HeMLN-SD: Substructure Discovery in Heterogeneous Multilayer Networks

by

KIRAN BOLAJ

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

December 2023

ACKNOWLEDGEMENTS

I extend my heartfelt appreciation and profound gratitude to Dr. Sharma Chakravarthy for not only affording me the invaluable opportunity to engage in this research project but also for consistently providing unwavering motivation, support, and guidance. His continual assistance and wise counsel were indispensable, playing a pivotal role in the completion of this work. Additionally, my sincere thanks go to Dr. Ashraf Aboulnaga and Dr. Abhishek Santra for generously contributing their time and expertise as members of my committee.

My sincere acknowledgments are also owed to Arshdeep Singh, Anamitra Roy, and Umme Hafsa Billah for their enduring guidance, unwavering support, and uplifting words throughout this academic endeavor. Furthermore, I express my gratitude to my family and friends for their steadfast support and encouragement, which have been a source of strength throughout my academic journey.

December 4, 2023

ABSTRACT

HeMLN-SD: Substructure Discovery in Heterogeneous Multilayer Networks

Kiran Bolaj, M.S.

The University of Texas at Arlington, December 2023

Supervising Professor: Dr. Sharma Chakravarthy

Graph mining analyzes the real-world graphs for finding core substructures in chemical compounds (e.g., Benzene), identify the structure that occurs frequently in a given graph or forest. These identified structures are important as they reveal an inherent feature or property in the given graph or forest. Substructures represent interesting and repeating patterns found within an application, offering insights into hidden regularities. Therefore, the process of finding these interesting and frequent patterns in an unsupervised manner is known as substructure discovery. SUBDUE was the first main-memory algorithm developed for substructure discovery. Since then, for scalability, the algorithm has been extended to Database approaches and more recently to Map/Reduce framework to exploit distributed processing.

Graph sizes have been increasing due to the advent of Internet and Social networks applications. To model complex data sets (sets with multiple types of entities and relationships), Multilayer networks, or MLNs, have been Proposed. MLNs have also been shown to be superior as compared to simple and attribute graphs for modeling complex data. MLNs are classified into three different types: Homogeneous (HoMLNs), Heterogeneous (HeMLNs), and Hybrid (HyMLNs). Homogeneous MLNS

have same type of entities in each layer but differ in connectivity as it represents a unique relationship. Heterogeneous MLNs have different types of entities in each layer and also differ in connectivity and have explicit interlayer edges that connects nodes between the layers. Hybrid MLNs are the combination of Homogeneous and Heterogeneous MLNs. Algorithms have been developed for substructure discovery in HoMLNs, which differ from earlier approaches of aggregating MLN layers to a single graph and using the single graph algorithms.

Drawing inspiration from the earlier work of finding substructures in Homogeneous Multilayer Networks, this thesis focuses on substructure discovery in Heterogeneous Multilayer Networks without aggregating the layers into a single graph. We use the decoupling-based approach by processing each layer separately/independently which can be done in parallel, if necessary, to find the substructures and then composing independently generated layer substructures after each iteration with interlayer edges to get the substructures across HeMLN for that iteration. The algorithm is implemented using the Map/Reduce paradigm. The main focus of this dissertation is the correctness of the algorithm by verifying with ground truth for which we use the SUBDUE algorithm. Another focus is the scalability of the approach to handle arbitrary MLN sizes. For this, we perform extensive experimental analysis on large synthetic data sets (generated by Subgen) and real-world datasets to analyze the speedup over diverse graph characteristics.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

LIST OF TABLES

CHAPTER 1

INTRODUCTION

Large applications data that can be modeled using graphs can be found everywhere. Examples include the analysis of the World Wide Web's structure [1–3], social-media data, representation of bio-informatics data using de Bruijn graphs [4], atoms and covalent relationships in chemistry [5], etc. Many open-source frameworks have been developed [6–8] as a result of the release of proprietary graph processing tools by academic research centers working with industry leaders like Facebook, Microsoft, and Google. They must manage enormous graphs, such as the Facebook graph, which has hundreds of billions of edges and billions of vertices [9].

Graphs are better than other representations (e.g., relational) for data that has relationships among objects in the data. A variety of applications such as chemical, bioinformatics, computer vision, social networks, text retrieval and web analysis come under this category. Graphs are also easy to understand. Graph models use vertices and edges where each vertex of the graph will correspond to an entity and each edge will correspond to a relationship between two entities. These models are used to find frequent patterns which is the same as discovering subgraphs which occurs frequently or compress the graph better over the entire graph or forest. The graph models are of three kinds as described below -

1. **Simple graphs**: Simple graphs (also called single graphs or monoplexes) express entities as nodes and relationships as edges. Node numbers are usually unique, although labels (for both nodes and edges) are not. This graph model is simple to grasp and analyze because it doesn't allow loops or multiple edges

between nodes. Figure 1.1(a) shows a simple graph with only one type of nodes and edges without any labels. This kind of graph can best represent one entity type and one relationship type. If more entity and relationship types are present, they need to be combined as a single node/edge for representation. Even with this limitation, this model has been extensively used for modeling. In addition, most of the algorithms have been developed for this type of graph representation. However, we cannot express several entity and relationship types with multiple edges and labels in this model. To overcome these limitations, an attributed graph is used.

2. **Attributed graphs**: Attributed graphs, also known as Multigraphs, can be represented as multiple entity types and relationships by allowing multiple edges between nodes and loops (identical start and end nodes). Figure 1.1(b) shows an attributed graph with multiple node and edge types, with different colors representing distinct types. Nodes and edges are allowed to multiple labels as well. This model is more expressive than a simple graph, but is also difficult to analyze. The added complexity makes graph analysis and interpretation more challenging. Algorithms developed for simple graphs cannot be directly used on multigraphs. This leads to the introduction of multilayer networks.
3. **Multilayer Networks (MLNs)**: MLNs are simple graph networks with multiple layers, each capturing a unique relationship between entities and their relationship. Figure 1.1(c) demonstrates a multilayer network for the attributed graph in Figure 1.1. each layer contain nodes of a particular entity type and relationship as edges. As an example, a MLN splits a multigraph into discrete layers, based on entity types and their relationships, each layer being a simple graph. This method improves data comprehension. Although new algorithms

need to be developed for MLNs, it may be possible to use existing algorithms for each layer when the decoupling-approach is used for new algorithms.

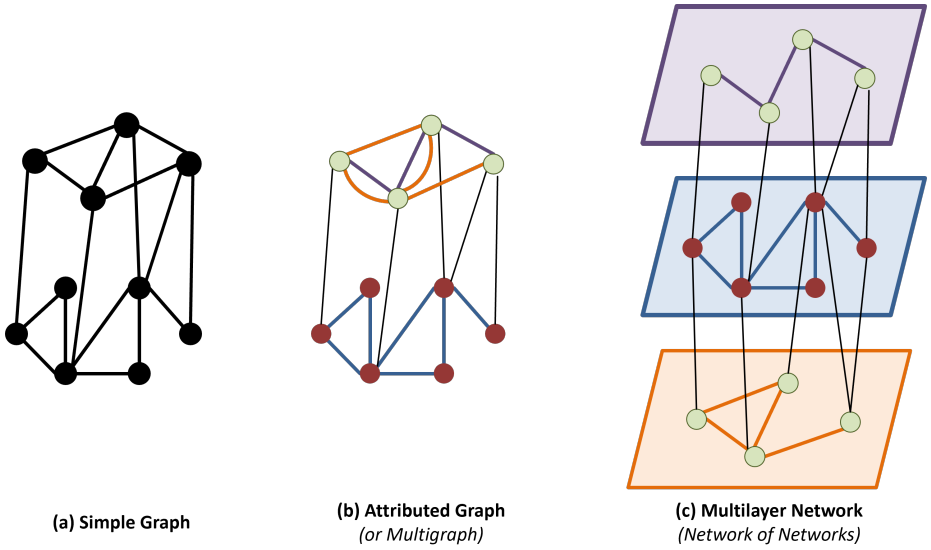


Figure 1.1: Different Types of Graph Models

1.1 Substructure Discovery in Graphs

Finding interesting patterns that occur in a graph or forest is a process known as substructure discovery, which is one of the well-studied problems in the field of graph mining. The goal of this approach is to uncover underlying regularities in the data, that is finding repetitive patterns that compresses the graph better as illustrated in Figure 1.2. Figure 1.2 shows how these substructures compresses a graph. Substructure S1 in Figure 1.2(a) can be used to compress the graph in Figure 1.2(a) to Figure 1.2(b) and the compressed graph can be further compressed hierarchically to Figure 1.2(c) using the substructure S2 in Figure 1.2(b).

The Minimum Description Length (MDL) concept is a information theoretical metric employed for this compression with the objective of identifying the substructure that best compresses a given graph. In the context of graph compression

through the identification of substructures, the MDL principle posits that an effective compression method should not only accurately capture the original graph but also incorporate a descriptive encoding of the identified substructures. The identification of significant substructures or patterns within the graph and their subsequent efficient encoding might result in a more succinct representation.

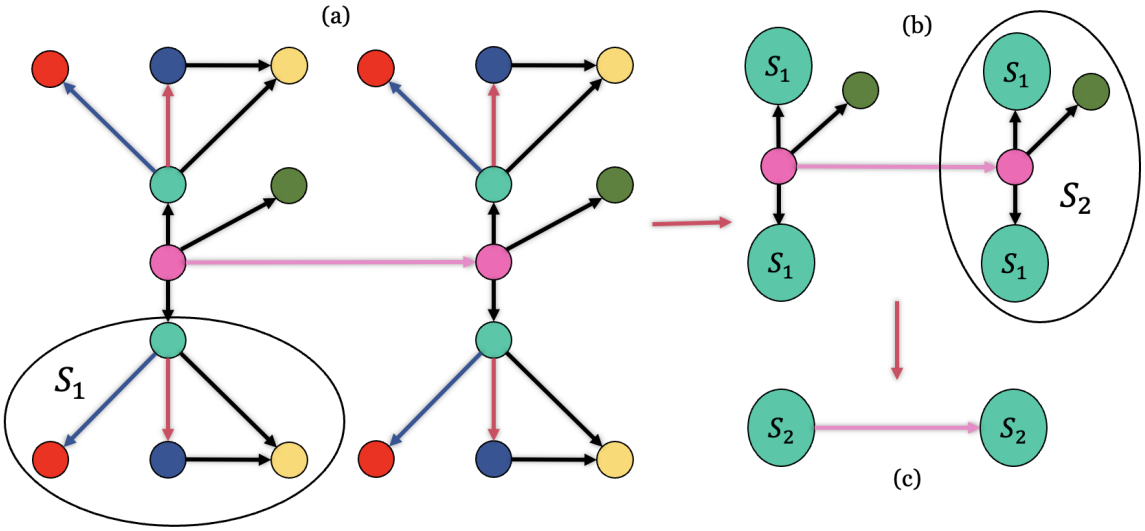


Figure 1.2: Substructures: (a) An Example Graph, (b) First Level Compression S1, (c) Second Level Compression S2

The process of uncovering substructures is done in two steps. The first step involves generating subgraphs of increasing sizes within the graph through an iterative process. The second step involves employing subgraph isomorphism to count the number of exact or similar substructure instances. The implementation of substructure discovery methods are often impacted by the following factors: graph representation, subgraph generation, detecting substructure isomorphism, and metric used (frequency or MDL) for evaluation.

1.2 Why Multilayer Networks (MLNs)?

Multilayer networks provide an alternate model for complex data sets. It can separate different relationships into independent graphs (termed layers) increasing the understandability through multiple layers. Hence, It is versatile, and easy to understand. Layers (single graphs) can also be connected by edges if relationships exits between two entity types. For example, actor and director layers can be connected by the "direct-actor' relationship where a director entity has an edge with each actor entity s/he directs in a movie. Formulation of multilayer network model for a data set is beyond the scope of this thesis and is discussed in [10]. Multilayer Networks, by their structure, also offers flexibility to process each layer individually (and in parallel) and subsets of layers can be grouped to analyze as well.

Multilayer networks, based on entity types in each layer and connectivity within and across layers, can be classified into homogeneous, heterogeneous, and hybrid multilayer networks. An example of each type of MLN is shown in Figure 1.3

Homogeneous Multilayer Networks (HoMLNs) have same (or a common subset) of entities as nodes, but connectivity may be different in each layer as it represents a unique relationship. Node labels are same across layers as they represent the same entity (e.g., same person on facebook and LinkedIn) It is used for problems where there are multiple relationships among the same set of entities/nodes.For example, consider social networks, such as Facebook, LinkedIn, and Call Network. All the entities are same and form different layers individually and relationships among them are different based on the social media application. We want to infer "groups of people who have strong "connections" in LinkedIn are also strongly connected "friends" on Facebook". Similarly, we want to find groups of people who are strongly connected through call networks as well as on Facebook and LinkedIn.

Heterogeneous Multilayer Networks (HeMLNs) have different entity types (the same entity may be an actor in one layer and director in another layer), and relationships that are different from layer to layer. There can also be edges between entities (or nodes) from two layers representing a different relationship. This model captures different associations or relationships among different entities both within (as intra-layer edges) and across (as inter-layer edges) of the data set. If we consider IMDb dataset, we have different layers with actors, directors, and movies. Actors have relationships among themselves based on the movies they have acted together in. Directors have relationships based on common genre types they have directed and Movies are linked together based on ratings. We want to find interesting patterns like actors who acted in the movies directed by same group of directors. Similarly, movies which has highest ratings directed by different directors. *This thesis mainly deals with this type of MLN. Our focus is to develop an algorithm that works directly on HeMLNs to infer best substructures without converting the HeMLN into a single graph.*

Hybrid multilayer networks (HyMLNs) as the name says "hybrid" means combination of two or more type of MLNs. Here it's the combination of Homogeneous and Heterogeneous MLNs. For example, in the IMDB dataset we have Actors as a layer, Directors as another layer now if we want to include another layer where two actors are linked if they are friends on Facebook. This is a combination of HoMLNs and HeMLNs. In this case Facebook is HoMLN and Actors and Directors are HeMLNs.

This thesis mainly focuses only on HeMLNs. Figure 1.3 shows an example of HeMLNs where layer one consists of co-actors and layer two consists of nodes representing same genre directors and a third layer consisting of movies.

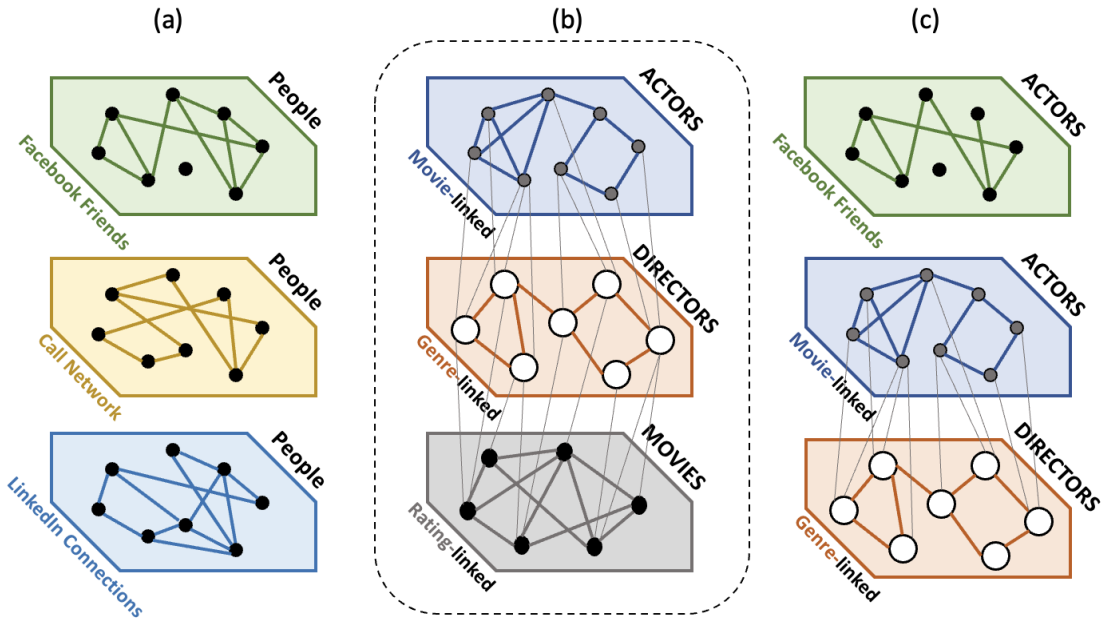


Figure 1.3: Types of MLN: (a) Homogeneous, (b) Heterogenous and (c) Hybrid

1.3 Current Approaches to Analyze MLNs

Conventional approaches for analyzing Multilayer Networks (MLNs) typically involve the mapping of networks to an equivalent single graph using various methods. In the case of homogeneous MLNs, this mapping entails aggregating the edges of the multilayer network into a single-layer network. However, this mapping process can result in the loss of valuable information inherent in the multilayer graphs.

An alternative method for analyzing MLNs, known as network decoupling, is introduced to address this issue without transforming the networks into another form. This innovative approach, proposed in [10–12], focuses on finding substructures in multilayer networks by decoupling the network into individual layers of the MLN. The decoupling approach preserves the structure and semantics of the layers in the result while leveraging existing algorithms. It operates similar to a *divide and conquer* strategy for MLNs, as illustrated in Figure 1.4(b). The application of this method is outlined as follows:

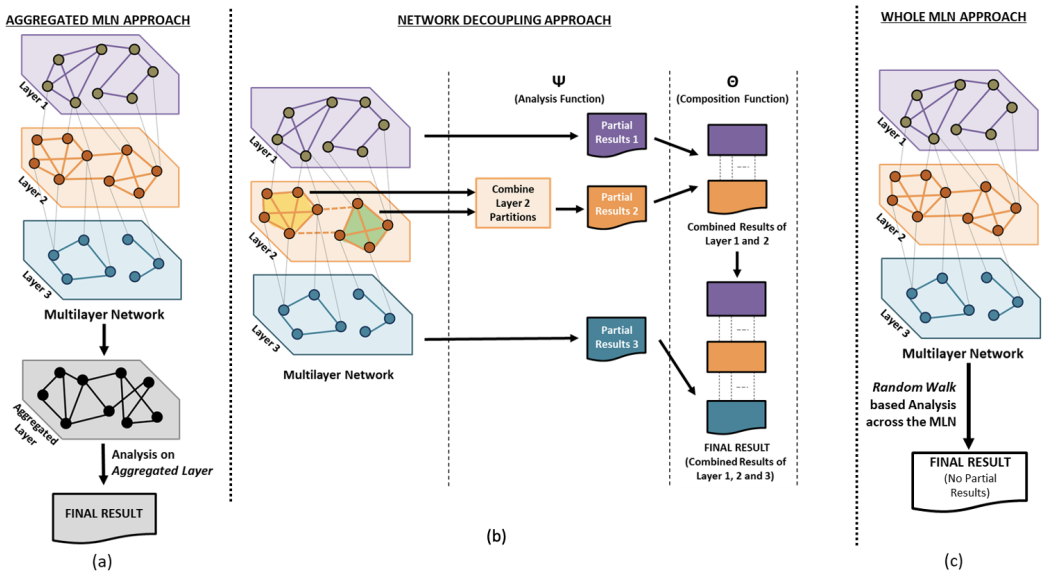


Figure 1.4: Approaches to Analyze MLNs

1. Employ the analysis function to analyze each layer individually and in parallel, considering aspects such as frequent subgraphs, community structure, centrality metrics, etc.
2. Implement a composition function to combine the partial results obtained from each layer for any two selected layers, producing intermediate results.
3. Iterate through the composition process until the desired expression is computed.

This approach stands in contrast to current methods, as illustrated in Figure 1.4(a), where aggregation-based approaches result in the loss of both structure and semantics. Additionally, Figure 1.4(c) depicts MLN approaches where only inter-layer edges are considered, neglecting all edges.

The effectiveness of the decoupling-based approach has been demonstrated, particularly in centrality [12–14] and community [11,15] analysis. However, its application to substructure discovery has been limited [16]. A comprehensive exploration

of the effectiveness of the decoupling-based approach for substructure discovery has yet to be undertaken.

1.4 Substructure Discovery in Multilayer Networks (MLNs)

There are two major approaches for finding interesting substructures in a Multilayer Network -

1. **Aggregating the layers to form a single graph:** This method aggregates the layers of the MLN to a single graph. Subsequently, simple graph mining methodologies can be employed to detect substructures within either the single or aggregated graph. However, there are certain limitations associated with this methodology.
    - When the layers are combined into one, the knowledge that is unique to each layer is lost. It's not possible to handle each layer separately and at the same time because of the aggregation process, which limits parallelization.
    - The computational cost of aggregating layers and processing might escalate as the size of the single graph grows, mostly because of the incorporation of edges from each layer. The scalability of the approach may be constrained by this factor. When dealing with exceedingly large multilayer networks, the aggregated graph becomes too large to be accommodated within the memory. Consequently, alternative strategies such as partitioning must be employed.
    - The use of aggregation may not be suitable in situations where it is necessary to focus on certain subsets of layers. In such cases, each subset would require individual aggregation, resulting in the generation of several graphs for examination.

2. **Decoupling based approach:** This method was proposed in [10–12] for finding substructures in multilayer networks that approaches the problem by decoupling the network into individual layers of the MLN. In this method, the substructures are identified inside each layer independently. Subsequently, the substructures are generated by employing a novel composition function spanning two layers, resulting in the ultimate substructures within the Multilayer Network. The binary composition has the potential to be iterated beyond two layers.

   The decoupling-based strategy possesses numerous advantages -

   (a) Preservation of MLN Structure: The Multilayer Network (MLN) structure and its modeling are conserved without any loss of information [17]. This preservation includes the semantics, such as labels in each layer. In contrast, aggregating all layers into a single graph might hide the original generated substructures, making it challenging to identify their respective layers.

   (b) Utilization of Existing Algorithms: This approach allows for the utilization of existing single-layer algorithms for substructure identification. Any existing algorithm applicable to single-layer graphs can be employed for substructure discovery in each layer, facilitating parallel processing. The natural decomposition of an MLN into layers, which are likely to be smaller, is leveraged, and the composition utilizes 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 for tailored analysis on selected layers without necessitating the processing of the entire combined MLN.

(d) Improved Parallel Processing: Handling each layer individually and in parallel enhances resource utilization efficiency. This approach enables independent processing of each layer, harnessing parallel processing capabilities for faster results.

1.5 **Problem Statement**

The problem being addressed in this thesis is to find interesting patterns in a given Heterogeneous Multilayer Network (HeMLN) without converting the MLN into a single graph. The main challenge here is to develop a composition function that produces the same result as the ground truth (GT). Union of the two layers and using the traditional algorithm is considered as the ground truth. This needs to be done by combining the interlayer edges with the intralayer edges/substructures correctly during the composition step to identify all the substructures that would have been generated in the ground truth. The existing algorithms solves the problem of finding the substructures in a single graph. We design and develop a composition function that correctly combines the substructures from each layer with the interlayer edges to find all missing substructures which exist across the layers and the resultant substructures are substructures in our HeMLN. There are many approaches introduced to find the substructures in a graph.

The algorithms in the literature [18–20] are not suitable for multilayer settings as they focus on identifying substructures in a single graph. Aggregating the MLN into a graph, which we used to build our ground truth, can detect these substructures. However, aggregated MLN analysis becomes computationally expensive and inefficient as the MLN grows.

Subdue [18] is the first main memory graph mining technique that identifies the best substructures using the minimum description length approach. Subdue builds

the graph as an adjacency matrix in main memory and mines by repeatedly expanding the vertex of a substructure of size k to a substructure of size $(k+1)$ in iteration k.

HDB-Subdue [19] uses a relational model to represent a simple graph and SQL to implement the subdue algorithm. Since RDBMS has no size restriction on relations, large graphs that cannot be accommodated in main-memory can be represented. Graphs with labeled edges, cycles, and multiple edge graphs are supported. Hierarchical substructure discovery is also supported. Unconstrained substructure expansion with duplication elimination lets HDB-Subdue analyze all feasible expansions, including numerous edges. The beam is also applied. But this technique cannot handle any graph size. M/R Subdue [20] uses the Map/Reduce distributed framework. Fundamental graph mining techniques, including systematic expansion and computation of graph similarity, have been successfully implemented within this Map/Reduce paradigm. This approach facilitates the horizontal scalability of substructure discovery through effective partitioning strategies. Consequently, Map/Reduce-based substructure discovery exhibits the capability to scale to larger graphs compared to traditional methods. We shall cover these tactics in Chapter 2.

Using M/R subdue technique, the first algorithm introduced for MLN substructure discovery was in [16]. This method uses decoupling based approach introduced in [21] to independently find substructures in each layer and then compose the substructures from each layer to find substructures across the layers. This method is an iterative method and in each kth iteration $(k+1)$ size substructures are found until a desired size substructures is found. Drawing inspiration from this approach we use similar strategy decoupling approach using map/reduce framework.

1.6 Map/Reduce Paradigm

The distributed paradigm of map/reduce [22] has been used to process very huge data. Researchers have developed Map/Reduce framework to handle massive amounts of data that can be partitioned and processed independently. Numerous businesses, including Google, Facebook, Yahoo, and Amazon use Map/Reduce to process large amount of data in the order of terabytes. Particularly, Apache Hadoop, an open-source Map/Reduce framework available in public domain, has been employed frequently.

We use Map/Reduce programming paradigm to leverage distributed and parallel processing. Map/Reduce framework has various advantages -

- Scalability
- Cost efficiency
- Flexibility
- Fast
- Parallel processing
- Availability and Resilient
- Simple programming model
- Security and Authentication

Map/Reduce automatically parallelizes and executes the program on a large cluster of commodity machines. The runtime system takes care of the details of partitioning the input data, scheduling the program's execution across a set of machines, handling machine failures, and managing required inter-machine communication. Also, one of the key features in hadoop implementation of Map/Reduce is fault tolerance [23].

A Map/Reduce framework is composed of three steps as shown in Figure 1.5:

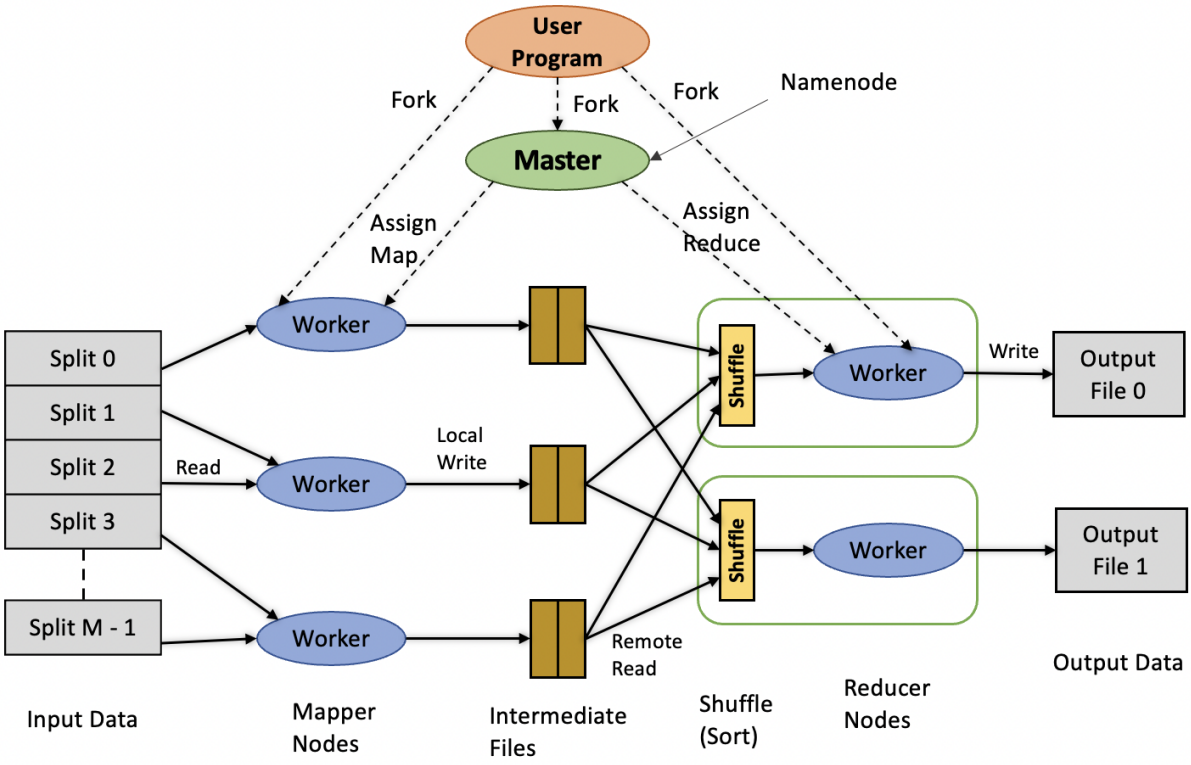


Figure 1.5: Map/Reduce Paradigm

- **Map phase**: Processes a key/value pair to generate a set of intermediate key/-value pairs
- **Shuffle phase**: Distribute the data based on partitions using keys such that all keys in a partition are sent to the same reducer.
- **Reduce phase**: Merges all intermediate values associated with the same intermediate key and invokes a user defined reduce function.

In order to find substructures in Map/Reduce, we employ a partitioning technique that was previously suggested [20]. Parallel processing of partitions has been accomplished using maps. To aggregate incomplete results and update partitions, we utilize the shuffle and reduce paradigm.

Currently, a Map/Reduce-based iterative technique is employed for the purpose of identifying substructures within partitioned graphs. The algorithm finds substruc-

tures with increasing sizes, starting from the smallest substructure consisting of two edges. It eliminates any duplicate substructures, maintains a count of identical or similar substructures, and subsequently use a metric, such as Minimum Description Length (MDL) as previously used [18], to establish a ranking for the substructures. The method is iterated until the desired substructure size is reached or there are no more substructures to generate. In order to restrict the expansion process, either all substructures or a selected subset of substructures (based on beam value) are propagated in each iteration, depending on the rank of the substructures.

This method divides a single large graph into several smaller parts and processes them into parallel. The development of algorithms that operate on a heterogeneous multilayered network graph is the main topic of this thesis.

1.7 Thesis Contributions

The contributions of this thesis are:

- Extending the work of a scalable graph mining approach for substructure discovery to a heterogeneous multilayer network, where each layer is processed separately and in parallel.
- Developing composition algorithm to correctly compose the intralayer edges and interlayer edges to find the missing substructures across the layers after each iteration.
- Developing the Map/Reduce based substructure discovery algorithm on large single graphs to work correctly on HeMLN.
- Validating the correctness of the composition algorithm.
- Extensive experimental analysis with both synthetic and real-world datasets with diverse graph characteristics.

1.8 Thesis Organization

Rest of the thesis is organized as follows -

- Chapter 2 addresses the relevant research work with regards to this thesis's inspiration.
- Chapter 3 explains all preliminaries that are used in the Map/Reduce framework for graph mining, such as input graph representation, layer expansion, duplicate removal, graph isomorphism, and partition management.
- Chapter 4 presents the design of iterative decoupling based graph mining approach and algorithm for substructure discovery in a heterogeneous multi-layer network.
- Chapter 5 explains implementation of our iterative composition algorithm more specifically giving details of how each component is utilized for substructure discovery in a Heterogeneous Multilayer Network (HeMLN).
- Chapter 6 provides drill-down analysis along with in-depth experimental study of several data sets.
- Chapter 7 draws conclusions and gives the direction for future work.

CHAPTER 2

RELATED WORK

In this chapter, we present related substructure discovery work. We will review substructure discovery studies and methods for complex datasets. We will also describe the memory-based, disk-based, and database-oriented techniques. We seek to fully comprehend the literature by critically examining each approach's strengths and weaknesses. This will let us compare our HeMLN-SD algorithm and their benefits to the work in the literature.

2.1 **Existing Graph Mining techniques**

The primary challenge in enumerating all occurrences of a specific substructure inside a graph lies in the formulation of an algorithm capable of identifying identical or similar subgraphs. The establishment of all substructures necessitates the use of a methodical approach. This process involves the systematic generation of increasingly larger substructures. In order to mitigate the exponential expansion of that particular space, a heuristic is required. The selection of top-k values (or beam size) can be determined using either frequency or Minimum Description Length (MDL) [24]. The presence of minimum description length (MDL) or higher frequency may indicate an interesting characteristic of the graph. As previously stated, graph mining for the purpose of substructure discovery can be implemented using main memory, disk, or database-based approaches. This thesis focuses on the extraction of recurring patterns and noteworthy substructures inside a Heterogeneous Multilayer Network (HeMLN). In this analysis, we will examine scholarly investigations pertaining to monoplexes

and multilayered graphs, with the aim of elucidating the progression of substructure mining techniques in response to the increasing complexity of data modeling.

2.1.1 Main Memory Approaches

Most of the early graph mining methods used main memory algorithms. These algorithms loaded a complete graph representation, usually an adjacency list or matrix, into memory for processing. This allowed algorithms to access and process the full graph and extract relevant patterns and insights from the data. This method worked because the graphs were small enough to fit in memory.

2.1.1.1 SUBDUE

The SUBDUE method is the first main memory algorithm [18]. It uses iterative process to find progressively larger substructures which are evaluated using Minimum Description Length (MDL) principle. SUBDUE uses beam to restrict the number of substructures carried to the next iteration. The algorithm starts with a one-edge substructure. This approach expands one edge in every iteration and generates best substructures after each iteration that forms candidates for next iterations. All expansions are done in this are unbounded expansion. Once all possible substructures are examined or the calculation exceeds a threshold, the method outputs the top best substructures that can best compress the graph.

The SUBDUE algorithm uses background knowledge to find better substructures, which is intriguing. Background knowledge can be applied to build associative rules in several fields. The above regulations affect algorithm framework evaluation. Every rule has a positive or negative weight that guides the substructure search. Thus, user's previous knowledge and the Minimum Description Length (MDL) concept affect substructure evaluation.

2.1.1.2 **Apriori-based approach**

AGM [25] and FSG [26] are two popular memory-based graph mining algorithms that use the Apriori-based methodology. The bottom-up Apriori-based method for finding recurrent substructures starts with smaller graphs and expands the search space. Each iteration expands substructures by merging two similar substructures with minor changes. The approach generates frequent (k+1) subgraphs from frequent k-subgraphs. Combining two substructures with two edges and one edge missing gives a three-edged substructure.

[25] introduced the Apriori property, which underpins the **Apriori Graph Mining (AGM)** approach. The AGM method optimizes search scope using the apriori property, making recurring substructure exploration more efficient and scalable.

The **Frequent Subgraphs (FSG)** technique identifies repeating substructures in graphs using an apriori approach [26]. This is different from Subdue since it entails finding intriguing substructures in a graph or forest. Canonical labeling is added to the Apriori association rule mining algorithm. The property that identical graphs have identical canonical labeling can be strategically used to identify frequent substructures. FSG determines canonical labels using a flattened graph adjacency matrix.

These main memory techniques cannot handle large and broad patterns, produce massive lists of prospective candidates, and require several database scans. The above limits and the growing size of database make current approaches unworkable for big data.

2.1.2 Disk-Based Approaches

More structural information has been added due to more sophisticated data collecting methods. Due to this, graph sizes can grow too huge for main memory storage. To address this issue, disk-based graph mining algorithms were [27–29] developed. Some graph data is saved in memory, while the rest is stored on disk. Indexing disk-based graphs seemed like the best option due to the difficulties and high costs of random access. Due to their database update stability, frequent substructures are ideal for indexing. This makes incremental index maintenance cost-effective. This is especially useful for indexing huge graphs due to their many substructures.

The gIndex approach [30] use frequent substructures as units for indexing. A substructure is considered to be common when its frequency surpasses a predetermined minimum support threshold. The process of indexing facilitates the immediate retrieval of frequent substructures. The construction of gIndex can be achieved using a single database scan by utilizing incremental updating.

However, many graph indexing methods need computationally demanding index building. For rapid and efficient access, a compact index structure that fits in main memory must limit indexing features. When the graph is huge, the index grows correspondingly. This causes expensive and inefficient index building and index structures that may be too large for memory. Considering these challenges, efficient methods are crucial.

2.1.3 Database-oriented Approaches

Disk-based algorithms efficiently manage graph segments that exceed memory available for processing. However, these solutions need explicit data marshalling between disk and main memory. This element must be carefully integrated into the algorithm's design and execution. The speed of data transmission between the disk

and memory, the buffer size, buffer maintenance and replacement rules, and hit rates can all affect disk-based methods. Database Management Systems (DBMSs) have effective buffer management and query optimization strategies. This involves matching graph mining techniques with SQL queries and using a Database Management System (DBMS) for data storage to take use of its sophisticated optimizations. We used the strategy from [19, 31] in this study. We will now discuss the methodologies.

2.1.3.1 **HDB-Subdue**

DB-Subdue and EDB-Subdue proposed in [32] are initial effort towards implementing a database-centric approach for graph mining. In this paradigm, the graphs are stored directly in a database as relations. The identification of the best substructures within a graph is achieved by determining the frequency of instances of those substructures present in the graph. These two approaches are modified to create HDB-Subdue [19]. It supports more graph structures, including cycles and multiple edges between vertices. HDB-Subdue permits unlimited substructure extension to overcome EDB-Subdue's constraints. The uncontrolled expansion can generate fake duplicate instances of the same substructure. This can happen when the same instance is enlarged or enumerated in a different sequence. HDB-Subdue sorts substructure instances by vertex counts and connection maps to discover and eliminate pseudo-duplicates. In order to count substructure instances, HDB-Subdue uses vertex labels and connection properties. Hierarchical graph reduction is another feature of HDB-Subdue. After finding the optimum substructure for an iteration, HDB-Subdue compresses the graph by replacing all instances with one vertex.

2.2 Distributed Approach to Graph Mining

As discussed, Graph sizes can be large, making main memory-based systems ineffective. In response, alternative methods save the graph on disk and load it into memory when needed. Disk-based techniques can handle graphs larger than memory, but they present additional issues like as buffer customization and I/O latency [33]. Big data makes conventional systems unsuitable, necessitating distributed data administration and processing. Numerous distributed graph databases, such as Neo4J, ArangoDB, and Dgraph, are suitable for large-scale processing. These databases favor graph data storage over graph mining.

A number of graph mining methods have shown success in cloud-based deployments [34–36]. Additionally, research has explored patterns in big graphs using the Map/Reduce framework [34]. However, the pattern searching technique in [34] requires a specified pattern to search for all instances of that pattern in the graph. If we need to find a pattern with the highest compressibility, we cannot supply it beforehand. Research is underway to split large graphs into manageable parts for distributed processing across computational resources [37].

2.3 Substructure discovery using MapReduce

M/R Subdue, as introduced in [20], adapts the Subdue algorithm to the Map/Reduce paradigm, transforming the original main memory approach into a distributed framework. By leveraging Map/Reduce's distributed processing capabilities, the algorithm operates on graph partitions, facilitating parallel processing and scalability across a cluster of commodity machines. This approach significantly enhances the efficiency of large-scale graph data mining.

The algorithm initiates substructure discovery by partitioning the input graph into smaller segments and aggregating results across these partitions. It iteratively generates substructures of increasing sizes, starting from a single-edge substructure. Duplicates are eliminated, and a ranking metric, such as Minimum Description Length (MDL), is applied. Each iteration expands substructures by one edge using adjacency lists for each partition. The process continues until a specified substructure size is reached or no further substructures can be generated. After each iteration, a subset of the best substructures is chosen for further consideration, updating substructure partitions accordingly. This allows for an efficient and distributed exploration of meaningful substructures within large graphs.

In the Map/Reduce process, the Mapper handles each input record independently. The input graph is represented as a sequence of graph edges, with each edge denoted by an edge label, source vertex ID and label, and destination vertex ID and label. Initially, substructure partitions are disjoint to prevent duplication of the same edge across multiple partitions. The adjacency list is partitioned accordingly. After the first iteration, new edges are integrated into existing substructure partitions, requiring updates to the adjacency list for each affected partition. Careful consideration of the partitioning and updating approach is essential for the algorithm's effectiveness.

2.4 **Substructure discovery in Multilayer Networks**

Although most of the methods up to this point have only dealt with single graphs, in practice, enriched graphs containing complicated relationships are the norm. A good example of this complexity is a graph with several layers, where each layer represents a different set of connections between vertices. One way to visualize these levels is in a Multilayer Network (MLN), where different types of relationships are represented by different layers.

The MLN-Subdue algorithm, as described in the previous work [16], utilizes a decoupling-based approach that is specifically tailored for Homogeneous Multilayer Networks (HoMLN). This approach is motivated by the substructure discovery techniques used for single graphs, such as M/R Subdue [20] which uses decoupling based approach introduced in [21]. The objective of this approach is to identify substructures within HoMLN, where each layer is analyzed independently and in parallel, without aggregating the layers into a single graph. In this technique, the composition algorithm executed within the Map/Reduce framework and possesses the capability to effectively process an arbitrary quantity of layers within the Homogeneous Multilayer Network. The processing of each layer occurs in parallel, with the substructures generated for each layer being combined after each iteration to find substructures that span across many layers of the MLN. Here the key components of graph mining, including as substructure generation, composition of substructures across layers, elimination of duplicates, and counting of isomorphic substructures, has been successfully included into the Map/Reduce paradigm.

This process entails an iterative approach to independently identify the substructures of size k within each layer. The process begins with k being set to 1. Afterwards, a composition function is employed to ascertain substructures of a specific size, k, that are present throughout all layers. The composition algorithm is based upon the idea that in a HoMLN, each connected substructure is connected to a common vertex or node. This vertex serves as a connection for edges that originate from different layers.

Taking into account the prior research conducted on Substructure Discovery on Homogeneous Multilayer Networks (HoMLN), our objective is to propose a substructure discovery algorithm specifically designed for a Heterogeneous Multilayer Network (HeMLN) in the context of a Map/Reduce distributed environment. Range partition-

ing is employed in order to create the initial partitions. In this study, we present a proposed composition technique that employs an iterative approach to identify the missing substructures across the layers. In the forthcoming chapters, we shall describe the foundational aspects pertaining to this algorithm and provide a comprehensive details of its design and implementation.

CHAPTER 3

PRELIMINARIES

In this chapter we will briefly introduce the concepts required for the substructure discovery. We will discuss all the definitions and concepts that are used in this thesis.

3.1 Graphs

Graphs, as a data structure, represent a collection of nodes or vertices interconnected by edges. This fundamental structure is characterized by its ability to model intricate relationships and connections within a system. Each node in the graph can hold data, and the edges define the relationships between these nodes. Graphs can be directed, where edges have a specific direction, or undirected, where edges have no inherent direction. Additionally, graphs may contain weighted edges, assigned with a numerical value or a label. These have applications in various domains, from computer science algorithms and network analysis to social network modeling and transportation systems. For instance, consider a social media platform where users are nodes, and friendships or connections are edges. This inherent flexibility of graphs makes them a fundamental and powerful concept in the representation and analysis of interconnected data.

3.2 Input Layer Graph Representation

The choice of graph representation can have a significant impact on the efficiency and effectiveness of various algorithms and analyses applied to the data. Different

types of graphs and representations are suited to different scenarios based on the nature of the graph, the operations to be performed, and efficiency considerations. There are three common graph representations - adjacency matrix, adjacency lists and edge lists. Adjacency matrix can be used for dense graphs applications where the number of edges is close to the maximum possible edges and can be more space-efficient than an adjacency list. If the graph is sparse (contains relatively few edges compared to the maximum possible), an edge or adjacency list [38] representations are often more memory-efficient than an adjacency matrix. In this thesis, we deal with the large *labeled* graphs, commonly exhibiting sparsity. So we represent our layer graph using both *adjacency list* and *edge list* data structures. These representations are applicable to both directed and undirected graphs, but we have specifically focused on directed graphs in our analysis. Figure 3.1 shows an example of input graph corresponding to a layer in a MLN.

Our methodology can be easily expanded to encompass undirected graphs as previously discussed in [20]. Given that undirected graphs lack a clear source-destination link, it is possible to transform them into directed graphs by substituting each edge with two directed edges. However, this conversion results in the creation of loops inside the graph. Graph traversals handles these loops by maintaining a record of visited nodes or edges. For graphs with bi-directional edges, a sixth element denoting the direction of the edge, need to be added to the current edge representation.

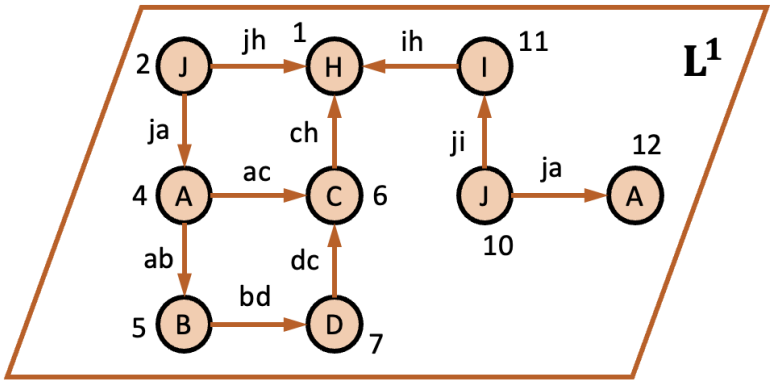


Figure 3.1: Input Layer Graph

3.2.1 **Edge List**

Our input layer graph is represented as a list of unordered edges (or 1-edge substructure) including its direction. Each edge is completely represented by a 5 element tuple - $< E_l, V_{sid}, V_{sl}, V_{did}, V_{dl} >$ where, E_l is edge label, V_{sid} is source vertex Id, V_{sl} is source vertex label, V_{did} is destination vertex Id and V_{dl} is destination vertex label. In this context, it is important to note that the vertex Ids (V_{sid} and V_{did}) are guaranteed to be unique. However, it is not necessary for the vertex labels (V_{sl} and V_{dl}) and the edge label (E_l) to possess uniqueness. Table 3.1 shows the edge list representation for the input layer graph shown in Figure 3.1.

This representation is generic and can be extended for multiple edges as well by using an edge identifier in the edge representation to demarcate multiple edges between same two nodes. This representation of a directed edge is used to represent a k-edge substructure (a connected graph with k edges) as a collection of k 1-edge substructures. Our algorithm takes a input as graph represented as a text file with a 1-edge substructure in each line.

3.2.2 Adjacency List

Adjacency list is a graph representation where each vertex is associated with an edge list (list of 1-edge substructures) in which each edge depicts the connection between the vertex and its neighboring vertex. For each vertex, the size of the edge list is the sum of the in and out degrees of that vertex. The adjacency list is used for performing expansion by an edge for each node in a subgraph. By sequentially traversing the edges in an adjacency list, one can systematically expand the substructure by one edge from any vertex, uncovering the interconnected relationships within the graph and facilitating a detailed exploration of its components. Table 3.2 shows the L1 adjacency list of the input layer graph as shown in Figure 3.1.

Edge List
⟨ab,4,A,5,B⟩
⟨ac,4,A,6,C⟩
⟨bd,5,B,7,D⟩
⟨ch,6,C,1,H⟩
⟨dc,7,D,6,C⟩
⟨ja,2,J,4,A⟩
⟨jh,2,J,1,H⟩
⟨ih,11,I,1,H⟩
⟨ja,10,J,12,A⟩
⟨ji,10,J,11,I⟩

Table 3.1: Edge List

Vertex ID	**Adjacency List**
1	⟨ch,6,C,1,H⟩;⟨jh,2,J,1,H⟩;⟨ih,11,I,1,H⟩;
2	⟨ja,2,J,4,A⟩;⟨jh,2,J,1,H⟩;
4	⟨ab,4,A,5,B⟩;⟨ac,4,A,6,C⟩;⟨ja,2,J,4,A⟩;
5	⟨ab,4,A,5,B⟩;⟨bd,5,B,7,D⟩;
6	⟨ac,4,A,6,C⟩;⟨dc,7,D,6,C⟩;⟨ch,6,C,1,H⟩;
7	⟨bd,5,B,7,D⟩;⟨dc,7,D,6,C⟩;
10	⟨ja,10,J,12,A⟩;⟨ji,10,J,11,I⟩;
11	⟨ih,11,I,1,H⟩;⟨ji,10,J,11,I⟩;
12	⟨ja,10,J,12,A⟩;

Table 3.2: Adjacency List

3.3 Graph Partitioning:

A MLN layer may be too large to fit in a main memory. If so, our goal is to partition the layer L_i into p partitions (L_1^i, L_2^i,..., L_p^i) that are small enough to fit in main memory.

Range − based partitioning is used to create the partitions using the vertex ids [20]. Each partition is a range of node ids and the size of each partition need not be same. There can be missing vertex ids in a given range. For example, if a graph has 5000 vertices starting from1 to 6000, ranges could be 1 to 1000, 1001 to 3000, 3001 to 6000 making 3 range-based partitions.Range information divides the adjacency list into a range of vertex Ids. We can change the size and number of partitions to accommodate RAM. As the ranges are disjoint, the adjacency list of the partitions are also disjoint. Each vertex Id in a range and its adjacency list corresponds to a single adjacency list partition. If neighboring nodes are in two partitions, the edge connecting them will be in the adjacency list of both partitions. As a result, during expansion, the same substructure can belong to many adjacency partitions. Because adjacency partitions are connected based on vertex Ids, each substructure is only expanded once. As a result, partitioning produces no duplicates but there may be duplicates across the partitions. As shown in Figure 3.2. Partition L_1^i is assigned vertex Ids from 1 to 6, whereas partition L_2^i is allotted vertex ids 7 to 12. Hence range 6 is fixed for both partitions. Both partitions are connected by blue edges, which appear in different adjacency list partitions. The adjacency list partitions are displayed in table 3.3 along with range info table. The edges of both adjacency list partitions are $< bd, 5, B, 7, D >$, $<dc, 7, D, 6, C>$, and $<ih, 11, I, 1, H>$.

3.4 **Graph Expansion**

For substructure discovery, we systematically generate substructures of progressively increasing size to get the best substructures that can compress the entire graph better . And in this context, graph expansion plays a crucial part in the process of uncovering substructures of varying sizes. Since the input to our algorithm is 1-edge substructure, we start by expanding this 1-edge by adding one edge in every possi-

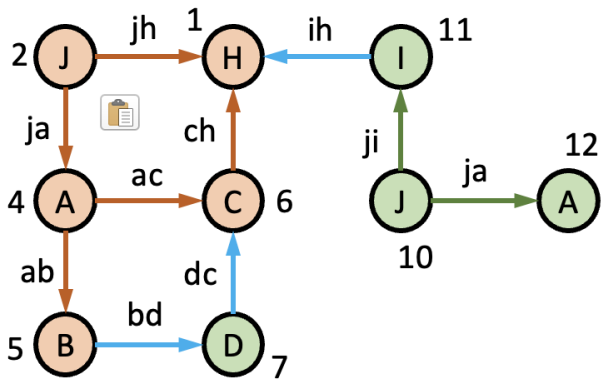


Figure 3.2: Graph partitioning

Vertex ID	Adjacency List Partition 1
1	⟨ch,6,C,1,H⟩;⟨jh,2,J,1,H⟩;⟨ih,11,I,1,H⟩;
2	⟨ja,2,J,4,A⟩;⟨jh,2,J,1,H⟩;
4	⟨ab,4,A,5,B⟩;⟨ac,4,A,6,C⟩;⟨ja,2,J,4,A⟩;
5	⟨ab,4,A,5,B⟩;⟨bd,5,B,7,D⟩;
6	⟨ac,4,A,6,C⟩;⟨dc,7,D,6,C⟩;⟨ch,6,C,1,H⟩;

Vertex ID	Adjacency List Partition 2
7	⟨bd,5,B,7,D⟩;⟨dc,7,D,6,C⟩;
10	⟨ja,10,J,12,A⟩;⟨ji,10,J,11,I⟩;
11	⟨ih,11,I,1,H⟩;⟨ji,10,J,11,I⟩;
12	⟨ja,10,J,12,A⟩;

Pid	Range
p1	1-6
p2	7-12

Table 3.3: Adjacency List Partitions

ble direction. Our expansion is based on independently growing each substructure into a number of larger substructures at each iteration, with each node of the input substructure being expanded by adding a single edge incident on that node. This expansion process is unconstrained to guarantee the accuracy of substructure discovery. And such an unconstrained expansion leads to **duplicates** as shown in Figure 3.3 which are exactly **same** in *labels*, *connectivity* and *vertex Ids*. To identify these duplicates, we introduce the concept of canonical instance.

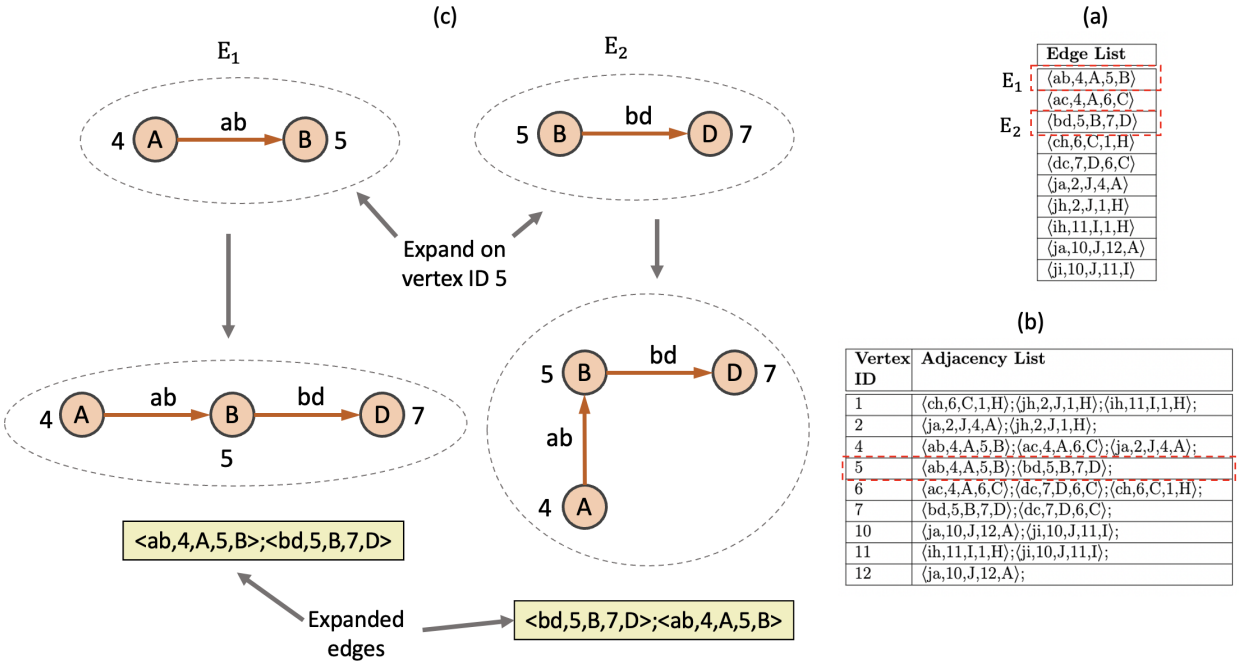


Figure 3.3: Graph Expansion: (a) Edge list, (b) Adjacency list, and (c) Expansion on vertex ID 5

3.5 **Canonical Instance:**

To remove **duplicates** we convert the expanded instance to its *canonical form*. We use a *lexicographic ordering technique* on edge label for canonical form. In the case of a substructure containing multiple edges with identical edge labels, these edges are arranged based on the source vertex label. If the label of the source vertex is the same, they are additionally sorted based on the label of the destination vertex. In the case where both the edge label and vertex labels are identical, the ordering of the source and destination vertex Ids is used. Figure 3.4 shows how the duplicates are represented in canonical form and are removed.

The use of canonical representation of instances proves to be advantageous in the process of finding and then eliminating duplicate instances. However, the task of determining the frequency of isomorphic substructures necessitates the use of a canonical form that is independent of vertex identification. Consequently, a canonical

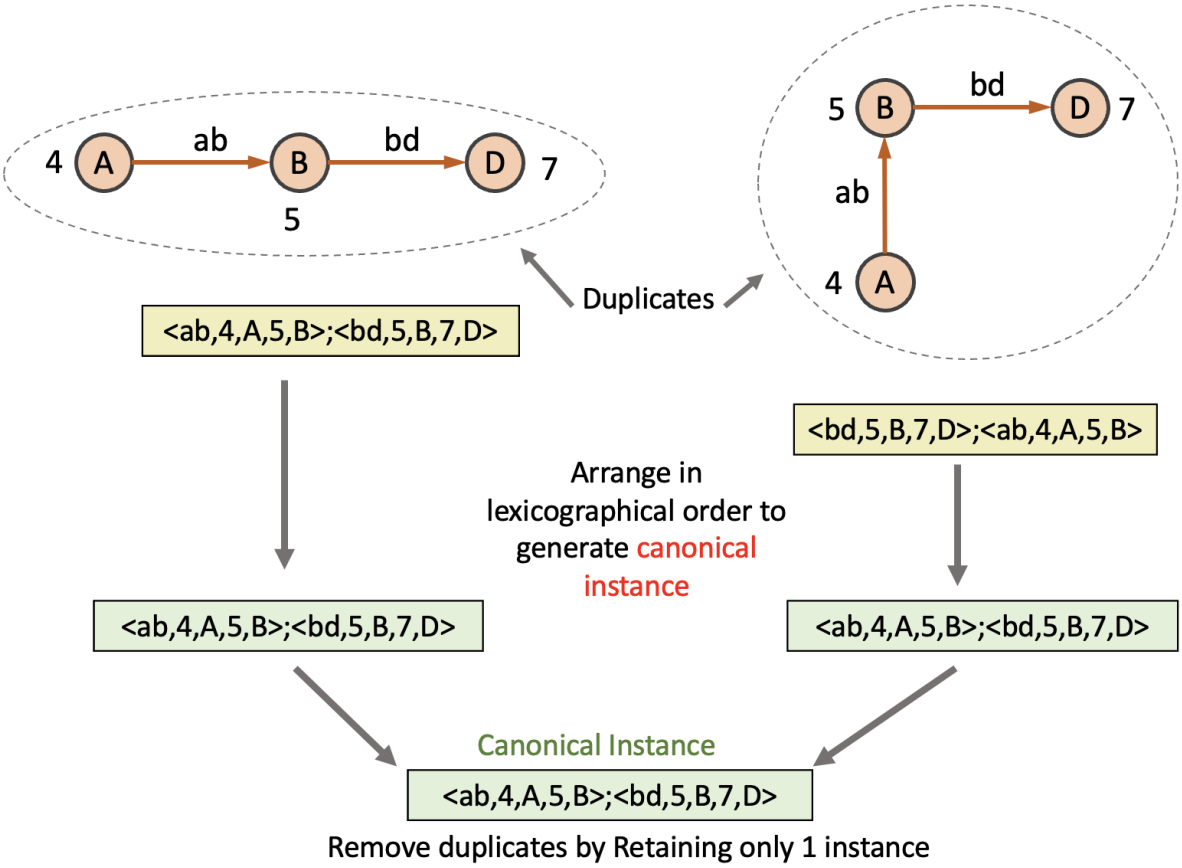


Figure 3.4: Canonical Instance

form of the substructure is obtained by ordering the vertex Ids relatively, using the canonical instances.

3.6 Graph Isomorphism

The concept of graph isomorphism refers to the idea that two graphs have the same underlying structure, regardless of any differences in their labels, vertex IDs, or edge orientations. The incorporation of isomorphs is crucial in accurately computing frequent substructures. Isomorphs are characterized by having identical labels for their vertices and edges, while maintaining different IDs for each vertex. To determine their identity, it is important to create a canonical substructure using

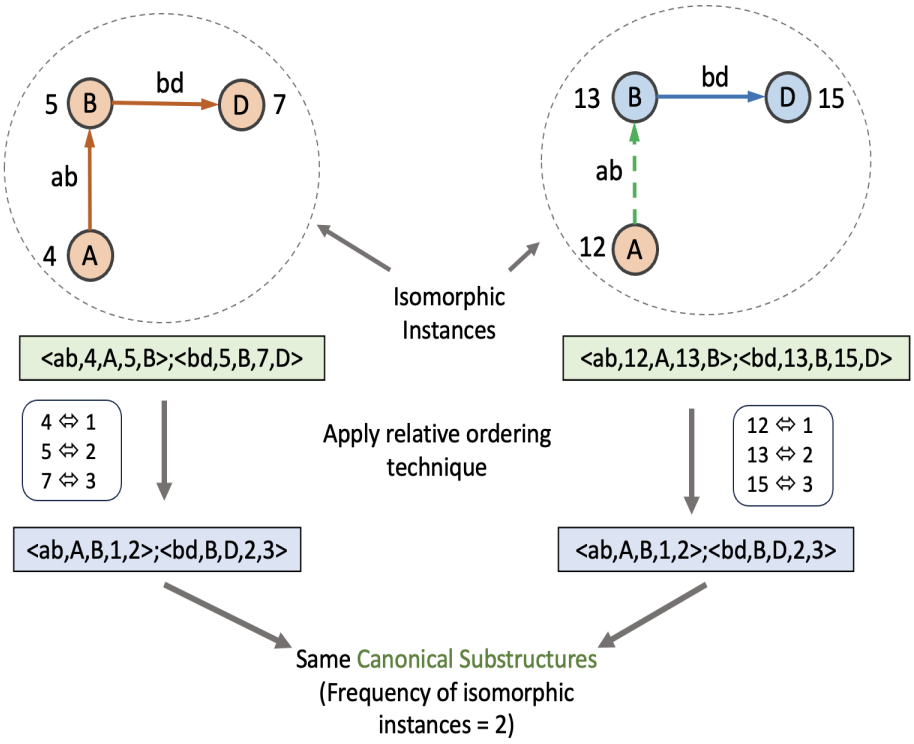


Figure 3.5: Canonical Substructure

the canonical instance. [20] states that the relative ordering of vertex identifiers in any two isomorphic substructures is same. As a result, they will have a comparable canonical substructure.

Given that the canonical instance already adheres to a lexicographic ordering, it is possible to create a canonical k-edge substructure by arranging the distinct vertex identifiers in the sequence of their occurrence within the canonical instance. Hence, the canonical substructure can be obtained by substituting each vertex identifier with its corresponding positional value within the instance. The resultant canonical substructure enables us to readily identify isomorphs. The creation of a canonical substructure from the canonical instance is exemplified in Figure 3.5. It should be noted that the isomorphs are distinct canonical instances, however their relative locations subsequent to the canonical instance remain consistent. The vertex id $(4, 5, 7)$ for the first instance and $(12, 13, 15)$ for the second occurrence can be simplified to $(1, 2, 3)$

by relative placement. Consequently, both entities possess an identical canonical substructure. The generation of canonical labels plays a vital role in our technique, since it facilitates the identification of duplicates and isomorphic instances. This thesis uses the notion of a canonical instance and a canonical substructure to distinguish between duplicate instances and isomorphic substructures, correspondingly.

3.7 Metric for Ranking the Substructures

When it comes to the process of discovering substructures, the notion of what constitutes an interesting substructure can change depending on the objective of the study. The metric choice depends on how you define an interesting substructure. Finding the most common substructure or the one that compresses a graph best may be the goal. Two common substructure ranking and relevance metrics are compression and frequency. In graph mining, MDL [3] and frequency are common ranking measures. The former emphasizes the substructure's ability to compress the graph, while the later just considers number of instances or frequency. Both metrics need counting instances.

3.7.1 Minimum Description Length

The Minimum Description Length (MDL) metric is a domain-independent measure that has been demonstrated to emphasize the significance of a substructure in terms of its ability to compress a complete graph. The description length of a graph G refers to the minimum number of bits needed to represent and encode the graph. The calculation of Minimum Description Length (MDL) for a substructure S in a graph G involves the use of the formula MDL = (DL(S) + DL(G—S))/DL(G). Here, DL(S) represents the description length of the substructure being evaluated, DL(G—S) represents the description length of the graph G when compressed by representing each

instance of the substructure as a node, and DL(G) represents the description length of the original graph. The pattern in a graph that achieves the highest level of compression, namely by minimizing the combined measure of DL(S) and DL(G—S), is regarded as the best substructure. Both the frequency of the subgraph and its connectivity have an impact on compression.

Instead of bits, we compute MDL using the number of vertices and edges called DMDL (Database Minimum Description Length) proposed in [19]. Below is the DMDL formula -

$$MDL = \frac{V + E}{[V - (v * f) + v] + [E - (e * f)] + [v + e]}$$

where V = Total no. of nodes in MLN,

E = Total no. of edges in MLN,

v = Total no. of nodes in Substructure,

e = Total no. of edges in Substructure,

f = frequency of the isomorphic instances of substructure.

3.7.2 Frequency Calculation

1. **Overlap-independent Frequency** In this metric, all instances of overlapping substructures are regarded as independent occurrences and are included in the counting process. As shown in the Figure 3.6, the frequency calculation yields a value of 4, despite the presence of a considerable degree of overlap among the cases. In the uppermost pair of substructures, there is an overlap between

vertex Ids 4 and 8. Conversely, in the lowermost pair of substructures, there is an overlap between vertex Ids 1 and 8. The presence of overlaps in the data leads to a discrepancy in the frequency count, making it an inaccurate measure of frequency when compared to separate substructures that share the same vertex and edge labels. Consequently, this frequency measure is not commonly employed in academic research.

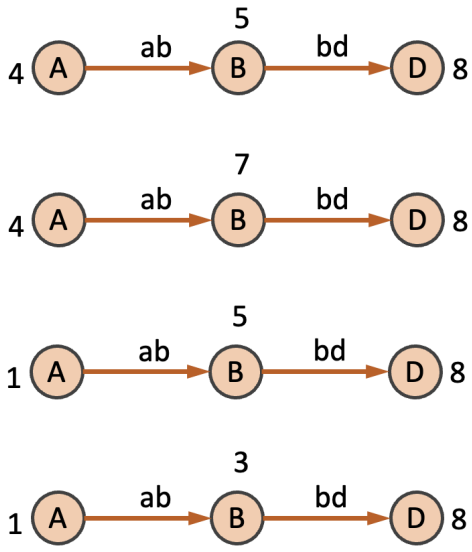


Figure 3.6: Overlapping Instances

2. **Overlap-cognizant Frequency** In this metric, overlapping instances of substructures are not considered as independent occurrences and, as a result, is not included in the count. In order to determine the number of non-overlapping occurrences, we use the Most Restrictive Node (MRN) metric, which has been previously introduced in scholarly works [39, 40].

   The Most Restrictive Node (MRN) refers to the specific node inside a certain substructure that exhibits the lowest frequency of occurrence throughout the

graph. The calculation of the frequency of a substructure involves multiplying the number of occurrences of the MRN in the graph by the number of occurrences of the remaining nodes in the substructure that are not directly connected to the MRN. The computation of overlap-cognizant frequency involves determining the maximum number of non-overlapping instances given a canonical substructure. As depicted in Figure 3.6, the function F(A) is represented by the interval [1,4], F(B) is represented by the interval [3,5,7], and F(D) is represented by the interval [8]. The minimum value within the set 2, 3, 1 is 1, which indicates that node D is the MRN. The frequency of the substructure is also 1. Nevertheless, the presence of overlapping instances can hold significance in specific settings. Therefore, we also monitor occurrences that overlap and employ both sets of data to calculate the frequency and Minimum Description Length (MDL) for a specific substructure.

In this chapter, we have discussed the preliminaries related to this thesis. In the next chapter, we shall discuss the design of our composition algorithm.

CHAPTER 4

DESIGN

This chapter focuses on the design of the composition algorithm and the overall methodology employed in the context of Substructure Discovery in Heterogeneous Multilayer Networks (HeMLNs). In this discussion, we will explore the challenges encountered during composition and subsequently explain why this approach is correct.

4.1 Overview of Design

The general framework proposed for analysing the Multilayer Networks (MLNs) which processes each layer of MLN independently and in parallel according to the **decoupling-based approach** developed in [21]. This approach is based on **divide-and-conquer** technique which basically involves two phases: **Analysis phase (ψ)** for analyzing each network layer independently and then **Composition phase (θ)** for combining the results of each layer using a composition function.

In the context of substructure discovery in HeMLN, based on the decoupling-based approach philosophy, composition will be done in each iteration. Note that it is also possible to compose after some iterations or at the end of substructure discovery for each layer. In this methodology, we systematically generate substructures of progressively increasing sizes within each layer referred to as intralayer substructures (analysis phase) and these expanded substructures will then be composed with inter-layer edges to get the missing substructures (referred to as composed or inter-layer substructures) across the layers (composition phase) to obtain all the substructures in HeMLN. During this process, we perform expansion, remove any duplicate instances,

quantify the number of isomorphic substructures, and apply a ranking metric. Next, we apply the top-beam technique to limit the scope of our search, exclusively considering these beam substructures as potential candidates for further expansion in the subsequent iteration.

The main focus of this thesis is to find the missing substructures that span layers using interlayer edges which ultimately lead to finding all the substructures within the entire Heterogeneous Multilayer Network (HeMLN).

In our composition algorithm, we basically perform expansion using interlayer edges. For this we generate adjacency list out of expanded substructures of each layer and then use it to expand interlayer substructures to generate substructures that span layers. Additionally, we find substructures consisting only of interlayer edges to obtain all the missing composed substructures in HeMLN. We also generate adjacency list from composed substructures to be used in the next iteration. This composition is performed iteratively in order to generate progressively increasing size of substructures and hence gives 100% accuracy, as there is no loss of information between iterations.

We address the following aspects in our design -

1. **Scalability:** Scalability is effectively managed by the use of the "Divide and Conquer" approach, which is employed to effectively handle large layer graphs that exceed the memory capacity of a single system. The input graph of i-th intralayer (L^i) is partitioned into p smaller graph partitions (shards), denoted as $L^i_1, L^i_2, \ldots, L^i_p$. Similarly, input graph of (i, j)-th interlayer ($L^{i,j}$) is partitioned into p smaller graph partitions, denoted as $L^{i,j}_1, L^{i,j}_2, \ldots, L^{i,j}_p$.
2. **Resource Utilization:** In order to exploit parallel processing capabilities on graph partitions, we choose a strategy where p partitions are processed concurrently, utilizing k processors simultaneously. It is worth noting that the value

of k might range from 1 to p. The value of k can be adjusted in response to the availability of resources. For instance, in the event that a sufficient quantity of resources is available, it is feasible to progressively augment the value of k until it reaches the upper limit of p.

4.2 Challenges involved in Composition

Substructures discovery in a Multilayer Network (MLN) differs from discovering them in a single graph as discussed in [20]. Finding substructures in a Multilayer Network (MLN) requires considering numerous parameters as stated below to effectively identify them -

1. Substructures can only be found in intralayer.
2. Substructures can exist across multiple layers (interlayer)

And finding substructures specifically in HeMLN involves considering interlayer edges in composition. And hence challenges involved in composition are -

1. To correctly compose the expanded intralayer substructures with interlayer edges to find missing substructures across the layers.
2. To correctly compose the substructures only using interlayer edges. (Substructures within interlayer graph)

Therefore, finding the substructures within layers (intralayer) and across the layers using interlayer edges is the matter of finding all the substructures within HeMLN. Since we will be having layer partitions (both intralayer and interlayer) as discussed previously, substructures will be spanning in these partitions. We need to correctly and efficiently find the substructures in all partitions by properly handling expansion and composition and removing duplicates in each of these steps.

4.3 Iterative Decoupling-Based Approach for Substructure Discovery in HeMLN

We use iterative Decoupling-Based approach for our HeMLN-SD developed in [41], to facilitate substructure discovery. The primary objective is to process each layer of the Multilayer Network independently and in parallel. The main difference lies in the composition phase, where we incorporate our composition algorithm to identify missing substructures across the layers. This approach is based on "divide-and-conquer" technique to generate substructures for individual layers and then compose substructures from individual layers using interlayer edges to find substructures across the layers. In particular, we use the composition function after each iteration, rather than after identifying all substructure sizes in each layer. To illustrate, we create a k-edge substructure in each layer ($k = 1$ for the 1st iteration) and expand each vertex by adding one edge. The composition function is applied to expanded instances of each layer to identify substructures across the layers. This process is repeated until a termination condition is applied. Figure 4.1 shows overall iterative decoupling-based approach for substructure discovery in a heterogeneous multilayer network.

Because we are primarily concerned with large graphs, we assume that a single machine's memory cannot possibly contain all of the necessary information (layer data and its adjacency list). As a result, partitioning graph across different processors is an absolute necessity. The earlier work that was done on Map/Reduce-based substructure discovery served as the basis for this partitioning technique that we have developed [20].

Using a decoupling approach instead of aggregating the layers into one graph has many benefits that we have already seen in chapter 1.

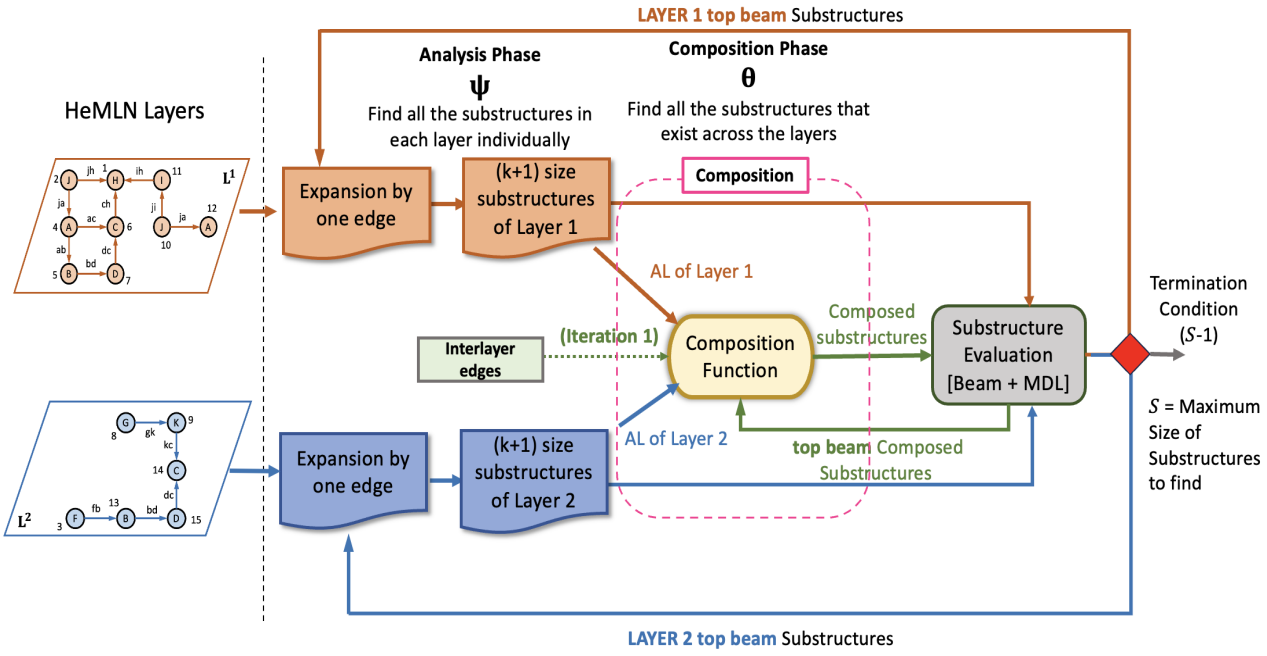


Figure 4.1: Iterative Decoupling-Based Approach for Substructure Discovery in HeMLN for k-iterations

4.4 HeMLN-SD Iterative Composition Algorithm (ICA)

In this section, we shall discuss the analysis and composition phases of our iterative decoupling based approach for HeMLN-SD in detail.

4.4.1 Analysis Phase

In this phase, we process each layer of HeMLN independently and in parallel to find the substructures of size k in iteration $(k-1)$. Finding the substructures follows the same technique of expansion where in we expand a k-edge instance to form a $(k+1)$-edge instance. Duplicates are generated during this phase and are eliminated by converting them to canonical instance. As we use these generated instances for composition, we create the adjacency list out of the expanded and unexpanded instances which forms the input to our composition phase. Here, we also add unexpanded instances in adjacency list because these edges may be composed with

interlayer edges to form $(k+1)$-edge expanded instance. This adjacency list helps us to find the missing substructures across the layers. Finally we generate the canonical substructure from canonical instance which will be used for substructure evaluation after composition phase.

4.4.2 Composition Phase

In this phase, we perform the composition of intralayer substructures using interlayer edges to generate missing substructures across the layers. Our composition has 3 main steps -

1. **Expand k-edge interlayer instance using L^1 adjacency list** $[L^1_{AL^k}]$ (Figure 4.2)

   In this step, we expand k-edge interlayer instance (1-edge interlayer edge in 1st iteration) by one edge to form $(k+1)$-edge interlayer instances using layer 1 intralayer adjacency lists generated in current kth iteration from expanded and unexpanded intralayer instances (i.e. k-edge input interlayer instances) of layer 1. Figure 4.3 shows the expansion in 2nd iteration using layer 1 intralayer adjacency list.

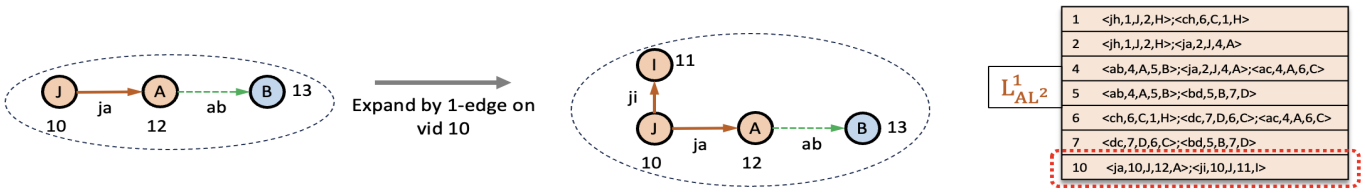


Figure 4.2: Expansion using $L^1_{AL^2}$ in iteration $k = 2$

2. **Expand k-edge interlayer instance using L^2 adjacency list** $[L^2_{AL^k}]$ (Figure 4.2)

In this step, we expand k-edge interlayer instance (1-edge interlayer edge in 1st iteration) by one edge to form $(k+1)$-edge interlayer instances using layer 2 intralayer adjacency lists generated in current kth iteration from expanded and unexpanded intralayer instances (i.e. k-edge input interlayer instances) of layer 2. Figure 4.3 shows the expansion in 2nd iteration using layer 2 intralayer adjacency list.

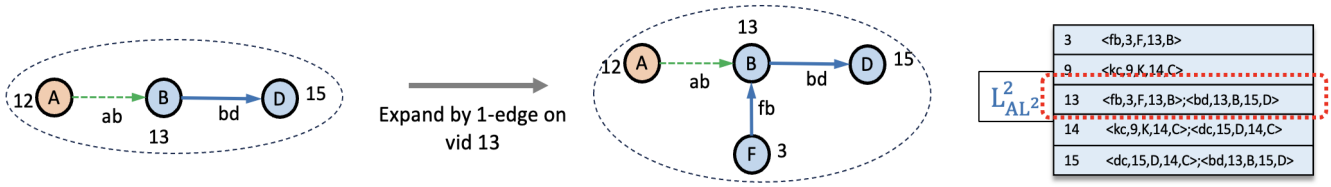


Figure 4.3: Expansion using $L^2_{AL^2}$ in iteration $k = 2$

3. **Expand k-edge interlayer instance using interlayer adjacency list** $[IL^{1,2}_{AL^k}]$ (Figure 4.4)

   In this step, we expand k-edge interlayer instance (1-edge interlayer edge in 1st iteration) by one edge to form $(k+1)$-edge interlayer instances using interlayer adjacency lists generated in previous $(k-1)$th iteration from top BEAM interlayer instances. Figure 4.4 shows the expansion in 2nd iteration using layer 2 intralayer adjacency list.

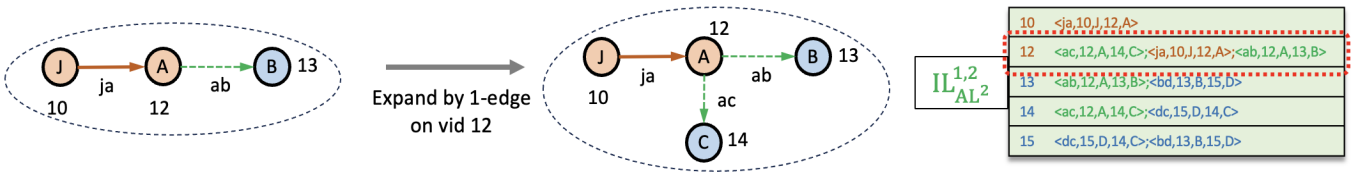


Figure 4.4: Expansion using $IL^{1,2}_{AL^2}$ in iteration $k = 2$

During the composition, duplicates may be generated and can be eliminated by converting each expanded instance into canonical instance. Finally we generate the canonical substructure from canonical instance which will be used for substructure evaluation after composition phase.

Substructure Evaluation: Now we have all the substructures and its instances from layer 1, layer 2 and composition, we use MDL metric evaluation to rank the substructures and top beam substructures will form the candidates to our next iteration. And this process continues till the termination condition is applied.

4.5 Need for updating Adjacency Lists

Adjacency lists changes with the iteration and the specific adjacency lists used in composition for each iteration is described in Table 4.1.

Table 4.1: Adjacency Lists used for Composition

Adjacency List	**Description**
L^1_{AL}	Initially generated L^1 adjacency list for iteration $k = 1$
L^2_{AL}	Initially generated L^2 adjacency list for iteration $k = 1$
$IL^{1,2}_{AL}$	Adjacency list of interlayer edges for iteration $k = 1$
$L^1_{AL^k}$	Adjacency list generated from k–edge instances of L^1 for iteration k where $k = 2, 3, \ldots, S-1$ and S is maximum substructure to obtain
$L^2_{AL^k}$	Adjacency list generated from k–edge instances of L^2 for iteration k where $k = 2, 3, \ldots, S-1$ and S is maximum substructure to obtain
$IL^{1,2}_{AL^k}$	Adjacency list generated from top BEAM k–edge interlayer instances in previous $(k-1)$th iteration for iteration k where $k = 2, 3, \ldots, S-1$ and S is maximum substructure to obtain

- **Updating Intralayer Adjacency Lists [$L^1_{AL^k}$ and $L^2_{AL^k}$]:** In iteration k, k-edge intralayer instances are responsible for composition with interlayer edges to form $(k+1)$-edge interlayer instances. And hence, no need to read original

intralayer adjacency list [L^1_{AL} and L^2_{AL}] in subsequent iterations. Figure 4.5 (a) shows an example of 4-edge composed instances from 3-edge intralayer instances in iteration $k = 3$.

- **Updating Interlayer Adjacency List** [$IL^{1,2}_{AL^k}$]**:** In iteration k, all k-edge substructure instances are subset of $(k+1)$-edge substructure instances [$S^k \subset S^{(k+1)}$], hence no need to use original interlayer adjacency list [$IL^{1,2}_{AL}$]. In layer graph, there may be disconnected components having isolated edges/instances. For these isolated edges/instances in L^1 and L^2, if we don't update the $IL^{1,2}_{AL}$, $(k-1)$-edge information won't be present in the $L^1_{AL^k}$ and $L^2_{AL^k}$ to generate $(k+1)$-edge interlayer instances. Figure 4.5 (b) shows an example of 4-edge composed instances from 3-edge interlayer instances in iteration $k = 3$.

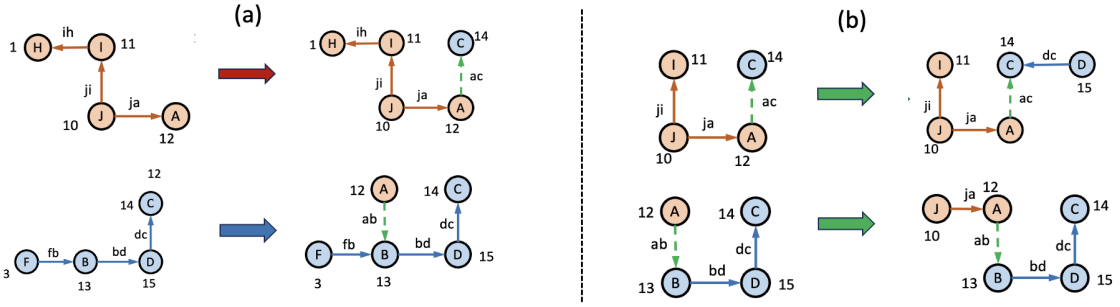


Figure 4.5: Composed Instances in iteration $k = 3$, (a) 4-edge composed instances from 3-edge intralayer instances (b) 4-edge composed instances from 3-edge interlayer instances

4.6 Composition Algorithm

In this we discuss the algorithmic approach of iterative composition algorithm of HeMLN-SD (ICA). As shown in Algorithm 1, the input to our composition algorithm is the list of k-edge composed instances (1-edge interlayer edge list for iteration 1) $IL^{i,j}_k$, adjacency list of composed instances generated in previous iteration (adjacency list of interlayer edges in iteration 1) $IL^{i,j}_{AL^k}$ and adjacency list of i-th and j-th layer

generated from k-edge expanded and unexpanded instances of i-th and j-th layer respectively ($L^i_{AL^k}$ and $L^j_{AL^k}$ respectively). The output generated is the $(k+1)$-edge composed substructures of k-th iteration where k stands for iteration.

For each of the k-edge instance, we expand on all the vertex Ids present in the instance using all the three adjacency lists as depicted from line 2 to 14. We get all the edge list corresponding to the vertex Id from the union of all the three adjacency lists as depicted from line 2 to 4. We expand each edge on vertex Id v and convert to *canonical instance* using *lexicographical order technique*. This helps us to eliminate duplicates as depicted from line 5 to 9. Hence, all the missing substructures are generated across the layers and within interlayer.

Algorithm 1 HeMLN Iterative Composition Algorithm (ICA) of kth iteration for HeMLN-SD

Input: $IL^{i,j}_k$, $IL^{i,j}_{AL^k}$, $L^i_{AL^k}$, $L^j_{AL^k}$
Output: $IL^{i,j}_{k+1}$ ▷ Set of $(k+1)$-edge composed substructures of kth iteration

1: $IL^{i,j}_{k+1} \leftarrow \emptyset$
2: **for** each k-edge instance $ks \in IL^{i,j}_k$ **do**
3: **for** each vertex-id $v \in ks$ **do**
4: $EL_v \leftarrow \{v \in IL^{i,j}_{AL^k} \bigcup L^i_{AL^k} \bigcup L^j_{AL^k} | v.edgelist\}$
5: **for** each edge $e \in EL_v$ **do**
6: **if** $e \notin ks$ **then**
7: $ci \leftarrow merge\ ks$ to e in lexicographical order
8: **if** $ci \notin IL^{i,j}_{k+1}$ **then** ▷ check for duplicates in the result set
9: $IL^{i,j}_{k+1} \leftarrow IL^{i,j}_{k+1} \bigcup \{ci\}$
10: **end if**
11: **end if**
12: **end for**
13: **end for**
14: **end for**

Table 4.2: Table of Notations

Notations	Description
k	Used as subscript for iteration and takes values $1, 2, \ldots, s-1$ where s is size of substructures to be obtained.
i, j	Used for for indicating interlayer -ids, where $i = 1, 2, \ldots, n$ and $j = i+1$ where n is total number of layers in HeMLN.
$IL_k^{i,j}$	Set of composed interlayer substructures of i,j and this changes with iteration. For $k = 1$, it is list of interlayer edges.
$IL_{AL^k}^{i,j}$	Adjacency list of composed substructures of previous iteration and this changes with iteration. For $k = 1$, its adjacency list of interlayer edges.
$L_{AL^k}^{i}$	Adjacency list of expanded substructures of i-th layer and this changes with iteration.
$L_{AL^k}^{j}$	Adjacency list of expanded substructures of j-th layer and this changes with iteration.
ks	Each $ks \in IL_k^{i,j}, ks =< E_l^1, V_{sid}^1, V_{sl}^1, V_{did}^1, V_{dl}^1 >; < E_l^2, V_{sid}^2, V_{sl}^2, V_{did}^2, V_{dl}^2 > ; \ldots; < E_l^k, V_{sid}^k, V_{sl}^k, V_{did}^k, V_{dl}^k >$ as list of 5 tuples where, E_l- edge label, $V_{sid}-$ source vertex-id, $V_{sl}-$ source vertex label, $V_{did}-$ destination vertex-id, $V_{dl}-$ destination vertex label.
ci	Canonical instance, generated after arranging the expanded instance in *lexicographical order*.

4.7 Correctness of the HeICA

We will now prove the correctness of our approach using induction method.

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

Proof:

Our goal is to prove that for any iteration k, HeICA correctly generates all substructure for the next iteration $(k+1)$. Following are the induction steps -

Base Case:

In iteration 1,

1. HeICA generates all L^1 intralayer substructures by using SUBDUE algorithm on L^1 using L^1_{AL} adjacency list
2. HeICA generates all L^2 intralayer substructures by using SUBDUE algorithm on L^2 L^2_{AL} adjacency list
3. Iterative composition algorithm (ICA) generates interlayer substructures using L^1_{AL}, L^2_{AL} and $IL^{1,2}_{AL}$

Step 3 will generate all the 2-edge interlayer substructures which were missed by SUBDUE when only processing L^1 and L^2.

Thus HeICA correctly generates all 2-edge substructure instances in the first iteration *(k = 1)*.

Inductive Step:

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

HeICA will expand composed substructure instances of previous iteration (*k-1*) to generate expanded composed substructure instances for iteration $(k + 1)$ in iteration k. Specifically it uses corresponding intralayer adjacency list and a composed adjacency list to generate composed instances.

1. It generates all intralayer substructures by expanding L^1 substructures using L^1_{AL} adjacency list and L^2 substructures using L^2_{AL} adjacency list.
2. It generates all composed substructures using composed instances from previous iteration and expanding them using $L^1_{AL^k}$, $L^2_{AL^k}$ adjacency lists generated in current iteration k and $L^{1,2}_{AL^k}$ adjacency list generated in previous iteration $(k-1)$

The unrestricted nature of this expansion process ensures that no instances are missed. Consequently, HeICA precisely generates all instances for the subsequent iteration $k + 1$.

Conclusion:

Based on the base case and the inductive step, we can confidently assert that HeICA consistently generates substructures of all sizes in the input data for a given iteration giving complete accuracy. We shall show empirical correctness of this approach in 6 section.

4.8 Resource Utilization

We use range-based partitioning to divide a single layer into different partitions. The utilization of a partitioning technique is of utmost importance due to the potential size constraints of the adjacency list of a single layer, which may exceed the capacity of the main memory.

It is important to note that the number of partitions of a layer is specified as p. Our approach has the capability to effectively utilize all accessible resources. In an ideal situation, it is possible to load and process all p partitions concurrently using separate processors, resulting in maximum parallelism.

Nevertheless, in situations when there is a scarcity of resources, it may not be practical to process all p partitions simultaneously. In instances of this nature, it is possible for numerous partitions to be allocated to a single CPU. Ideally, the objective is to achieve a one-to-one correspondence between the number of partitions and the number of available processors in order to facilitate effective parallel processing.

Another scenario that supports the use of more divisions occurs when there is an uneven distribution of data across the partitions. The use of range-based partitioning, employing fixed ranges, might lead to uneven partition sizes as a consequence of the inherent properties of the graph. Unequal partitions can give rise to load imbalance, wherein specific processors are burdened with a greater workload compared to others, hence leading to sub optimal utilization of resources. In this particular scenario, the

utilization of a greater number of partitions compared to the number of processors can yield advantageous outcomes in terms of enhancing load balancing.

In this chapter we have explained the design of the HeMLN-SD system and also elaborated on the detailed HeMLN iterative composition algorithm (HeICA). Having provided this discussion, in the next chapters we elaborate the implementation aspects of our design in the Map/Reduce framework and present our analysis of the map/reduce approach.

CHAPTER 5

IMPLEMENTATION

This chapter provides detailed description of the implementation of scalable HeMLN-SD approach based on partitioning and parallel processing. HeMLN-SD developed as part of this thesis uses Map/Reduce framework for a distributed processing to utilize parallel processing.

5.1 Layer Graph Representation

Our input graph is represented as a sequence of unordered edges (often known as 1-edge substructures). This enables the distribution of data among different machines. Our approach can have both undirected and directed graphs. However, in our HeMLN-SD system, we have chosen to focus solely on directed graphs due to their explicit representation of relationships through directions. Table 3.1 presents the edge input representation of the graph depicted in Figure **??** of chapter 3. The representation of each edge consists of a tuple with five elements, namely the edge label, the identifier of the source vertex, the label of the source vertex, the identifier of the destination vertex, and the label of the destination vertex.

5.2 Layer Graph partitioning

As previously mentioned in the preceding chapter, the partitioning of our layer graph is undertaken as a means to overcome the limitations of main memory. The partitions of adjacency lists are generated in order to facilitate processing on individual processors. This is achieved through the use of range-based partitioning technique, as

previously explored in the work [20]. In the context of our HeMLN-SD, the map/reduce architecture is employed, and as a result, input splits of the layer graph are generated in order to ensure that each split is processed by a single map task. Note that splits are different from range partitioning of the graph. Splits are used by the Map/reduce framework for distributed processing. Range-based partitions are used for the substructure discovery algorithm. As we shall see, creation of substructures belonging to a range as well adjacency list required for each range are also generated using the ma/reduce job. Routing of substructures to appropriate range-based partitions are all done using map/reduce.

5.3 HeMLN-SD using Map/Reduce Paradigm

The map/reduce architecture is employed in our HeMLN-SD system to harness the capabilities of distributed and parallel computing. Figure 5.1 shows overall map/reduce flow for HeMLN-SD. The first and second map reduce jobs are utilized for performing layer-wise expansion, which is analogous to the graph expansion task outlined in [20]. In our thesis, we conduct independent and parallel analyses to identify all intralayer substructures within layer 1 and layer 2 in job1 and job2, respectively (Analysis phase). The adjacency list is derived from both the expanded and unexpanded substructure instances, and it serves as a valuable input throughout the composition phase. Figure 5.2 and Figure 5.3 shows detailed flow of M/R job1 and job2.

In our composition, we employ adjacency lists to incorporate interlayer edges during the first iteration. Additionally, we leverage the composed adjacency list generated in the previous iteration to generate all missing substructures across the layers. In addition, we generate the substructure within the interlayer by exclusively utilizing interlayer edges in the initial iteration, and composed instances in subse-

quent iterations for expansion purposes. Figure 5.4 shows detailed flow of M/R job 3 (Composition phase).

All the substructures from layer 1, layer 2 and composition are input to the job4 for substructure evaluation. We rank the substructures based on MDL metric and top beam size substructures are carried to next iteration for further expansion. And this process continues till the termination condition is applied. Figure 5.4 shows detailed flow of M/R job 4.

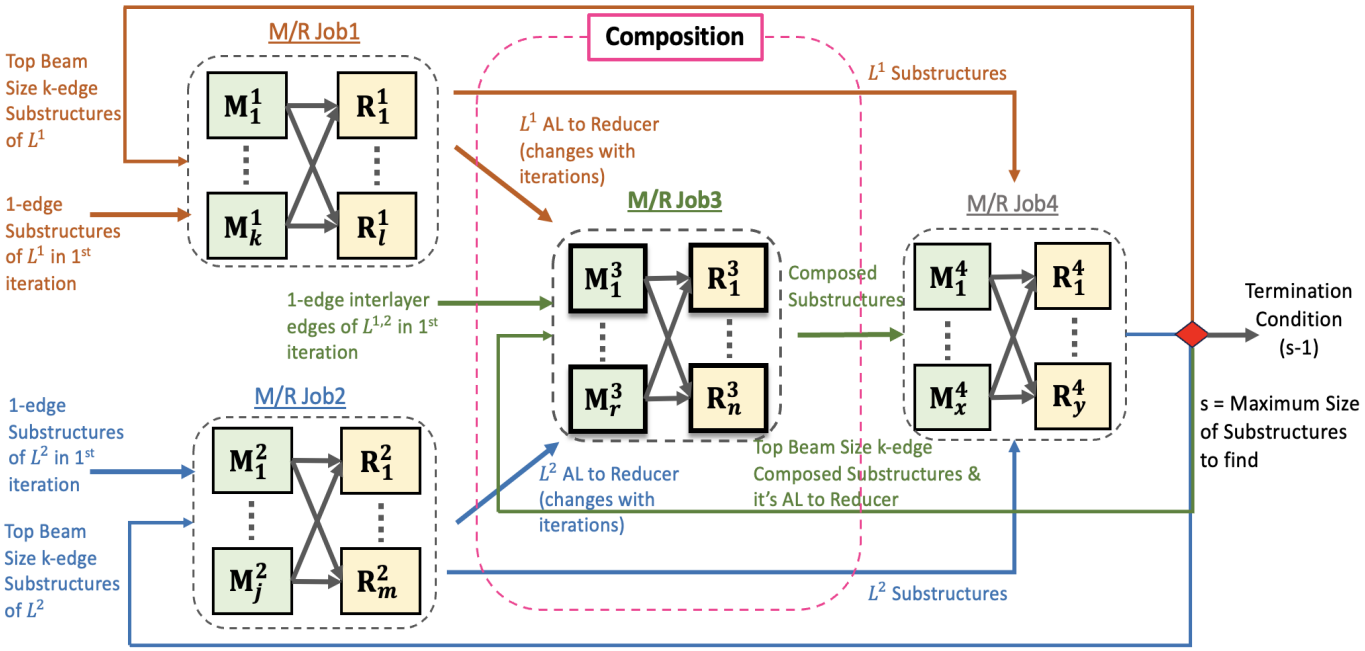
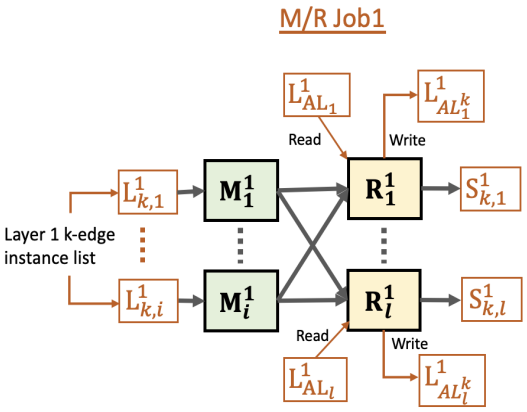


Figure 5.1: Overall Map-Reduce Workflow for kth-iteration



As pre-processing, l partitions $L^1_{AL_{1..l}}$ of adjacency list are generated for layer 1.

Input: k-edge instance list (j input splits - $L^1_{k,1..j}$) of layer 1. Partitions can be arbitrary.
Setup: Load range info of l adjacency list partitions.
Map process: Find partition Id of adjacency list for all vertices of each k-edge instance based on range information.

Emit key: partition Id
Emit value: k-edge instance

Reduce process: For given partition Id,

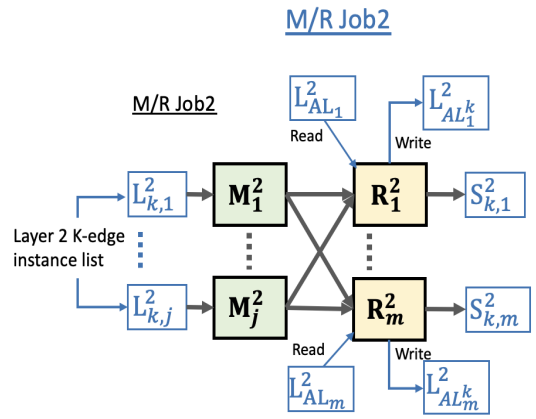
- Read corresponding adjacency list from partitions – $L^1_{AL_{1..l}}$ of layer 1
- Expand each k-edge instance by one edge to form k + 1 edge and convert it to canonical form.
- Remove duplicates.
- Generate canonical substructure.
- Create adjacency list from expanded instances and unexpanded instances (partitions -$L^1_{AL^k_{1...l}}$).

Emit key: Canonical Substructure
Emit value: Canonical Instance (partitions – $S^1_{k,1..l}$)

Cleanup:
Write (partitions -$L^1_{AL^k_{1...l}}$) adjacency list to HDFS file.

k = iteration
i = input splits and max. map tasks <= i
l = partitions

Figure 5.2: Map-Reduce Job1 - Layer 1 Expansion for kth-iteration



As pre-processing, m partitions $L^2_{AL_{1..m}}$ of adjacency list are generated for layer 2.

Input: k-edge instance list (j input splits – $L^2_{k,1..j}$) of layer 2. Partitions can be arbitrary.
Setup: Load range info of m adjacency list partitions.
Map process: Find partition Id of adjacency list for all vertices of each k-edge instance based on range information.

Emit key: partition Id
Emit value: k-edge instance

Reduce process: For given partition Id,

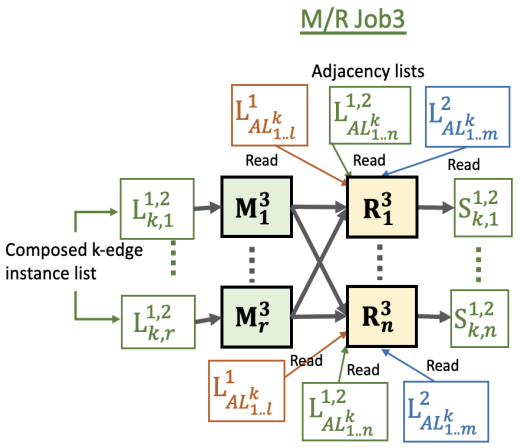
- Read corresponding adjacency list from partitions – $L^2_{AL_{1..m}}$ of layer 2
- Expand each k-edge instance by one edge to form k + 1 edge and convert it to canonical form.
- Remove duplicates.
- Generate canonical substructure.
- Create adjacency list from expanded instances and unexpanded instances (partitions - $L^2_{AL^k_{1...m}}$)

Emit key: Canonical Substructure
Emit value: Canonical Instance (partitions – $S^2_{k,1..m}$)

Cleanup:
Write (partitions - $L^2_{AL^k_{1...m}}$) adjacency list to HDFS file.

k = iteration
j = input splits and max. map tasks <= j
m = partitions

Figure 5.3: Map-Reduce Job2 - Layer 2 Expansion for kth-iteration



As pre-processing, n partitions $L^{1,2}_{AL^k_{1...n}}$ of adjacency list are generated for interlayer edges. This adjacency list changes with iteration.

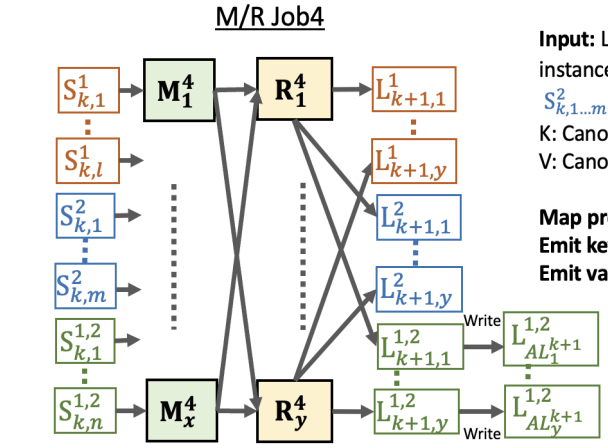
Input: k – edge composed instance list (r input splits – $L^{1,2}_{k,1...r}$) (1-edge interlayer edge list in 1st iteration)
Setup: Load range info of n adjacency list partitions (iteration 1).
Map process: Find partition Id for all vertices of each K-edge instance based on range information.
Emit key: partition Id
Emit value: k-edge instance

Reduce process: For given partition Id,

- Read composed adjacency list created in M/R Job 4 of previous iteration from partitions – $L^{1,2}_{AL^k_{1...n}}$ (For iteration 1, it is n interlayer adjacency list partitions, each partition is read in each reducer tasks)
- Read adjacency list created from instances in job1 and job2 (partitions - $L^1_{AL^k_{1..l}}$ and $L^2_{AL^k_{1..m}}$).

Expand each k-edge instance by one edge to form k + 1 edge canonical instance
Generate corresponding canonical substructure.
Remove duplicates.
Emit key: Canonical Substructure
Emit value: Canonical Instance ($partitions$ – $S^{1,2}_{k,1...n}$)

Figure 5.4: Map-Reduce Job3 - Composition phase for kth-iteration



Input: Layer 1, Layer 2 and combined instances (input splits - $S^1_{k,1...l}$, $S^2_{k,1...m}$ and $S^{1,2}_{k,1...n}$)
K: Canonical Substructure
V: Canonical Instance

Map process:
Emit key: Canonical substructure
Emit value: Isomorphic instances

Reduce process:
Count all isomorphs. Apply MDL metric to rank the substructures limited to BEAM size.
Emit key: Null
Emit value: Canonical instances to be carried in next iteration
Cleanup:

- Write canonical instances of top BEAM size substructures (partitions - $L^1_{k+1,1...y}$ and $L^2_{k+1,1...y}$) to corresponding layers and combined instances to job3 (partitions -$L^{1,2}_{k+1,1...y}$) in the next iteration (k+1).
- Also generate the adjacency list of composed instances to be taken to next iteration$L^{1,2}_{AL^{k+1}_{1...y}}$

Figure 5.5: Map-Reduce Job4 - Substructure Evaluation for kth-iteration

5.4 Algorithmic Approach

5.4.1 Layer Expansion

Algorithm 2 details our layer wise expansion routine in the reducer. Each k-edge instance is read as Mapper input (one substructure instance at a time), while range information of adjacency list partition is loaded in setup function of mapper and kept in memory (line 2). For Mapper, the input key is the line no. and the value is k-edge instance. For each of the vertex in k-edge instance, we add corresponding partition Id (where that vertex Id lies in the range) into the set of partition Ids (line 3 to 6). We also append the layer Id for each k-edge instance which will be useful to separate the layer wise instances after substructure evaluation (line 8). For each of partition Id we emit the key as partition Id and value as k-edge instance.

Here, we don't need combiner as, partition Ids are unique and each of the value emitted is also unique corresponding to partition Id and hence duplicates are generated.

In reducer, we load the adjacency list partition from HDFS based on key. And for each of the k-edge instance we generate $(k + 1)$-edge instance. To remove the duplicates we convert the expanded instance to canonical instance. We maintain a set for removing duplicates. If the canonical instance not present in the duplicates set we convert the canonical instance to canonical substructure for substructure evaluation and emit the key as canonical substructure and value as canonical instance. Line 5 to 15 depicts all these steps. In clean up phase of reducer, we create the adjacency list from expanded and unexpanded instances (i.e. from k-edge intralyer instances) and write to HDFS (line 16 to 19).

5.4.2 **Composition Phase**

Algorithm 3 details our composition phase implemented as job 3. Each k-edge instance is read as Mapper input (one substructure instance at a time), while range information of adjacency list partitions is loaded in setup function of mapper and kept in memory (line 2). For Mapper, the input key is the line no. and the value is k-edge instance. For each of the vertex in k-edge instance, we add corresponding partition Id (where that vertex Id lies in the range) into the set of partition Ids (line 3 to 6). We also append the layer Id for each k-edge instance which will be useful to separate the layer wise instances after substructure evaluation (line 8). For each of partition Id we emit the key as partition Id and value as k-edge instance. From iteration $k = 2$, though we read all partitions in each reducer, we use the range information of adjacency list of interlayer edges in order to just partition the data to reducer.

In the reducer phase, the adjacency list partition is loaded from the Hadoop Distributed File System (HDFS) depending on the key which is partition Id. In the 1st iteration, the interlayer adjacency list is loaded based on the partition key. In subsequent iterations, all the constructed adjacency list partitions formed in job4 of the previous iteration are read. In addition, the adjacency list that is generated from the expanded and unexpanded instances (i.e from k-edge intralayer instances) is loaded. We will merge all the adjacency lists into one in memory. For every instance with k edges, we create an instance with $(k+1)$ edges by utilizing the merged adjacency list. In order to eliminate duplicate elements, it is necessary to transform the expanded instance into a canonical instance. We employ a set collection for the purpose of eliminating duplicate elements. In the event that the canonical instance is not found inside the set of duplicates, we proceed to transform the canonical instance into a canonical substructure for the purpose of substructure evaluation. The resulting key

is emitted as the canonical substructure, while the value is emitted as the canonical instance. The sequence of steps is illustrated in lines 21 to 37.

5.4.3 **Substructure Evaluation**

Algorithm 4 is our forth Map/Reduce job where we evaluate Canonical substructures from the isomorphic instances. Count their frequency and apply a metric to restrict future expansion in the Reducer.

We have already generated canonical substructures and its instances in job1, job2 and job3. In Mapper we just route these substructures and its instances. We emit key as canonical substructure and value as canonical instance (line 2). Here we don't use the combiner as the duplicates generated in job1 and job2 are removed locally in their respective reducers of their jobs. Some duplicates are generated in composition phase as we use interlayer adjacency list $[IL^{1,2}_{AL^k}]$ generated in previous iteration. There are chances to generate the intralayer and interlayer instances as $IL^{1,2}_{AL^k}$ will have both intralayer and interlayer edges and these duplicates can be removed in reducer of job4 itself.

Frequency Counting by the Reducer: The reducer is responsible for receiving instances from the mappers, which are categorized based on their canonical substructure. In this, we will employ the concept of a "beam" to effectively store the best substructure instances. Specifically, on line 40, a beam of size B is allocated as a hash map. This beam will be responsible for storing MDL values together with their corresponding instances. Lines 41-46 involve the process of removing duplicates, counting and identifying instances that possess the highest beam Minimum Description Length (MDL) values. This is done in order to limit the future expansion to the best substructures in the subsequent iteration. In clean up phase, we segregate the instances to their corresponding layers and emit the value as canonical instance to correspond-

ing layer partition. We also write the adjacency list of composed instances to HDFS to be used in the next iteration for composition.

And all these steps continue till the termination condition is applied.

5.5 Configuration Parameters

The HeMLN-SD system accepts parameters for different tasks from a configuration file. We have provided options for various parameters such as beam size, metric, maximum substructures to generate, number of substructures to output, layer and interlayer input path in HDFS file system, initial adjacency list of layers, number of mappers and reducers for each job and so on. In the case where certain parameters (related to hadoop configurations) are absent in the configuration specification, the system uses default values for the same. Listed below are the most important parameters of configuartion file that we process to our HeMLN-SD system -

1. Input Layer Graph File: It specifies the path in HDFS where the input graphs of intralayers and interlayer in MR format is loaded.
2. Input Adjacency List Files: It specifies the path in HDFS where the initial adjacency list partitions of intralayer and interlayer edges are loaded.
3. Beam Size: It indicates beam size for substructure evaluation using MDL metric and the value represents the number of top substructures carried to the next iteration. Its default value is 4.
4. Range Information: It specifies the vertex ids range for each partition. This is provided in form of mapping between partition id and vertex start and end range (inclusive).
5. Layer Details: It specifies 4 elements - minimum vertex Id, maximum vertex Id, total nodes and total edges present in corresponding intralayer and interlayer.

6. Metric: This indicates which metric to use for substructure evaluation. We use "MDL" as default metric and the only one which is handled in our HeMLN-SD algorithm.
7. Number of Mappers: This parameter states how many mappers are to be used for each Map/Reduce job.
8. Number of Reducers: This parameter states how many reducers are to be used for each Map/Reduce job.
9. Maximum Size Substructure: This specifies the maximum size of the substructure to be obtained. Basically, one less than this value indicates number of iterations till which we run our algorithm.
10. Number of substructures to output: This indicates how many substructures to store in result set.

Following is the sample input parameter file used in our implementation of HeMLN-SD system for 50KV100KE dataset -

```
// Graph: 50KV100KE.graph, HeMLN, random partitioning,
// No. of layers: 2, Distribution ratio: [0.7, 0.3], disconnected layers
// base name to be used to construct other parameters and file names
BASE_NAME = "50KV100KE_HeMLN"
NO_OF_LAYERS = 2  // no. of layers
// minVid,maxVid,total_nodes,total_edges (added by graph-to-layers.py)
DETAILS_OF_50KV100KE_HeMLN_L1 = 1,50000,35000,49250
DETAILS_OF_50KV100KE_HeMLN_L2 = 6,49996,15000,8880
DETAILS_OF_50KV100KE_HeMLN_IL_L1L2 = 1,50000,46692,41870
// 4 ranges written by partitioned-adj-lists.py
RANGE_INFO_50KV100KE_HeMLN_L2 = 6-12503,12504-25001,25002-37499,37500-49996
RANGE_INFO_50KV100KE_HeMLN_IL_L1L2 = 1-12500,12501-25000,25001-37500,37501-50000
```

```
RANGE_INFO_50KV100KE_HeMLN_L1 = 1-12500,12501-25000,25001-37500,37501-50000
// below are manually added parameters
USER = "kiranbolaj"
BASE_DIRECTORY = "/user/"
BEAM_SIZE = 4    // beam size
// "FREQUENCY", "MINSUP", "MDL" - same as SIZE evaluation in SUBDUE
METRIC = "MDL"
NO_OF_MAP_TASKS_JOB1 = 4
NO_OF_MAP_TASKS_JOB2 = 4
NO_OF_MAP_TASKS_JOB3 = 4
NO_OF_MAP_TASKS_JOB4 = 4
NO_OF_REDUCE_TASKS_JOB1 = 4
NO_OF_REDUCE_TASKS_JOB2 = 4
NO_OF_REDUCE_TASKS_JOB3 = 4
NO_OF_REDUCE_TASKS_JOB4 = 4
MAX_SIZE_SUBSTRUCTURE = 6   // Max size of substructures generated
N_SUBSTRUCTURES_TO_OUTPUT = 10    // Best substructures to print
MR_LAYERS_INPUT_PATH = "/layers_mr_input"
MR_INTERLAYERS_INPUT_PATH = "/interlayers_mr_input"
LAYERS_ADJACENCY_LIST_PATH = "/layers_adjacency_list"
INTERLAYERS_LAYERS_ADJACENCY_LIST_PATH = "/interlayers_adjacency_list"
//renamed after job completion later based on BASE_NAME, NO_OF_LAYERS, etc.
TIME_ANALYSIS_FILE = "analysis.csv"
SUPPORT = 0    // in case of MINSUP
```

5.6 Implementing Analysis

Implementing Job Counters:

In order to evaluate the performance of our Map/Reduce jobs. We have utilized the built-in counters and incorporated a few custom counters (user-defined) in our Mapper and Reducer program. Hadoop Map/Reduce counters are a valuable channel for collecting statistics pertaining to the Map/Reduce operation, serving purposes such as cost and space analysis and quality control. A list of all utilized job counters appears below.

- **Setup time**: The inclusion of a counter to track setup time facilitates the analysis of the setup cost associated with a Mapper/Reducer. In the Map/Reduce task conducted for HeMLN-SD, the setup time cost is deemed insignificant due to the absence of any computational operations being executed. Here, the global parameters utilized in the mapper and reducer functions are initialized.
- **Map time**: The mapper calls the map method for every key/value pair. The map time for a mapper refers to the time taken to process all the assigned key/-value pairs. The time taken for mapping in our HeICA algorithm exhibits variation across different jobs. The map time refers to the time taken to determine the partition Id of the substructure instance in job1 through job3. Regarding job4, the mapping process involves the routing of all the substructures (both interlayer and intralayer) to their respective partitions.

  We compute the map time by finding the maximum value among the map times of all mappers. This enables the measurement of the total time dedicated to the mapping step. In our algorithm, it has been observed that by increasing the number of mappers while keeping the graph size constant, there is a noticeable decrease in the time required for mapping. This reduction in map time can be

attributed to the fact that each mapper is assigned a smaller portion of the data to analyze.

- **Reduce time**: As reduce method in our algorithm carries partial answers to appropriate partition, this counter details the time taken in reducer for expansion, reading and writing the adjacency lists (job1 through job3), substructure evaluation (job4). The reduce time here is significantly high in comparison to map time because most of the computation is performed in the reducer.
- **Cleanup time**: The reducer clean up time has significant important than the mapper clean up time which is negligible as no work is done in mapper clean up time. Reducer clean up time corresponds to the time taken to write the adjacency lists and writing the substructures to corresponding layers.
- **I/O time**: This time determines the total I/O cost for reading and writing the HDFS files.
- **Duplicates removed by reducer**: This counter tells us how many duplicates are removed by the reducer. And all the duplicates are removed within the partition and not across the partitions.
- **Substructures written**: This counter determines total substructures written to all the reducer partitions. This gives important information about how many substructure instances are in the beam for that iteration. This counter aids in figuring out how big the beam substructure set should be for every iteration.
- **Read and write time of the adjacency lists**: These are the time taken reading and writing adjacency lists from and to the HDFS file system in the reducers of all the jobs.
- **Adjacency list file size**: As we write the adjacency lists in job1, job2 and job4 we want to determine the size of the file that will be read in the reducer.

Controlling the number of Mappers and Reducers: In the context of Map/Reduce model, response time is optimized by controlling mappers and reducers. We control the mappers by generating input splits. The Map/Reduce system divides the entire data into splits and are handled on different machines in a cluster. The default input split size is 128 MB which can be altered. So, for instance let us say we have a cluster of 4 Machines which can be used in parallel and you have 1 GB of input data. If you go by the default input split size provided by Map/Reduce it would divide your data into 2 partitions which will be processed in parallel on 2 Machines. But if you want to utilize all of your resources (4 Machines) to decrease your response time, then we can change the input split size to 256 MB, and it would create 4 partitions (4 Map tasks) which can be processed in parallel and it will decrease the response time. We used this approach in our implementation to control the Map tasks and eventually the mappers. We control the reducers based on adjacency list partitions. For our experimentation, we use same number of reducers as that of number of adjacency list partitions (for first 3 jobs). Since we want to make consistent, we keep number of mappers and reducers same for all the jobs.

On Expanse, each compute node has maximum of 128 processors, and each task will be running on single processor. So if we want 16 map/reduce tasks we specify 16. If less than 16, then all the processors will not be utilized. So we can alter the size of split size as discussed above to match the number of processors.

In the next chapter, we discuss the detailed experimental analysis by evaluating the correctness, scalability and performance on large synthetic and real-world datasets using different partitions. We also examine the impact of distributions on different data sizes.

Algorithm 2 First/Second Map/Reduce Jobs for Layer Expansion (L^1 and L^2) for k-th iteration

INPUT: k-edge substructure instances of i-th layer ($i = 1, 2$)
OUTPUT: $(k+1)$-edge substructures of i-th layer
Class Mapper

function SETUP
    R = Load the range info of adjacency list partitions of i-th layer
end function
function MAP(key = line no, $value$ = k-edge 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,R)
    end for
    for each partition–id p in PIds **do**
        $value$ = append layer Id L^i to $value$
        emit(key = p, $value$ = $value$)
    end for
end function

Class Reducer

function REDUCE(key = partition-id, $values$ = list of k-edge instances)
    Load partition key from HDFS
    create an empty set of *duplicateSet*
    for each k-edge canonical instance ks in $values$ **do**
        ci = expand ks to $(k+1)$-edge canonical instance
        // Remove duplicates
        if $ci \notin duplicatesSet$ **then**
            cs = convert $(k+1)$-canonical instance to canonical substructure
            emit(key = $(k+1)$-canonical substructure, $value$ = $(k+1)$-canonical instance)
        end if
    end for
end function
function CLEANUP
    Create adjacency list from $(k+1)$-edge canonical instance
    Write the adjacency list to HDFS
end function

Algorithm 3 Third Map/Reduce Job for Composition for k-th iteration

INPUT: k-edge composed instances (1-edge interlayer edges in 1st iteration)
OUTPUT: $(k+1)$-edge composed substructures
Class Mapper

function SETUP
    R = Load the range info of adjacency list partitions of interlayer edges
end function
function MAP(key = line no, $value$ = k-edge 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,R)
    end for
    for each partition–id p in PIds **do**
        $value$ = append layer Id $L^{1,2}$ to $value$
        emit(key = p, $value$ = $value$)
    end for
end function

Class Reducer

function REDUCE(key = partition-id, $values$ = list of k-edge instances)
    if $k = 1$ **then**
        Load partition key from HDFS
    else
        Load all composed adjacency list partitions from HDFS
    end if
    Load all L^1 and L^2 adjacency list partitions generated in M/R job1 and job2 from HDFS
    merge all the adajcency list to single adajcency list
    create an empty set of $duplicateSet$
    for each k-edge canonical instance ks in $values$ **do**
        ci = expand ks to $(k+1)$-edge canonical instance using all adjacency lists
        // Remove duplicates
        if $ci \notin duplicatesSet$ **then**
            cs = convert $(k+1)$-canonical instance to canonical substructure
            emit(key = $(k+1)$-canonical substructure, $value$ = $(k+1)$-canonical instance)
        end if
    end for
end function

Algorithm 4 Forth Map/Reduce Job for Substructure Evaluation for k-th iteration

INPUT: $(k+1)$-edge composed substructures and its instances
OUTPUT: Top $(k+1)$-edge Beam size substructures taken to next iteration **Class Mapper**

1: **function** MAP(key = canonical substructure, $value$ = canonical instance)
2: emit($key = key$, $value = value$)
3: **end function**

Class Reducer

38: **function** REDUCE(key = partition-id, $values$ = list of k-edge instances)
39: create Set isoSet to store isomorphs
40: create a local beamMap to store MDL as key and instances as value
41: **for** each canonical k-edge instance ks in $values$ **do**
42: add ks to $isoSet$ //remove duplicates
43: **end for**
44: c = count(substructures in $isoSet$)
45: mdl = MDL(c,#vertices and #edges in key)
46: update beamMap with mdl
47: **end function**
48: **function** CLEANUP
49: Create an empty composed adjacency list $IL^{1,2}_{AL^{(k+1)}}$
50: **for** each instance ks in beamMap **do**
51: **if** ks has single layer Id **then**
52: emit(key = null, $value = ks$) to that layer Id partition.
53: **else**
54: emit(key = null, $value = ks$) to composed layer Id partition.
55: add all the vertex Ids and corresponding edges in $IL^{1,2}_{AL^{(k+1)}}$
56: **end if**
57: **end for**
58: Write $IL^{1,2}_{AL^{(k+1)}}$ to HDFS
59: **end function**

CHAPTER 6

EXPERIMENTAL ANALYSIS

In this chapter we will discuss the results and analysis of various experiments performed on diverse range of synthetic and real-world graphs. Our objective is to examine the effects of MLN graph characteristics, such as graph sizes, layer distributions, the influence of partitions, and the scalability of our HeMLN-SD algorithm.

6.1 Experimental Environment

All experiments are performed on the Expanse cluster located at SDSC (San Diego Supercomputer Center). The Expanse cluster is organized into 13 SDSC Scalable Compute Units (SSCUs), comprising 728 standard nodes, 54 GPU nodes, and 4 large-memory nodes. For tasks involving Java with Hadoop MapReduce, we utilized the standard nodes. These standard compute nodes on Expanse are equipped with dual 64-core AMD EPYC 7742 processors and possess 256 GB of DDR4 memory. Additionally, each compute node has access to a 12 PB parallel file system and is equipped with a 1 TB SSD for local scratch space. Job scheduling on Expanse is managed through the SLURM workload manager. The configuration details for each compute node are outlined in Table 6.1.

6.2 Dataset Generation

In our HeMLN-SD algorithm, we represent a single graph as a Heterogeneous Multilayer Network (HeMLN), where each layer is treated as an independent graph. Consequently, we have the capability to generate a HeMLN from a single 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

taking into account various graph characteristics such as the number of nodes in each layer. Also, for testing the empirical correctness of our algorithm, we can embed substructures of different size (5 and 10) with known frequency. This ensures that the same substructures are obtained whether the dataset is processed as a single graph or as a HeMLN graph. As a result, our data generation process entails initially generating a single graph and subsequently generating a HeMLN from that single graph.

6.2.1 Graph Generation

This section provides an overview of the graph generator used for our dataset generation. The synthetic graph generator utilized to create the input graphs for this thesis was developed by the AI Lab at the University of Texas at Arlington as discussed in [42]. This graph generator is configurable and accepts various parameters for generating the graph. The following parameters are accepted:

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. (The label can have any alpha numeric characters. Ex. v0, v1, v2, etc.)
    (d) Number of edges
    (e) For each substructure edge
        i. The edge label.(The label can have any alpha numeric characters. Ex. e0, e1, e2, etc.)
        ii. The first vertex ID to which this edge is connected.
        iii. An integer ranging from 0 to (number of substructure vertices - 1)
        iv. The second vertex ID to which this edge is connected.
        v. An integer ranging from 0 to (number of substructure vertices - 1)

Following is an example of input file for graph generator -

```
50KV100KE_embed_5E_3000_15vl30el.g
50000
100000
15
30
1
3000
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 indicated on a separate line. In this example, a graph is generated with 50 thousand vertices and 100 thousand edges. There are 15 distinct vertex labels and 30 distinct edge labels. Within the graph, a substructure of size 5 is embedded with a frequency of 3000. This substructure comprises of 6 vertices (labeled y1, y2, y3, y4, y5, y6) and 5 edges (labeled c1, c2, c3, c4, c5). The resulting graph file is named as **50KV100KE_embed_5E _3000_15vl30el.g** denoting its size 50KV100KE, the embedded substructure size, the frequency (5E and 3000), and the number of distinct vertex labels (15vl) and the edge labels (30el). The generated graph file has **.g** extension. Given input file just shows an example to embed 5-edge substructure. But, we have embedded both 5-edge (as shown in Figure 6.1) and 10-edge size substructures in all our graphs to test the scalability of our algorithm.

6.2.2 Layer Generation

The input to our HeMLN-SD algorithm is in MR (Map/Reduce) format and the single graph generated is in Subgen format. Therefore, we will use a python

script: **graph-to-layers.py** which will partition a single graph to layers and each layer graph generated is in MR format.

The initial step is to create the layers from the single input graph. The script randomly partitions the nodes into the layers based on the distribution ratio and then distributes the corresponding edges connecting those nodes to the layers. If an edge has both the nodes in two different layers, then that edge is written to respective interlayer edge file. Here all the layer files are in MR format. We have used different distributions - 50/50, 70/30 and 90/10 to divide the graph into 2 layers for all our synthetic graph experiments. The script also provides option to connect the layers. To connect the disconnected components within a layer, we leverage the Python package NetworkX [43] to identify all connected components. Subsequently, each connected component is connected to another by introducing an edge between them, marked by a unique edge label. This process is iterated as needed to establish a fully connected layer. The newly added edges are then reintegrated into the single aggregated subgen graph, ensuring the consistency of our Multilayer Network (MLN) with the ground truth. It's worth noting that each connecting edge is assigned a distinct label, preventing it from forming a frequent substructure. This is because any substructure with a connecting edge will always have only one instance.

Below is an example of input file (input_parameters_layer_generation.config) to our graph-to-layers.py script -

```
INPUT_GRAPH_PATH = "50KV100KE.graph"
MLN_OPTION = "hemln"  //"hemln" or "homln"
LAYER_DISTRIBUTION_OPTION = "random"
NO_OF_LAYERS = 2  // no. of layers to generate
DISTRIBUTION_RATIO = "50:50,70:30,90:10"
CONNECTIVITY_OPTION = "disconnected" // "connected" or "disconnected"
```

Below is an example of the output file (50KV100KE_HeMLN_2L_G2L.config) generated by graph-to-layers.py script for 50/50 node distribution which forms input to our partitioned-adj-lists.py script -

```
// Graph: 50KV100KE.graph, HeMLN, random partitioning,
// No. of layers: 2, Distribution ratio: [0.5, 0.5], disconnected layers
// base name to be used to construct other parameters and file names
BASE_NAME = "50KV100KE_HeMLN"
NO_OF_LAYERS = 2  // no. of layers
// minVid,maxVid,total_nodes,total_edges (added by graph-to-layers.py)
DETAILS_OF_50KV100KE_HeMLN_L1 = 2,49999,25000,24727
DETAILS_OF_50KV100KE_HeMLN_L2 = 1,50000,25000,25239
DETAILS_OF_50KV100KE_HeMLN_IL_L1L2 = 1,50000,43014,50034
```

Our next step is to generate the adjacency list and its range-based partitions, based on the required number of partitions. We have a python script: **partitioned-adj-lists.py** to generate these partitions. It reads .config file generated by graph-to-layers.py script and input as the number of adjacency list partitions to generate. Generating the adjacency list for each layer is crucial, as the expansion process in each layer functions independently. While we create the adjacency list in memory, it's worth noting that the Map/Reduce framework can be utilized if the graph size is exceptionally large. The partition script updates the .config file with all the required ranges for each layer including interlayer and other parameters which are crucial parameters to our HeMLN-SD algorithm. An example of a configuration file is shown in the chapter 5 under Configurations Parameters section which will be generated from adjacency list partitioning script.

6.3 Dataset Description

We performed a series of experiments to evaluate the performance, accuracy, speedup by varying the configurations of mappers and reducers, and the scalability

Dataset	Used For	M/R configs (per layer)
Synthetic 50KV_100KE	Accuracy, Response Time	2M/2R, 4M/4R 8M/8R, 16M/16R
Synthetic Large 1MV_4ME and 2.5MV_10ME	Response Time, Scalability	16M/16R, 32M/32R 64M/64R, 128M/128R
Amazon	Response Time, Scalability	8M/8R, 16M/16R 32M/32R, 64M/64R
DBLP and Word-Association	Response Time	2M/2R, 4M/4R 16M/16R, 32M/32R

Table 6.2: Dataset description

of our approach. The datasets used in our experiments are detailed in Table 6.2 and its distributions are detailed in Table 6.3. We included synthetic graphs of diverse sizes featuring multiple embedded substructures with user-defined frequencies. This allows us to cross-verify our results against known frequencies and substructures. Furthermore, our experiments incorporated real-world datasets sourced from Amazon, DBLP and Word-Association. This comprehensive dataset selection ensures a thorough evaluation, encompassing both synthetic datasets with embedded substructures and real-world datasets for a more realistic assessment of our approach.

Dataset	Node Distribution	L^1 #Nodes	L^2 #Nodes	L^1 #Edges	L^2 #Edges	$L^{1,2}$ #Edges
50KV_100KE	50/50	25000	25000	29236	29048	49858
	70/30	35000	15000	49250	8880	41870
	90/10	45000	5000	81203	1040	17757
100KV_500KE	50/50	50000	50000	124719	124837	250444
	70/30	70000	30000	245122	45196	209682
	90/10	90000	10000	404942	4998	90060
400KV_1ME	50/50	200000	200000	250297	249961	499742
	70/30	280000	120000	490686	89987	419327
	90/10	360000	40000	809920	9967	180113
800KV_3ME	50/50	400000	400000	749725	750265	1500010
	70/30	560000	240000	1468707	269877	1261416
	90/10	720000	80000	2429768	29972	540260
1MV_4ME	50/50	500000	500000	998911	1000756	2000333
	70/30	700000	300000	1959976	359709	1680315
	90/10	900000	100000	3239820	40233	719947
2.5MV_10ME	50/50	1250000	1250000	2497921	2498771	5003308
	70/30	1750000	750000	4902860	899571	4197569
	90/10	2250000	250000	8100658	99761	1799581
AMAZON	50/50	367661	367661	653541	652406	1304644
[0.74MV_2.6ME]	70/30	514726	220596	234961	234961	1095233
	90/10	661790	73532	2115827	25559	469205
WORD-ASSOCIATION	50/50	5308	5308	8839	9226	18051
[10.6KV_36KE]	70/30	7431	3185	17947	3075	15094
	90/10	9555	1061	29811	300	6005
DBLP	–	16918	18	2483	18	29984

Table 6.3: Dataset Distributions

6.4 **Empirical Correctness**

We evaluate the empirical correctness to assess the performance of our algorithm, instead of just relying only on theoretical or operational correctness.

To validate the accuracy of our algorithm, we conducted experiments using small synthetic graphs generated by Subgen. Subgen is used to generate synthetic graphs containing predefined embedded substructures. We have generated synthetic

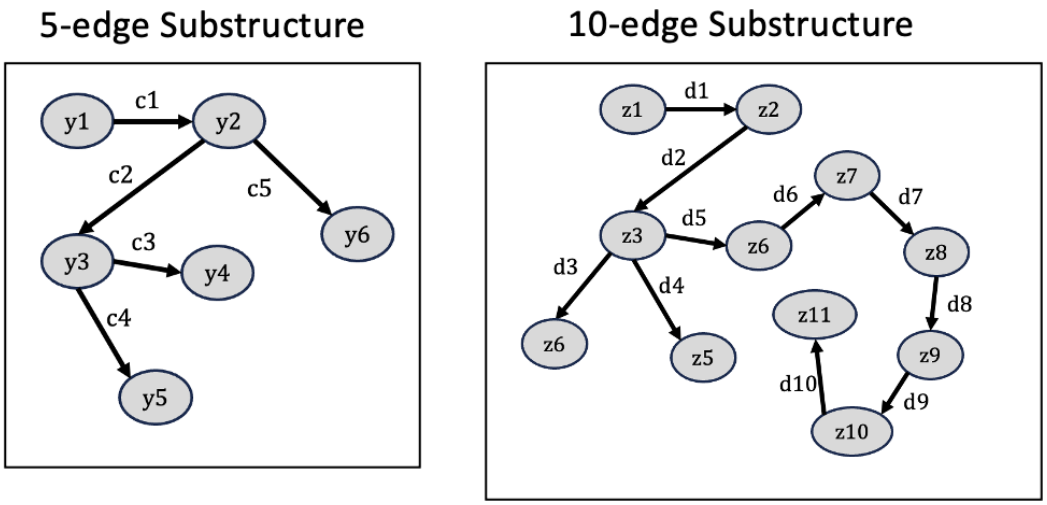


Figure 6.1: Embedded substructures

graphs ranging from 50KV_100KE to 2.5MV_10ME sizes. Both SUBDUE [18] and HeICA consistently identified same substructures when applied to these synthetic graphs. Furthermore, we evaluated the correctness and efficiency of our algorithm by comparing its results with SUBDUE on the same dataset which is converted into HeMLN layers. Since, SUBDUE is a main-memory approach it has many challenges when dealing with large graph sizes beyond 100KV_500KE.

To verify correctness on these synthetic graphs, we have embedded the substructures with a user-defined frequency, aiming to find the same substructures from our algorithm. Figure 6.1 shows the 5-edge and 10-edge substructures that we have embedded in the synthetic datasets.

6.5 **Accuracy**

Throughout our experimental analyses using synthetic datasets, HeICA consistently identifies all instances of the embedded substructure with exact frequencies,

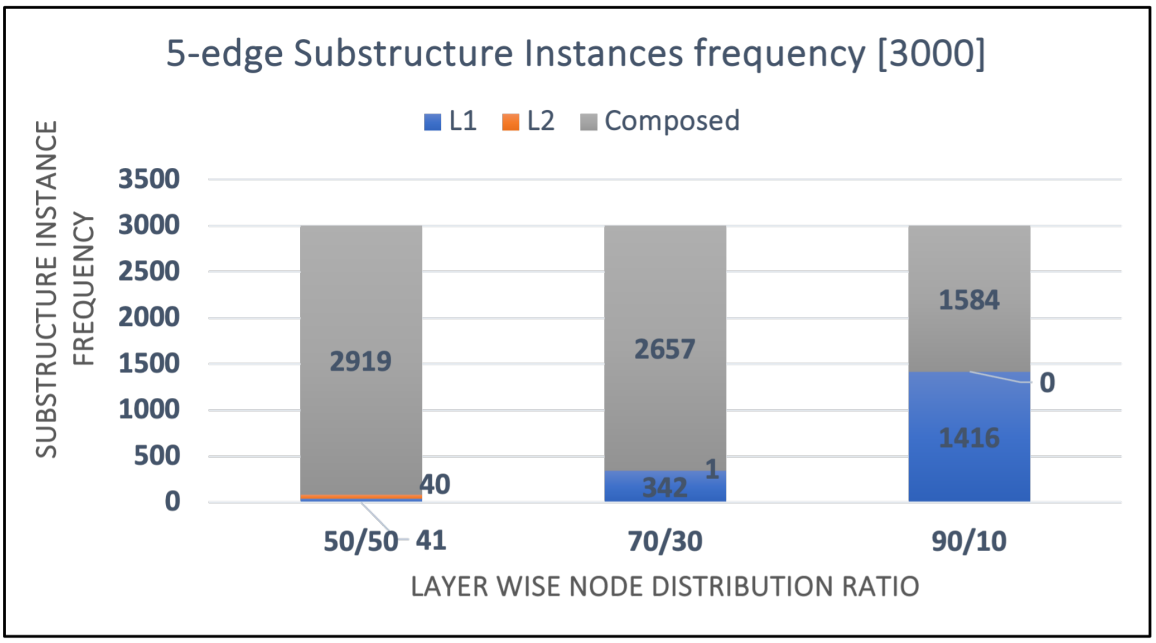


Figure 6.2: Accuracy: 50KV_100KE [5-edge embedded substructure with frequency 3000]

thereby achieving complete accuracy. Now we will discuss how this is verified using different parameters like layer distribution and connectivity.

6.5.1 Effect of Layer Distribution on Accuracy

HeICA consistently achieves complete accuracy irrespective of the layer distribution, demonstrating robustness to variations in distribution ratios, whether they are 50/50, 70/30, or 90/10. As shown in the Figure 6.3, we get the exact frequency for 5-edge substructure embedded for 50KV_100KE graph. Substructures tend to extend more towards a skewed layer (90/10) due to the increased involvement of nodes in layer-wise expansion and composition. In the case of a uniform distribution (50/50), the substructures tend to span across both layers, resulting in the emergence of the most extensive composed substructures.

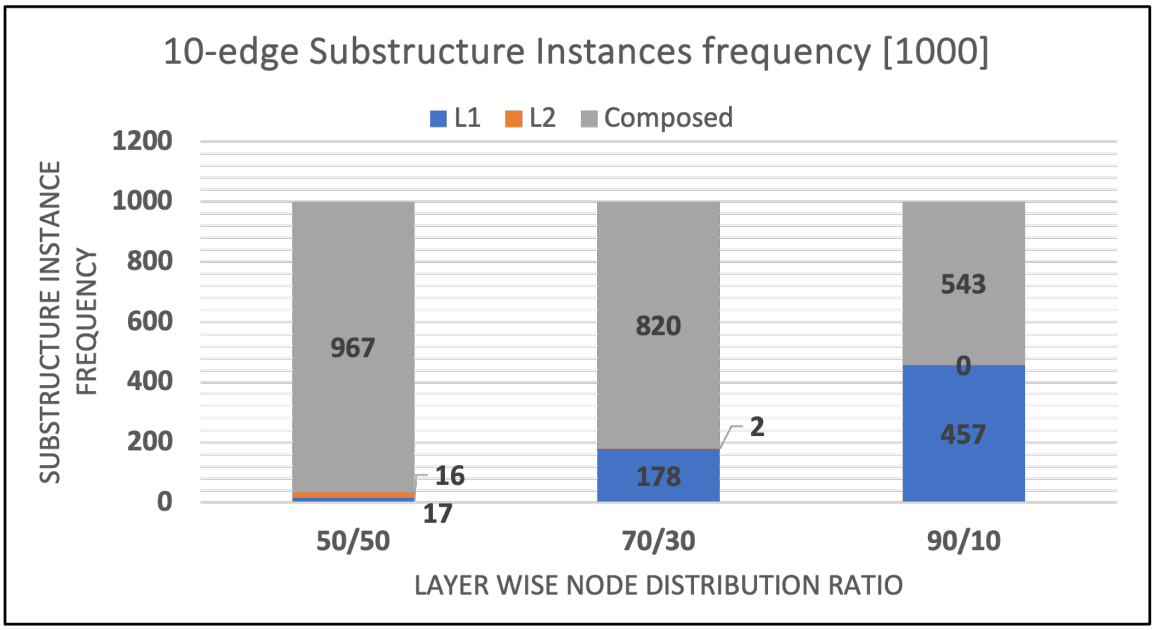


Figure 6.3: Accuracy: 50KV_100KE [10-edge embedded substructure with frequency 1000]

6.5.2 Effect of Layer Connectivity on Accuracy

In order to verify whether the connectivity of layers affects the accuracy, we have performed our experiment on both connected and disconnected layers. Surprisingly, there is no noticeable difference in accuracies emerged between connected and disconnected layers during the experimental analysis. This absence of distinction can be clarified by considering the introduction of a unique connecting edge, which does not constitute a frequent substructure. Due to its distinctive nature, the presence of this connecting edge and the resulting instance in the beam substructures is improbable. Since all these edges have frequency of 1, they always come under lower MDL values. In reality, the only circumstance in which these instances could be part of the beam is when the entire graph is present in the beam. Such a scenario is undesirable, as including the entire graph in the beam is to be avoided. Consequently, the introduction of a connected layer does not have a major impact.

6.6 Response Time and Scalability

To evaluate our response time, we will use different number of partitions alongside an equivalent number of mappers and reducers to maximize parallelization. The goal is to observe a decrease in the time required to complete our experiments as we scale up resources. We seek to comprehend the speedup achievable, exploring whether it adheres to a linear trend or demonstrates diminishing returns over time. Our verification of the speedup and scalability of our approach involves the use of large synthetic and real-world datasets.

6.6.1 Synthetic Graphs

For the analysis of speedup achieved on synthetic datasets, we explored various dataset sizes, ranging from 50,000 vertices and 100,000 edges (50KV_100KV) to 2.5 million vertices and 10 million edges (2.5MV_4ME). Despite variations in layer size, each layer was partitioned into the same number of partitions. We conducted experiments with 2, 4, 8, 16, 32, 64, and 128 partitions, maintaining an equal number of mappers and reducers for each configuration. A consistent trend was observed across all datasets, and for the sake of focus, we delve deeper into the analysis of the largest datasets, specifically 1 million vertices and 4 million edges (1M_4ME) and 2.5 million vertices and 10 million edges (2.5MV_10ME).

As we can observe in Figure 6.4(a), we get an average speedup of 30% by increasing both the number of partitions and the count of mappers/reducers from 16 to 32. Subsequently, a speedup of 22% was noted upon further increasing them from 32 to 64. Finally, a speedup of 13% was observed by further increasing them from 64 to 128. It's important to highlight that the achieved speedup was not linear; doubling the mappers and reducers did not result in halving the time taken. As the number

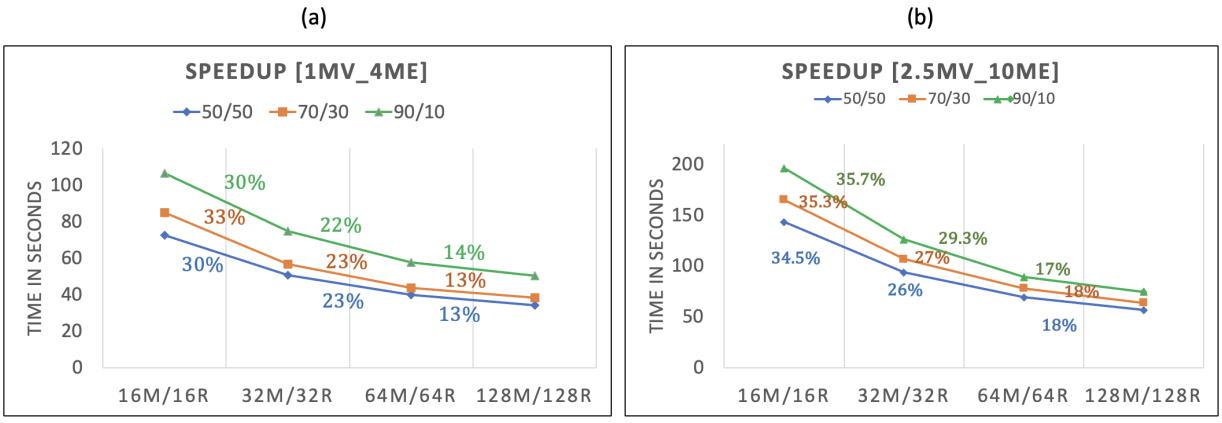


Figure 6.4: Speedup: (a) 1MV_4ME, (b) 2.5MV_10ME

of partitions increased, the speedup exhibited diminishing returns. Similar trend was observed in Figure 6.4(b).

6.6.2 **Real-World Dataset**

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 speedup exhibited no difference when compared to the speedup achieved in synthetic graphs. Figure 6.5 illustrates the response time and the average speedup achieved for varying numbers of partitions and the count of mappers/reducers, ranging from 8 to 16 to 32 to 64.

The similarity in speedup for the layer distribution can be attributed to the fact that the most frequent substructures in the dataset were of size 2. Considering

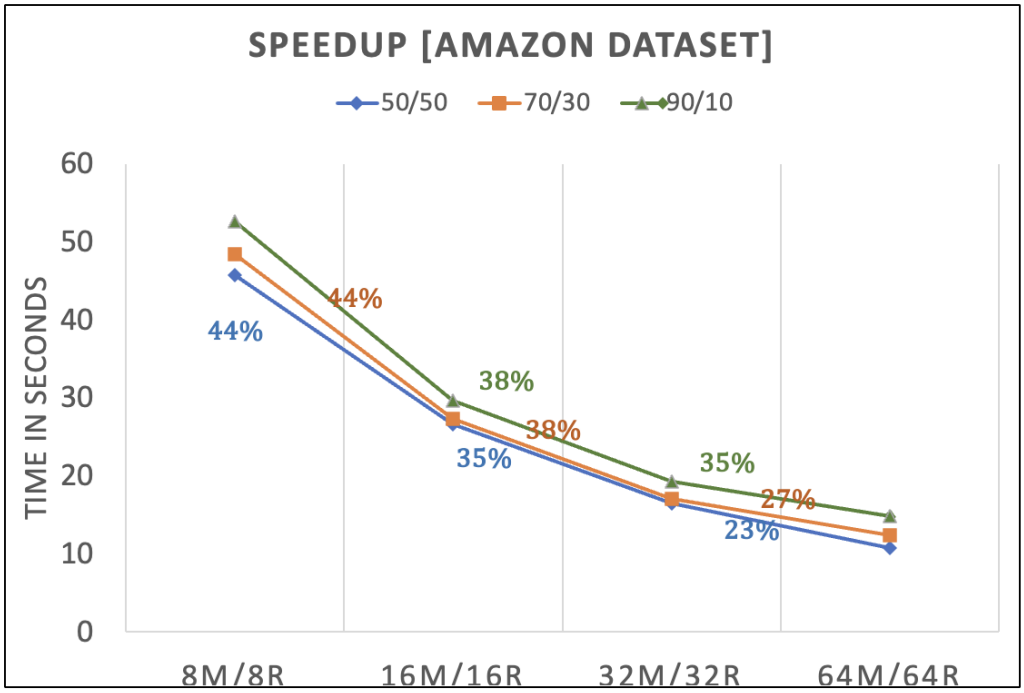


Figure 6.5: Speedup: Amazon Dataset

that real-world datasets may or may not contain frequent substructures, and larger substructures were not embedded in these datasets, our algorithm demonstrate similar performance in capturing substructures of size 2.

Word-Association(10.6KV36KE): Its a graph describing the results of an experiment where the nodes correspond to words and edges represent a cue-target pair as described in [13]. This graph has a maximum degree of 269 and generates 2-edge substructure of varying frequency. As depicted in Figure 6.6, we observe an average speedup of 46% despite variations in the number of mappers and reducers. This is attributed to the fact that the most frequent substructure identified was of size 2. Consequently, the work involved in composition becomes more pronounced, as the maximum degree node leads to increased expansion of edges, subsequently escalat-

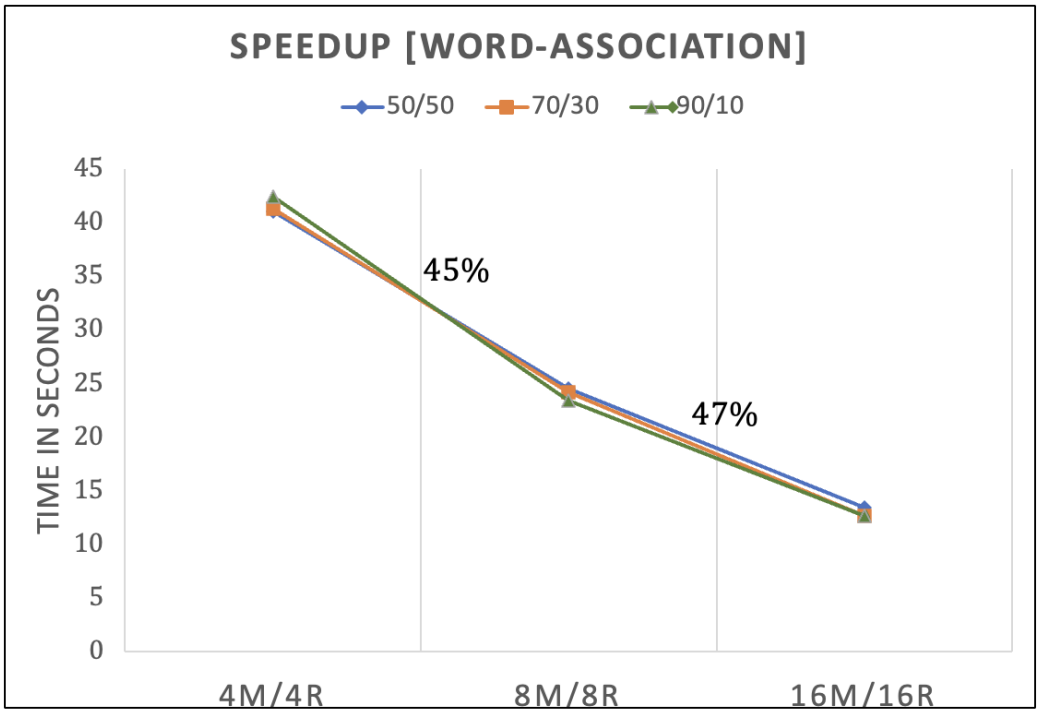


Figure 6.6: Speedup: Word-Association

ing the workload for a single partition reducer task and consequently prolonging the composition time.

DBLP: We are dealing with 2 layers of DBLP dataset - CoAuthors (Nodes: 16918, Edges: 2483) and Years (Nodes and Edges: 18) . Intralayer edges represent the year of publication (29984 edges). This graphs has maximum degree of 756. Again as observed in Figure 6.7, we observe an average speedup of 45% despite variations in the number of mappers and reducers. This is attributed to the fact that the most frequent substructure identified was of size 3. Consequently, the work involved in composition becomes more pronounced, as the maximum degree node leads to increased expansion of edges, subsequently escalating the workload for a single partition reducer task and consequently prolonging the composition time.

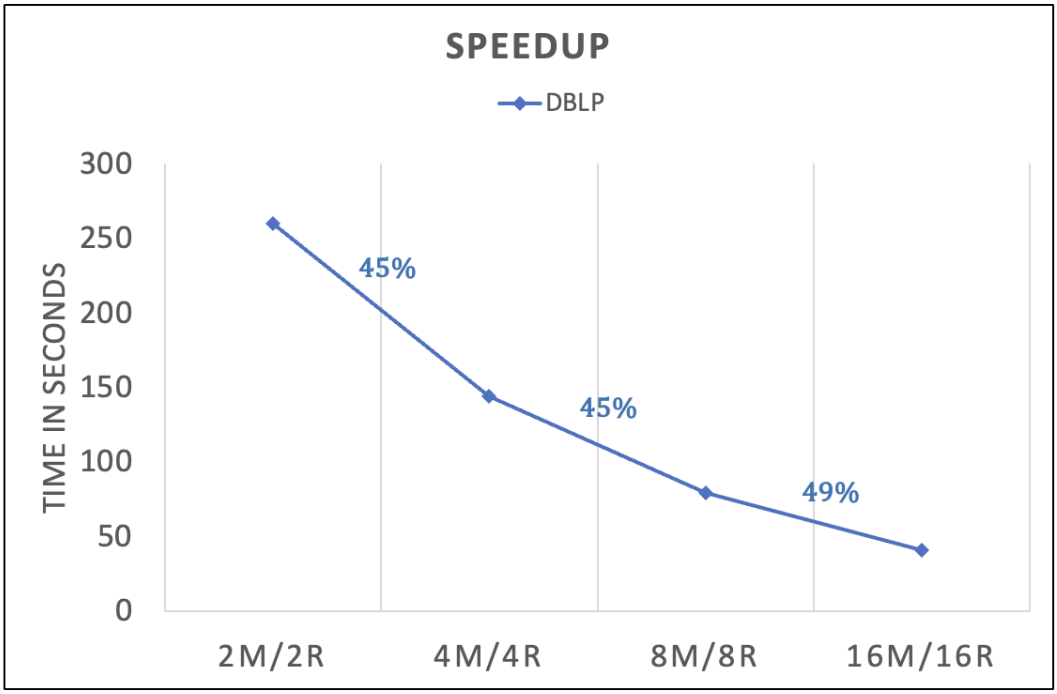


Figure 6.7: Speedup: DBLP

6.7 Response Time for All Experiments Performed

Our goal is to examine the variations in response time across different datasets, each characterized by diverse sizes and features. To provide a comprehensive overview, we have synthesized the results from various experiments into two graphs—one for real-world datasets and the other for synthetic datasets.

In Figure 6.8 and Figure 6.9 show all the experiments executed on real world and synthetic datasets respectively.

In summary, our study involved comprehensive experiments conducted on a diverse set of datasets, emphasizing the differences between the two proposed approaches. The graphs employed in these experiments demonstrated varied characteristics. Through this extensive experimentation, we successfully validated our HeMLN-

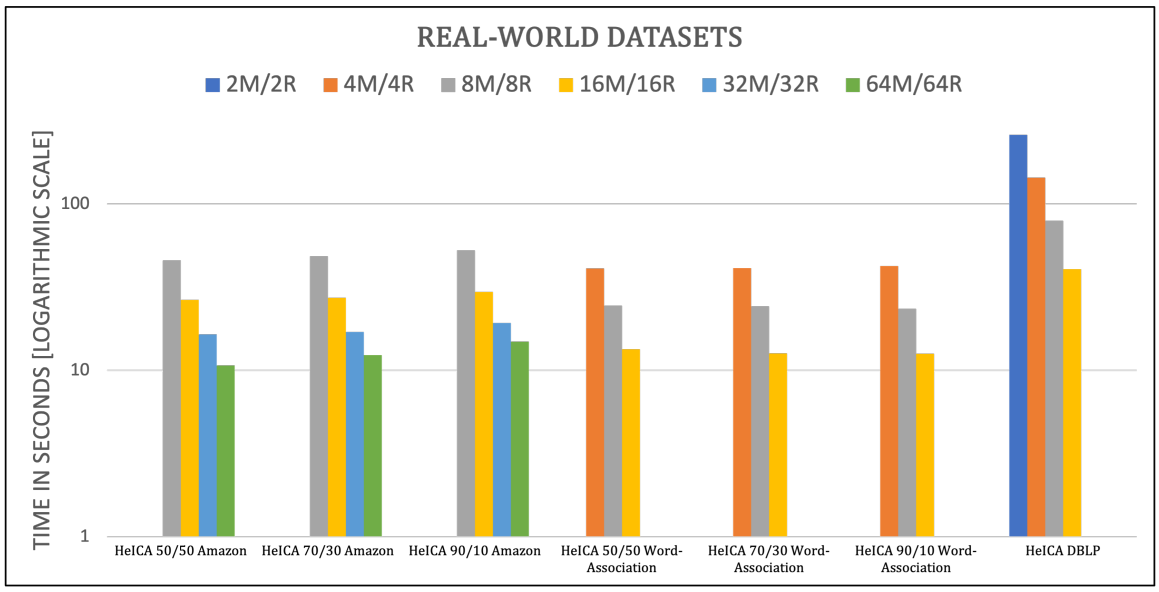


Figure 6.8: Speedup: Real-world datasets

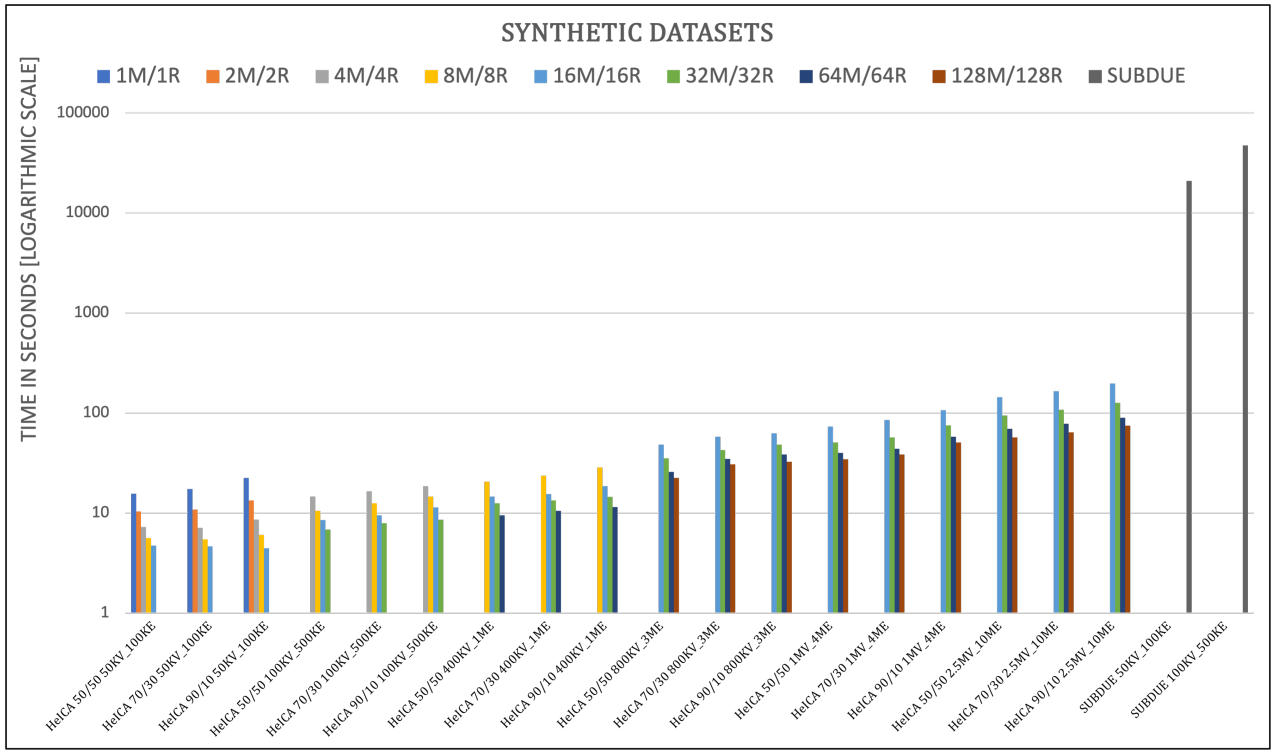


Figure 6.9: Speedup: Synthetic datasets

SD algorithm across a spectrum of graph sizes and diverse graph characteristics. These thorough experiments have robustly confirmed the validity and effectiveness of our proposed approach.

CHAPTER 7

CONCLUSIONS AND FUTURE WORK

This thesis introduced a scalable algorithm for substructure discovery in Heterogeneous Multilayer Networks (HeMLN) using an iterative decoupling-based approach. We designed and implemented a composition algorithm aimed to identify missing substructures across HeMLN layers by leveraging interlayer edges using the Map/Reduce paradigm with a focus on both correctness and efficiency. The algorithm incorporates essential graph mining elements such as sub-graph generation, elimination of duplicates, and counting isomorphic substructures. Furthermore, we validated the correctness of our HeMLN-SD algorithm and conducted extensive experimental analyses on synthetic and real-world datasets.

The field of partitioned graph mining offers interesting opportunities for future research. Currently, our composition algorithm reads all the adjacency lists in each reducer and stores them in memory. To further enhance efficiency and scalability, we can introduce a new map/reduce job before the composition phase to perform range-based partitioning on the adjacency lists which are read as input to the composition. Moreover, our HeMLN-SD algorithm can be extended to include more than two layers. We can also consider alternative approaches to substructure discovery in HeMLN, such as composing substructures at the end of each iteration or composing them at the end of all iterations.

REFERENCES

[1] A. P. F. Miguel E. Coimbra and L. Veiga, "An analysis of the graph processing landscape," *Journal of Big Data*, 2021.

[2] V. S. Boldi P, "The webgraph framework ii: codes for the world-wide web," *2004 data compression conference (DCC 2004), 23–25 March 2004, Snowbird, UT, USA. IEEE Computer Society*, 2004.

[3] S. Brin and L. Page, "The anatomy of a large-scale hypertextual web search engine," *Computer Networks and ISDN Systems*, 1998.

[4] N. G. de Bruijn, "A combinatorial problem," *Proc. Akad. Wet. Amsterdam*, vol. 49, pp. 758–764, 1946.

[5] A. T. Balaban, "Applications of graph theory in chemistry," *Journal of Chemical Information and Computer Sciences*, vol. 25, no. 3, pp. 334–343, 1985. [Online]. Available: https://doi.org/10.1021/ci00047a033

[6] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache flink™: Stream and batch processing in a single engine. ieee data eng. bull. 38, 4 (2015), 28–38," 2015.

[7] Microsoft. (2017) Graph engine (ge): serving big graphs in real-time. Accessed 24 Apr 2020. [Online]. Available: URL_OF_THE_SOURCE

[8] A. Ching, "Scaling apache giraph to a trillion edges," *Facebook Engineering Blog*, p. 25, 2013, https://engineering.fb.com/2013/10/10/data-infrastructure/scaling-apache-giraph-to-a-trillion-edges/.

[9] G. Donnelly. (2020) 75 super-useful facebook statistics for 2018. Accessed 05 May 2020. [Online]. Available: https://www.wordstream.com/blog/ws/2018/02/28/facebook-statistics

[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, Aug. 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, Univ. 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] 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.

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

[20] 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

[21] 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

[22] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[23] B. Memishi, S. Ibrahim, M. Pérez, and G. Antoniu, *Fault Tolerance in MapReduce: A Survey*, 10 2016, pp. 205–240.

[24] L. R. Quinlan and R. L. Rivest, "Inferring decision trees using the minimum description length principle," 1987.

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

[26] 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.

[27] 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.

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

[29] 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.

[30] 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.

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

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

[33] 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.

[34] 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.

[35] 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/

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

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

[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] 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.

[40] 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.

[41] 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

[42] S. Padmanabhan, "HDB-Subdue: A Relational Database Approach to Graph Mining and Hierarchical Reduction." Master's thesis, The University of Texas at Arlington, December 2005.

[43] A. Hagberg, P. J. Swart, and D. A. Schult, "Exploring network structure, dynamics, and function using networkx," 1 2008. [Online]. Available: https://www.osti.gov/biblio/960616

BIOGRAPHICAL STATEMENT

Kiran Bolaj is originally from Belagavi, India. She finished her Bachelor's Degree in Computer Science and Engineering from Visvesvaraya Technological University located at Belagavi, India, in 2016. Kiran worked as a software engineer at GlobalLogic from July 2016 to July 2019 and then at ACL Digital from July 2019 to July 2021. She started her Master's studies in Computer Science at The University of Texas, Arlington, in August 2021 and completed her Master's degree in December 2023. Kiran's research interests include Graph Mining and Big Data Analysis.