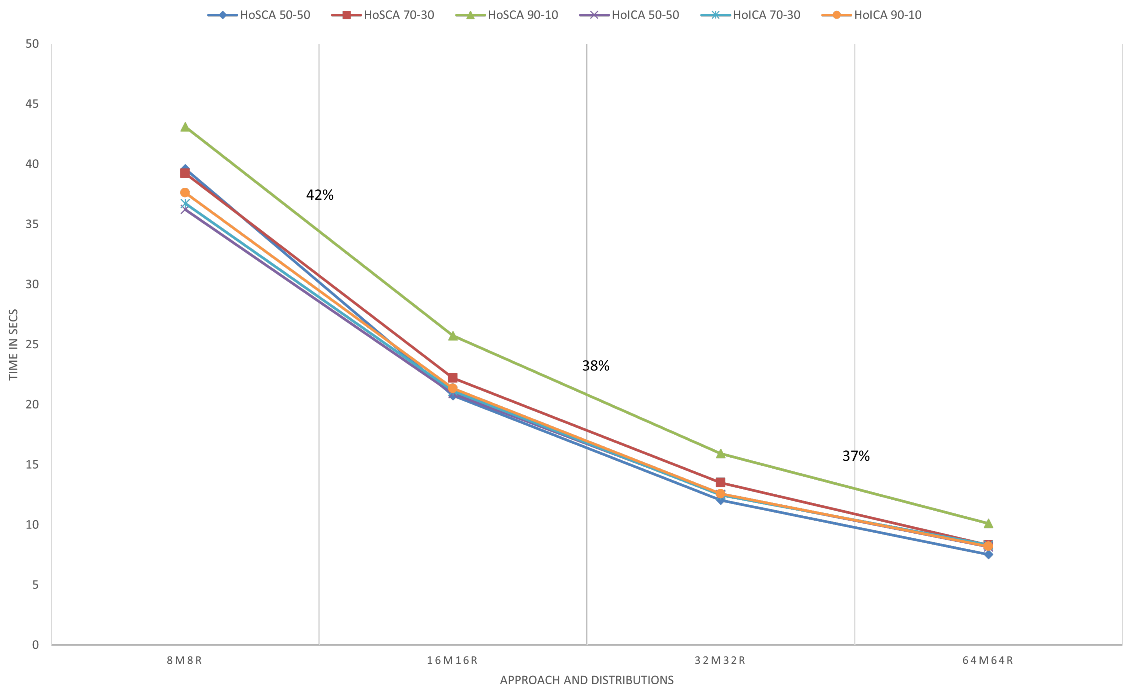


Figure 6.6: Speedup: Amazon Dataset

were still substructures to be found of size 3 and above, the time taken in the composition job keeps increasing as the number of substructures found and their size increases. Larger substructures have more vertex ids that they can be expanded on, requiring more computational work. This leads to the steady increase in the reduce time from iteration 2 onwards as expansion is done in the reducer. The map time only slightly increases, as the mapper solely handles the routing of substructures based on vertex ids. This implies that the map time is solely dependent on the number of substructure instances received by the mapper, irrespective of their size. It is essential to note that the dip from iteration 1 to iteration two is due to the beam being applied after the first iteration. In other words, all the edges in the graph are processed in the first iteration, but only the top beam substructures are processed starting from

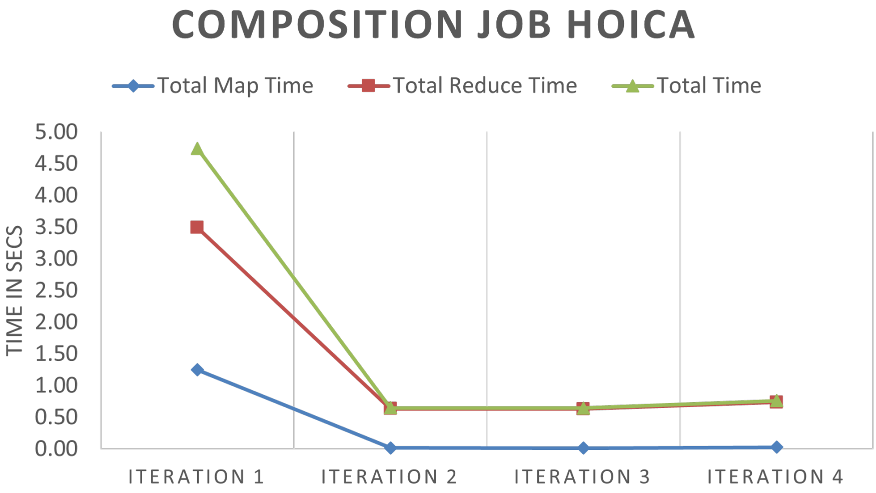


Figure 6.7: Composition Job for Amazon Dataset: HoICA

the second iteration. This greatly reduces the number of substructures considered for expansion. Thus, a significant dip is observed between iteration 1 and iteration 2.

LiveJournal(4.9MV69ME):

The LiveJournal dataset exhibits similar trends to those seen in the previous Amazon dataset, as depicted in Figure 6.9. This dataset is notably larger at 4.9 Million Vertices and 69 Million Edges compared to the previous datasets. Both approaches were able to complete the substructure discovery process. The average speedup while varying number of partitions and mappers/reducers is greater than that observed in the Amazon dataset.

Specifically, a consistent decrease of nearly 50% is observed in transitions from 8 to 16 and 16 to 32 partitions. However, from 32 to 64 partitions, the speedup diminishes to 39%, but it remains higher compared to other datasets. This can be attributed to the larger size of the graph, which helps alleviate the added overhead from adding more tasks/processors. Having a large overhead due to a higher number of tasks for a small dataset would diminish the speedup, but for a sufficiently large

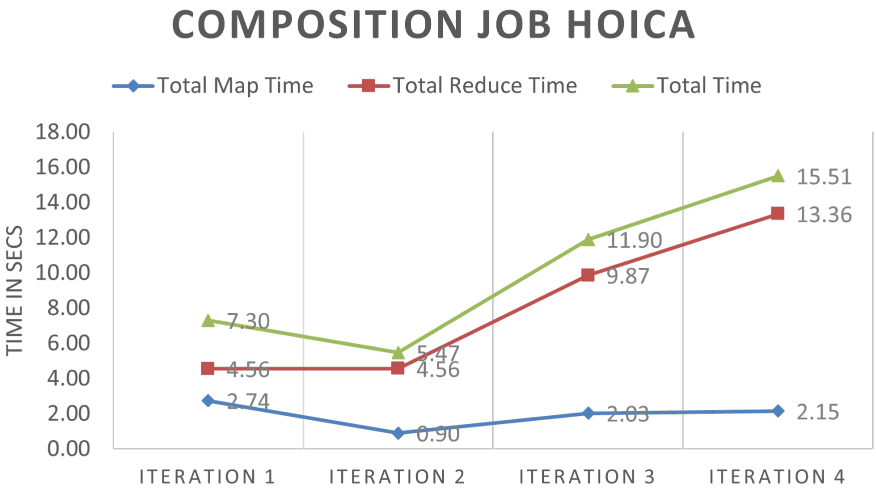


Figure 6.8: Composition Job for 1MV4ME Dataset: HoICA

dataset like LiveJournal, doubling the processors results in nearly double speedup (approximately 50% reduction in time).

Once again, both approaches exhibit similar execution times, a phenomenon also observed in the Amazon dataset. This similarity can be attributed to the fact that, as in the previous dataset, the most frequent substructures found in the LiveJournal dataset are still of size 2, minimizing the necessity for composition.

6.7 Response Time for All Experiments Performed

We aim to analyze the variations in response time across different datasets characterized by diverse sizes and features. For a comprehensive overview, we have consolidated the results from various experiments into two graphs, one for real world datasets and other for synthetic datasets.

In Figure 6.10 and Figure 6.11 show all the experiments ran on real world and synthetic datasets respectively. Figure 6.11 also has the timing for Subdue which was our ground truth.

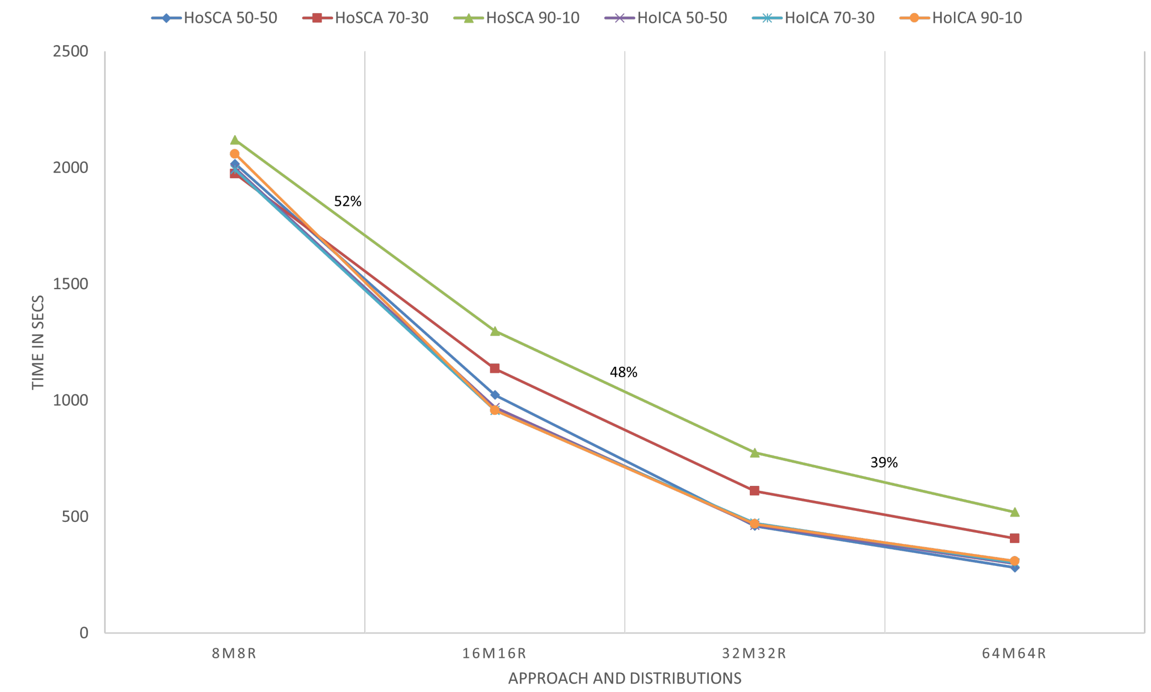


Figure 6.9: Speedup: LiveJournal Dataset

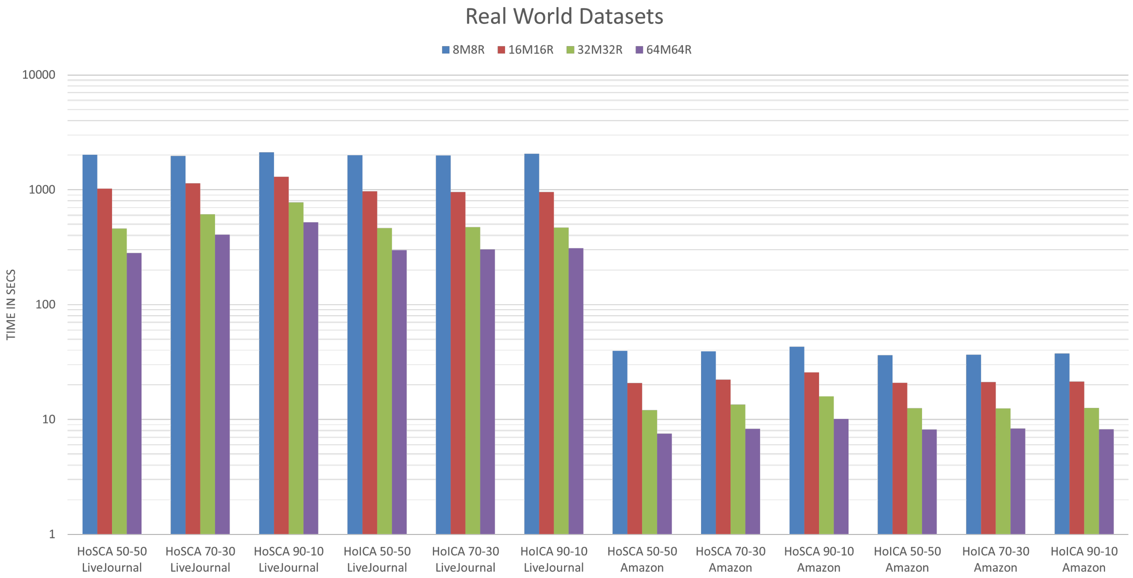


Figure 6.10: Speedup: Real World Datasets

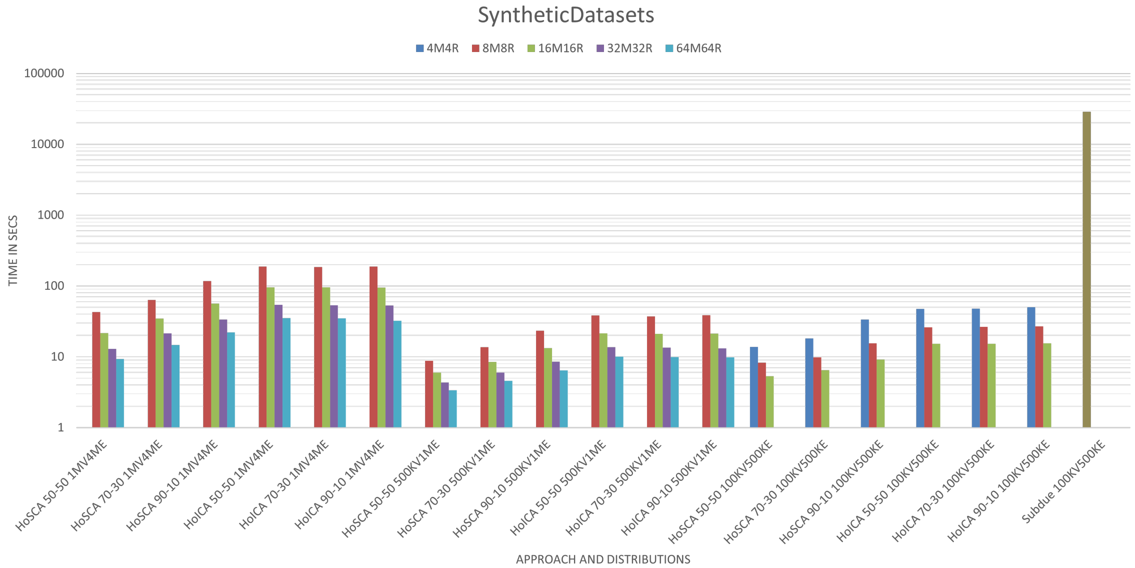


Figure 6.11: Speedup: Synthetic Datasets

In conclusion, we have presented thorough experiments conducted on a variety of datasets, highlighting the distinctions between the two proposed approaches. The graphs utilized in these experiments exhibited diverse characteristics. Through these experiments, we successfully validated both approaches across a range of graph sizes and a wide spectrum of graph characteristics. The extensive experiments have provided robust verification of the validity and efficacy of our approach.

CHAPTER 7

CONCLUSIONS AND FUTURE WORK

This thesis introduces two scalable approaches for substructure discovery within a Homogeneous Multilayer Network (HoMLN), employing a decoupling-based strategy. Two distinct approaches to composition were proposed and implemented using the MapReduce paradigm, emphasizing correctness and effectiveness. The algorithm encompasses critical graph mining components, including subgraph generation, inter-layer substructure combination, duplicate elimination, and isomorphic substructure counting. Through extensive experiments, we validated the efficiency of our MapReduce-based approach, demonstrating its capacity to scale effectively to large graphs with arbitrarily large layers. Experiments were conducted at a very large scale, providing a thorough verification of our findings.

The domain of partitioned graph mining presents intriguing avenues for future research. Further exploration of partitioning techniques for MLN layers, including arbitrary partitioning, can help address challenges associated with uneven partitions resulting from range-based partitioning. Additionally, future work may involve extending our approach to other distributed frameworks, such as Spark, to validate and extend our findings.

REFERENCES

[1] Neo4j. [Online]. Available: https://neo4j.com

[2] S. Wasserman and K. Faust, "Social network analysis: Methods and applications," 1994.

[3] A. Santra, "Analysis of Complex Data Sets Using Multilayer Networks: A Decoupling-based Framework," Ph.D. dissertation, The University of Texas at Arlington, July 2020. [Online]. Available: https://itlab.uta.edu/students/alumni/PhD/Abhishek_Santra/ASantra_PhD2020.pdf

[4] A. Santra and S. Bhowmick, "Holistic analysis of multi-source, multi-feature data: Modeling and computation challenges," in *Big Data Analytics - Fifth International Conference, BDA 2017*, 2017.

[5] M. Zitnik and J. Leskovec, "Predicting multicellular function through multi-layer tissue networks," *Bioinformatics*, vol. 33, no. 14, pp. i190–i198, 07 2017. [Online]. Available: https://doi.org/10.1093/bioinformatics/btx252

[6] Facebook users statistics. [Online]. Available: https://investor.fb.com/financials/

[7] Linkedin users statistics. [Online]. Available: https://news.linkedin.com/2023/july/linkedin-business-highlights-from-microsoft-s-fy23-q4-earnings

[8] Imdb statistics. [Online]. Available: https://www.imdb.com/pressroom/stats/

[9] A. Santra, S. Bhowmick, and S. Chakravarthy, "Efficient community detection in boolean composed multiplex networks," *University of Texas at Arlington*, vol. June, 2019. [Online]. Available: http://itlab.uta.edu/research/current/Multi\%20Source\%20Data\%20Analysis/ArXiv2019-HoMLN-Final.pdf

[10] A. Santra, K. S. Komar, S. Bhowmick, and S. Chakravarthy, “From base data to knowledge discovery - *A life cycle approach* - using multilayer networks,” *Data Knowl. Eng.*, vol. 141, p. 102058, 2022. [Online]. Available: https://doi.org/10.1016/j.datak.2022.102058

[11] A. Santra, S. Bhowmick, and S. Chakravarthy, “Efficient community re-creation in multilayer networks using boolean operations,” in *International Conference on Computational Science, ICCS 2017, 12-14 June 2017, Zurich, Switzerland*, ser. Procedia Computer Science, P. Koumoutsakos, M. Lees, V. V. Krzhizhanovskaya, J. J. Dongarra, and P. M. A. Sloot, Eds., vol. 108. Elsevier, 2017, pp. 58–67. [Online]. Available: https://doi.org/10.1016/j.procs.2017.05.246

[12] ——, “Hubify: Efficient estimation of central entities across multiplex layer compositions,” in *2017 IEEE International Conference on Data Mining Workshops, ICDM Workshops 2017, New Orleans, LA, USA, November 18-21, 2017*, R. Gottumukkala, X. Ning, G. Dong, V. Raghavan, S. Aluru, G. Karypis, L. Miele, and X. Wu, Eds. IEEE Computer Society, 2017, pp. 142–149. [Online]. Available: https://doi.org/10.1109/ICDMW.2017.24

[13] K. Mukunda, “Decoupling-based approach to centrality detection in heterogeneous multilayer networks,” Master’s thesis, The University of Texas at Arlington, May. 2021. [Online]. Available: https://itlab.uta.edu/students/alumni/MS/Kiran_Mukunda/KMukunda_MS2021.pdf

[14] H. R. Pavel, A. Santra, and S. Chakravarthy, “Degree centrality algorithms for homogeneous multilayer networks,” in *Proceedings of the 14th International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management, IC3K 2022, Volume 1: KDIR, Valletta, Malta, October 24-26, 2022*, F. Coenen, A. L. N. Fred, and J. Filipe, Eds. SCITEPRESS, 2022, pp. 51–62. [Online]. Available: https://doi.org/10.5220/0011528900003335

[15] A. Santra, K. S. Komar, S. Bhowmick, and S. Chakravarthy, “A new community definition for multilayer networks and A novel approach for its efficient computation,” *CoRR*, vol. abs/2004.09625, 2020. [Online]. Available: https://arxiv.org/abs/2004.09625

[16] A. Rai, “Mln-subdue: Decoupling approach-based substructure discovery in multilayer networks (mlns),” Master’s thesis, The University of Texas at Arlington, May. 2020. [Online]. Available: http://itlab.uta.edu/students/alumni/MS/Anish_Rai/ARai_MS2020.pdf

[17] A. Santra, K. S. Komar, S. Bhowmick, and S. Chakravarthy, “A new community definition for multilayer networks and A novel approach for its efficient computation,” *CoRR*, vol. abs/2004.09625, 2020. [Online]. Available: https://arxiv.org/abs/2004.09625

[18] S. Das, “Divide and Conquer Approach to Scalable Substructure Discovery: Partitioning Schemes, Algorithms, Optimization and Performance Analysis Using Map/Reduce Paradigm,” Ph.D. dissertation, The University of Texas at Arlington, May 2017. [Online]. Available: http://itlab.uta.edu/students/alumni/PhD/Soumyava_Das/SDas_PhD2017.pdf

[19] N. Ketkar, L. Holder, and D. Cook, “Subdue: Compression-based frequent pattern discovery in graph data,” *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 08 2005.

[20] S. Padmanabhan and S. Chakravarthy, “Hdb-subdue: A scalable approach to graph mining,” in *DaWaK*, 2009, pp. 325–338.

[21] L. R. Quinlan and R. L. Rivest, “Inferring decision trees using the minimum description length principle,” 1987.

[22] R. Agrawal and R. Srikant, “Fast Algorithms for Mining Association Rules,” in *Very Large Data Bases*, 1994, pp. 487–499.

[23] M. Deshpande, M. Kuramochi, and G. Karypis, "Frequent Sub-Structure-Based Approaches for Classifying Chemical Compounds," in *IEEE International Conference on Data Mining*, 2003, pp. 35–42.

[24] J. Pei, J. Han, B. Mortazavi-Asl, H. Pinto, Q. Chen, U. Dayal, and M.-C. Hsu, "PrefixSpan,: mining sequential patterns efficiently by prefix-projected pattern growth," in *ICDE*, 2001, pp. 215–224.

[25] H. Bunke and K. Shearer, "A graph distance metric based on the maximal common subgraph," *Pattern Recognition Letters*, vol. 19, pp. 255–259, 1998.

[26] S. Alexaki, V. Christophides, G. Karvounarakis, and D. Plexousakis, "On Storing Voluminous RDF Descriptions: The Case of Web Portal Catalogs," in *International Workshop on the Web and Databases*, 2001, pp. 43–48.

[27] X. Yan, P. S. Yu, and J. Han, "Graph indexing: a frequent structure-based approach," in *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, 2004, pp. 335–346.

[28] S. Chakravarthy and S. Pradhan, "DB-FSG: An SQL-Based Approach for Frequent Subgraph Mining," in *DEXA*, 2008, pp. 684–692.

[29] S. Chakravarthy, R. Beera, and R. Balachandran, "DB-Subdue: Database Approach to Graph Mining," in *PAKDD*, 2004, pp. 341–350.

[30] J. Huang, W. Qin, X. Wang, and W. Chen, "Survey of external memory large-scale graph processing on a multi-core system," *The Journal of Supercomputing*, vol. 76, no. 1, pp. 549–579, 2020.

[31] Arangodb. [Online]. Available: https://www.arangodb.com/

[32] Dgraph. [Online]. Available: https://dgraph.io/

[33] Y. Liu, X. Jiang, H. Chen, J. Ma, and X. Zhang, "MapReduce-Based Pattern Finding Algorithm Applied in Motif Detection for Prescription Compatibility Network," in *Advanced Parallel Programming Technologies*, 2009, pp. 341–355.

[34] F. N. Afrati, D. Fotakis, and J. D. Ullman, “Enumerating subgraph instances using map-reduce,” Stanford University,” Technical Report, December 2011. [Online]. Available: http://ilpubs.stanford.edu:8090/1020/

[35] S. Suri and S. Vassilvitskii, “Counting triangles and the curse of the last reducer,” in *World Wide Web Conference Series*, 2011, pp. 607–614.

[36] S. Yang, X. Yan, B. Zong, and A. Khan, “Towards effective partition management for large graphs,” in *SIGMOD Conference*, 2012, pp. 517–528.

[37] G. Karypis and V. Kumar, “Multilevel k-way partitioning scheme for irregular graphs,” in *JPDC*, vol. 48, no. 1. Elsevier, 1998, pp. 96–129.

[38] H. Singh and R. Sharma, “Role of adjacency matrix & adjacency list in graph theory,” *International Journal of Computers & Technology*, vol. 3, no. 1, pp. 179–183, 2012.

[39] J. Leskovec, A. Rajaraman, and J. D. Ullman, *Mining of massive data sets.* Cambridge university press, 2020.

[40] R. B. Rao and S. C. Lu, “Learning engineering models with the minimum description length principle.” in *AAAI*, 1992, pp. 717–722.

[41] B. Bringmann and S. Nijssen, “What is frequent in a single graph?” in *Pacific-Asia Conference on Knowledge Discovery and Data Mining.* Springer, 2008, pp. 858–863.

[42] M. Elseidy, E. Abdelhamid, S. Skiadopoulos, and P. Kalnis, “Grami: Frequent subgraph and pattern mining in a single large graph,” *Proceedings of the VLDB Endowment*, vol. 7, no. 7, pp. 517–528, 2014.

[43] A. Hagberg, P. J. Swart, and D. A. Schult, “Exploring network structure, dynamics, and function using networkx.” [Online]. Available: https://www.osti.gov/biblio/960616

BIOGRAPHICAL STATEMENT

Arshdeep Singh was born in Chandigarh, India. He received his Bachelors Degree in Electronics and Communication Engineering from Thapar University, India in May, 2018. He worked as a software developer at Cadd Primer, Chandigarh, India from July, 2018 till July, 2021. He started his Masters studies in Computer Science at The University of Texas, Arlington in August, 2021 and received his Masters degree in Dec 2023. His research interests include graph mining, big data engineering and machine learning.