MLN-Subdue: Decoupling Approach-Based Substructure Discovery In Multilayer Networks (MLNs)

by

ANISH RAI

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

May 2020

Copyright © by Anish Rai 2020

All Rights Reserved

ACKNOWLEDGEMENTS

I would like to thank my supervising professor Dr. Sharma Chakravarthy for providing me an opportunity to work on this project and constantly motivating me and guiding me throughout this thesis work. Without his constant support and guidance this research work would not have been successful. I would like to thank Dr. Leonidas Fegaras and Mr. David Levine for taking time to serve in my thesis committee.

I would like to extend my gratitude to all the people involved in this project, IT - Lab present members and alumni Harshit Modi, Abhishek Santra, Enamul Karim and Soumyava Das for their constant support and guidance. Also, I would like to thank my family and friends for their constant support and encouragement throughout my academic career.

May 14, 2020

ABSTRACT

MLN-Subdue: Decoupling Approach-Based Substructure Discovery In Multilayer Networks (MLNs)

Anish Rai, M.S.

The University of Texas at Arlington, 2020

Supervising Professor: Dr. Sharma Chakravarthy

Substructure discovery is well-researched for single graphs (both simple and attribute) as it is an important component of knowledge discovery for many applications such as finding the core substructure in a protein, important concept in a large graph, etc. However, multilayer networks or MLNs (instead of attribute graphs) have been shown to be better for modeling complex data sets that have multiple entity and feature types. This model provides more clarity on semantics of data sets as well as the ability to use an arbitrary subset of layers for analysis. However, the challenge is that many algorithms such as community and centrality detection as well as substructure discovery need to be extended to MLN representation.

With the representation of complex data sets as MLNs brings new challenges in terms of finding substructures in MLNs or a subset of MLNs. A naive approach would be to collapse (or aggregate) all (or a subset of) layers into a single attribute graph and use extant algorithms. There are a number of substructure discovery algorithms ranging from memory-based, disk-based, SQL-based, and partitioned using map/reduce framework.

While substructure discovery has been widely used for the analysis of single networks, attribute graphs, and forests, the development of an efficient substructure discovery methods for multilayer networks without aggregation is currently not available. This thesis proposes a new decoupling approach-based substructure discovery algorithm for homogeneous MLNs (or HoMLNs). HoMLNs are MLNs where each layer has the same set of nodes but different connectivity in each layer.

In this dissertation, we propose a decoupling approach-based algorithm for HoMLNs where aggregation is not needed. Further, the algorithm has been implemented using the Map/Reduce framework in order to handle arbitrary number of layers and improving the response time through parallelism. Each layer is processed individually/separately in parallel, but the substructures generated for each layer are *combined after each iteration* to identify substructures across layers (or MLN). The focus is on correctness of the algorithm and resource utilization with respect to number of layers. The proposed algorithm is validated analytically as well through extensive experimental analysis on large real-world and synthetic graphs.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS iv

ABSTRACT v

LIST OF ILLUSTRATIONS x

LIST OF TABLES xii

Chapter Page

1. **INTRODUCTION** 1

  1.1 **Need For Multilayer Networks** 3

  1.2 **Problem Statement** 5

    1.2.1 **Alternatives For Substructures Discovery in a HoMLN** 6

  1.3 **Map/Reduce: A Distributed Framework** 7

    1.3.1 **Substructure discovery using Map/Reduce** 9

  1.4 **Thesis Contributions** 9

  1.5 **Road Map** 10

2. **RELATED WORK** 11

  2.1 **Main Memory Approaches** 11

  2.2 **Disk based approach** 12

  2.3 **Database approach** 13

  2.4 **Partition based approach** 13

  2.5 **Subgraph Mining in a Multilayer Network** 15

3. **Preliminaries** 17

  3.1 **Graphs** 17

    3.1.1 **Input Graph Representation** 17

3.1.2 **Adjacency list** . . . 18
3.2 **Graph expansion** . . . 19
3.2.1 **Canonical Instance** . . . 20
3.2.2 **Graph Isomorphism** . . . 20
3.2.3 **Frequency Calculation** . . . 22
4. **DECOUPLING-BASED APPROACH** . . . 23
4.1 **Decoupling-Based Approach** . . . 23
4.2 **Combining Algorithm** . . . 25
4.2.1 **Challenges for Combining** . . . 28
4.3 **Algorithmic Approach** . . . 29
5. **Implementation Details** . . . 33
5.1 **Layer Generation** . . . 33
5.1.1 **Partitioning Schemes** . . . 34
5.2 **Adjacency List Generator** . . . 35
5.3 **Algorithm in the Map/Reduce Framework** . . . 35
5.4 **Resource Utilization** . . . 43
5.5 **Input Parameters File** . . . 44
5.6 **Controlling the Number of Mappers** . . . 45
5.6.1 **Implementing Job Counters** . . . 46
6. **EXPERIMENTAL ANALYSIS** . . . 48
6.1 **Experimental Setup** . . . 48
6.2 **Empirical Correctness** . . . 48
6.3 **Multiple Partitioning Schemes** . . . 50
6.4 **Scalability** . . . 51
6.4.1 **Same number of Mappers and Reducers as Layers** . . . 52
6.4.2 **Effect on Response Time With change in Reducers** . . . 54

6.4.3 **Effect on Response time With change in Mappers** . . . 55
6.5 **Effect of Graph Connectivity** 56
7. **CONCLUSION AND FUTURE WORK** 58
REFERENCES 59
BIOGRAPHICAL STATEMENT 63

LIST OF ILLUSTRATIONS

Figure Page

1.1 An example graph 2

1.2 Frequent patterns in graphs used for compression 3

1.3 Types of MLN 5

1.4 Frequent subgraphs in HoMLN using existing approach 6

1.5 Frequent subgraphs: Two Layers Combined 7

3.1 Input Graph 18

3.2 Graph Input Along with Adjacency List 19

3.3 Duplicate generation 20

3.4 Canonical Instance 21

3.5 Graph Isomorphism 21

4.1 Overview of the Decoupling Approach for Substructure Discovery . . 24

4.2 Substructures grouped on Vertex id on 1 26

4.3 Example of the Recursive function used for Combining 27

4.4 3-edge MLN Substructures Generated by Combine-MLN 28

4.5 Combining Inter-Layer edges on Vertex id 29

4.6 Example of Duplicates Generated during combining 30

5.1 Workflow of Substructure Discovery in MLN 36

6.1 5-edge Embedded Substructure 49

6.2 Total response time using different partitioning schemes 51

6.3 Map/Reduce time using multiple partitioning schemes 51

6.4 Speed up achieved on LiveJournal Data 53

6.5 Map and Reducer time for LiveJournal 53
6.6 Total Time with change in Reducers 54
6.7 Map/Reduce time with change in Reducers 55
6.8 Total time with w.r.t change in Mappers 56
6.9 Map/Reduce time with w.r.t change in Mappers 57
6.10 Effect of density on Response time 57

LIST OF TABLES

Table | Page
6.1 Data Sets Used in Experiments and their Sizes 49
6.2 Edge Distribution 50
6.3 No of substructures generated in each iteration 54

CHAPTER 1

INTRODUCTION

Developing efficient algorithms for mining frequent itemsets in a large database has been one of the key areas of data mining research. We can use these itemsets for discovering association rules [1] or extracting prevalent patterns that exist in the database. In the recent years, data mining techniques are facing challenges to cope with complex types of objects with inherent relationships, as they are difficult to be modeled with the traditional approaches. Such data are typically modelled as graphs.

Graphs are considered to be a useful natural data structure which can be easily applied to model relationships among complex objects in a variety of applications such as chemical, bioinformatics, computer vision, social networks, text retrieval and web analysis. In particular, when graphs are modeled such that each vertex of the graph will correspond to an entity and each edge will correspond to a relationship between two entities, then the problem of finding frequent patterns becomes that of discovering subgraphs which occurs frequently or compress the graph better over the entire graph or forest.

Substructure discovery is one of the well addressed problems in graph mining domain and is the process of finding interesting patterns that occur in the graph/set of graphs. The motivation for this process is to find inherent regularities in the data. For example:

- What products are often purchased together?
- What kinds of DNA are sensitive to this new drug?
- What are the subsequent purchases after buying a PC?

- Can we classify web documents using frequent patterns?

Figure 1.1 shows an example graph with repetitive patterns, these patterns can also be used to see how it compresses the graph.

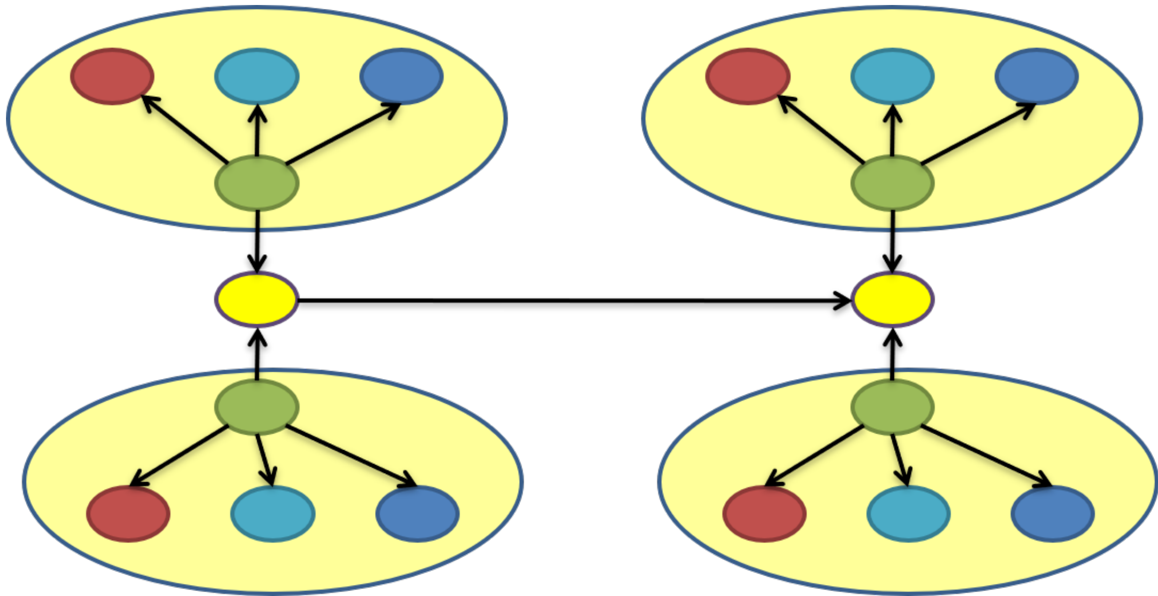

Figure 1.1: An example graph

Figure 1.2 shows how frequent patterns in Figure 1.2(a) can be used to compress the graph given in Figure 1.1 to Figure 1.2(b) and the compressed graph can be further compressed hierarchically to Figure 1.2(c).

The substructure discovery process can be divided into two major steps. The first step is to iteratively generate subgraphs of all sizes in the graph. The next step is to find matching ones in the given graph/graphs in each iteration using subgraph isomorphism. In general, the following four aspects influence the execution of various substructure discovery algorithms and also the output generated by them.

- Graph representation
- Subgraph generation
- Algorithmic approach

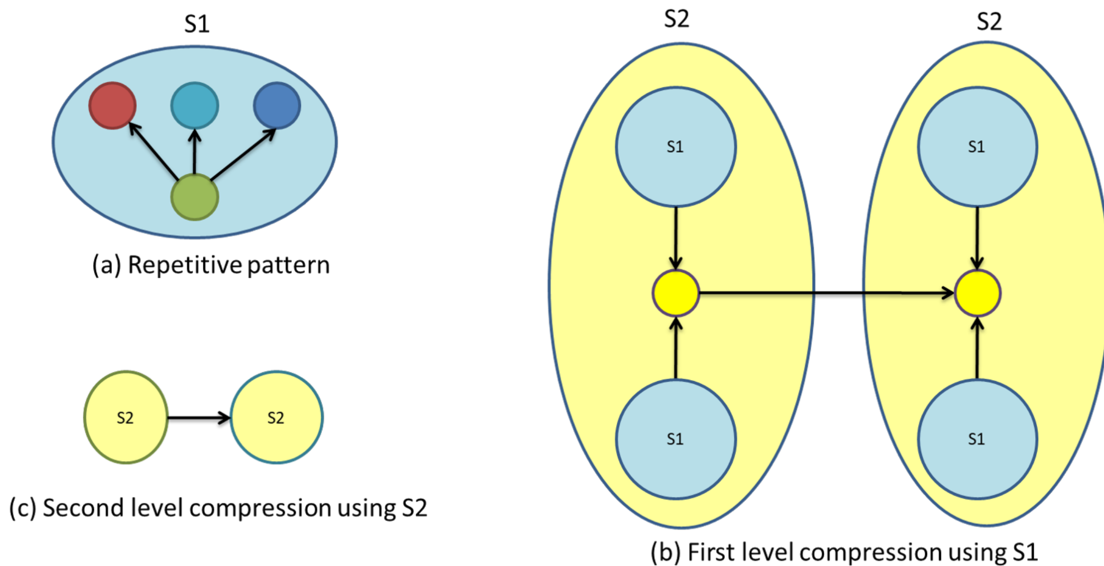

Figure 1.2: Frequent patterns in graphs used for compression

- Frequency evaluation

1.1 Need For Multilayer Networks

As research on complex systems has matured, it has become increasingly essential to move beyond simple graphs and investigate more complicated and realistic frameworks. For example, edges often exhibit heterogeneous features: Modern social networks frequently encompass multiple types of connectivity information; for instance, explicitly acknowledged friend relationships might complement behavioral measures that link users according to their actions or interests. One way to represent these networks are as a Multilayer graph, where each layer contains a unique set of edges over the same underlying vertices (users). Multilayer Networks arise naturally when we have more than one source of connectivity information for a group of users. However, it is not obvious to extend standard graph analysis techniques to the Multilayer setting in a flexible way. In this paper, we focus on developing algorithms and methods for mining Multilayer Networks for substructure discovery.

The advantages of modelling data using MLN's are discussed in [2,3]. Multilayer Networks can be of different types.

- Homogeneous Multilayer Networks
- Heterogeneous Multilayer Networks
- Hybrid Multilayer Networks

Homogeneous Multilayer Networks are used to model distinct relationships that exist among the same type of entities as intralayer edges) and the *interlayer* edge sets are implicit as the same set of nodes are present in every layer. Relationships among different types of entities are modeled through heterogeneous Multilayer network. The *interlayer* edges are explicitly represented to demonstrate the relationship across layers. In addition, for modeling multi-featured data that capture multiple relationships within and across different types of entity sets, a combination of homogeneous and heterogeneous Multilayer Networks can be used, called hybrid Multilayer Network. Figure 1.3 below shows two types of types of Multilayer Networks.

Graph mining in Multilayer Networks has attracted significant attention in the past several years, many algorithms have been proposed for mining complex networks [4, 5] for Cliques and Communities, but these algorithms assume that the data structure of the mining task is small enough to fit in the memory of a computer. But this assumption doesn't hold any longer as very large scale data sets are ubiquitous in today's world: worldwide web, online social network, huge search and query logs regularly collected and processed by search engines. Because of this massive scale, doing analysis and computations on these data sets is infeasible for individual machines. Therefore it is imperative to look for distributed ways for storing and processing these data.

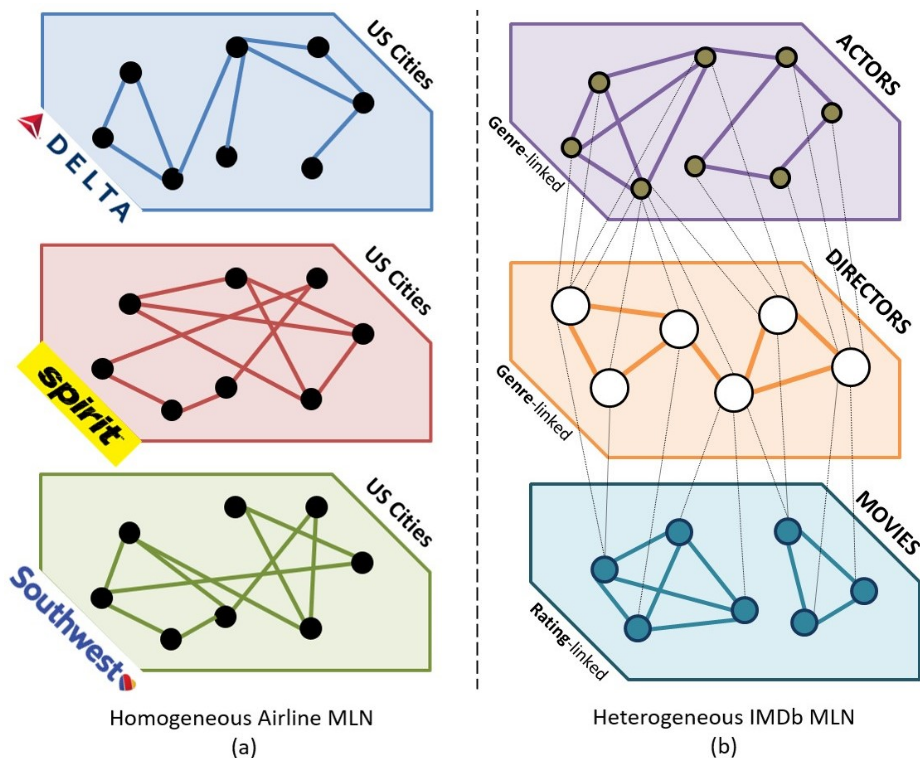

Figure 1.3: Types of MLN

1.2 Problem Statement

The problem addressed by this thesis is to find interesting substructures in a given Homogeneous Multilayer Network without converting the MLN into a single graph. The main challenge here is to compute the *interlayer* substructures across multiple layers correctly and efficiently.

The existing algorithms in literature [6, 7] cannot be used in the multilayer setting as they are designed to find substructures in a single graph. If used to find substructures in each layer, many substructures that exist across layers will not be found. For example Figure 1.4 shows the use of existing algorithms to find substructures of size 2 in each layer independently, but when we combine the graph as single graph and then find the substructures as we see in Figure 1.5, we see that many substructures that exist across layers were not found by the existing algorithms when

we processed them independently. So, the main objective of this thesis is to process multiple layers independently and find frequent patters in each layer and across layers effectively, such that no relationship remains unexplored.

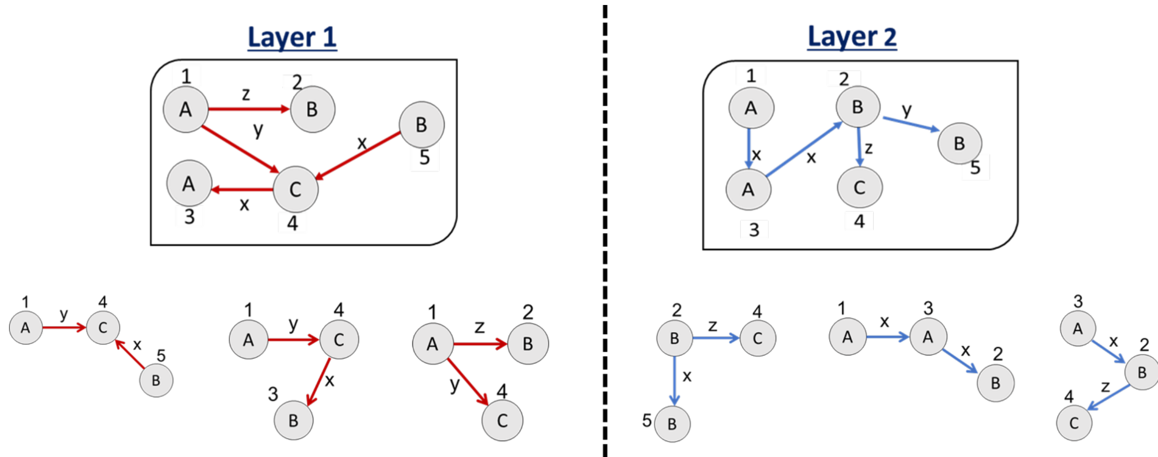

Figure 1.4: Frequent subgraphs in HoMLN using existing approach

1.2.1 Alternatives For Substructures Discovery in a HoMLN

1. **Collapse/aggregate the layers to form a single graph** is an alternative which can be used to find substructures correctly in a given Homogeneous Multilayer network, but using this approach has its downside such as
    - Combining the layers to generate a single graph, which is costly, moreover combining the layers can generate a graph of large sizes which cannot be processed in the memory of single processor and then again we need to resort to the partitioned approach [8] for finding substructures.
    - It restricts the use of parallel processing as we cannot process each layer independently and in parallel.
    - Does not provide clarity to the system, as it is difficult to determine that the substructures generated are from which layer and for every subset of

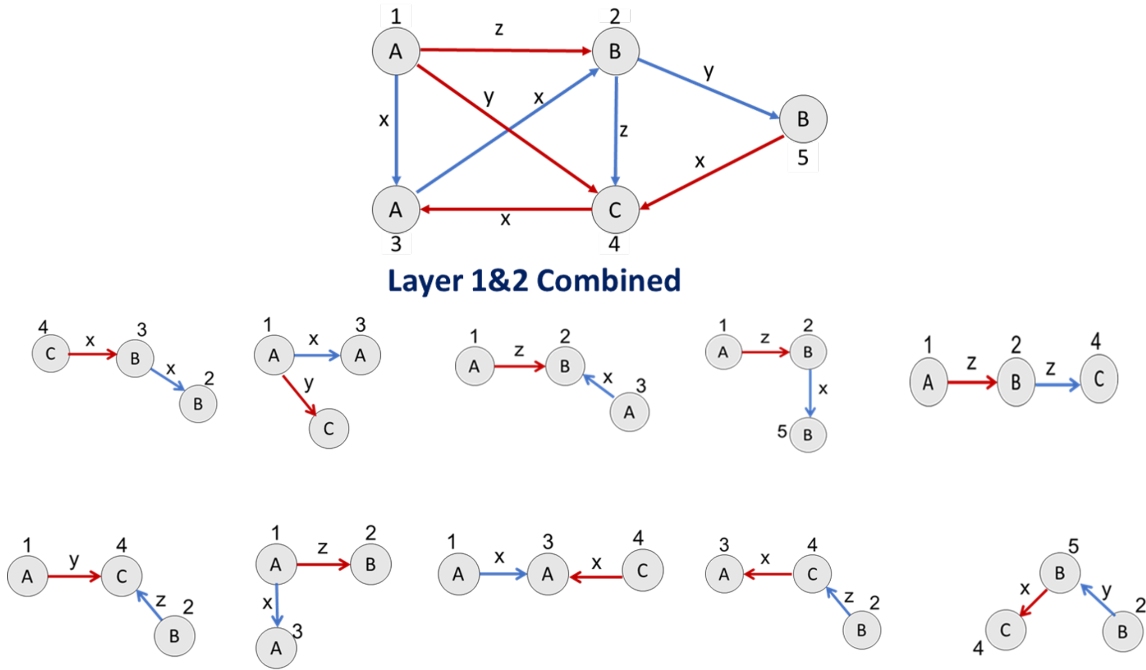

Figure 1.5: Frequent subgraphs: Two Layers Combined

layers we want to find substructures, we need to combine them separately which is inefficient and expensive.

2. **Decoupling based approach** equivalent to the "Divide and Conquer" approach is another alternative to find interesting patterns/substructures in a Multilayer network. The goal is to find substructures in each layer independently and the combine the results to generate substructures that exist across layers. We use the decoupling based approach to address our problem as it allows us to process the layers independently and in parallel, moreover the structure of the MLN is preserved and it provides the flexibility to work on a subset of layers.

1.3 Map/Reduce: A Distributed Framework

Map/Reduce is a framework for processing huge datasets using distributed computation using appropriate number of processors. Its effectiveness and simplicity has

resulted in its implementation by different companies and global adoption for a wide range of applications including large scale graph computations [9, 10]. It is primarily popular because of its model of processing enormous amounts of data in a massively parallel fashion using large number of commodity machines. Its ability to handle lower level issues such as job distribution, data storage, flow and fault tolerance automatically provides a simple computational abstraction. In Map/Reduce computations are done in three phases:

- **Map Phase:** It reads a collection of values or key/value from a input source (HDFS: File system used by hadoop to store data) and invokes a user defined **Mapper** function on each input element independently and in parallel, and emits zero or more key/value pairs associated with that input element.
- **Shuffle Phase:** This phase groups together all the key/value pairs emitted by the mapper sharing the same key, and outputs each distinct group to the next phase.
- **Reduce Phase:** It invokes a user-dened Reducer function on each distinct group, independently and in parallel, and emits zero or more values to associate with the group's key. The emitted key/value pairs can then be written on the disk or be the input of a Map phase in a following iteration.

We have chosen the Map/Reduce paradigm to address our problem of substructure discovery in a Multilayer Network as we are interested in processing the layers independently and in parallel and the Map/Reduce framework provides the architecture to accomplish that goal in a flexible way. The Mapper allows us to process the layers in parallel to find substructures in each layer and then the Reducer enables us to group all the substructures generated in each layer to find the substructures that exist across all layers. Using this framework, our goal to preserve the MLN structure and process the layers independently is also accomplished. Moreover if we don't have

enough resources (Mappers/Reducers) to match the number of layers, Map/Reduce framework also allows us to process multiple layers in a single processor.

1.3.1 Substructure discovery using Map/Reduce

Experiments indicate that instead of using high computation supercomputer, it is feasible to use a cluster of cheap commodity machines for graph mining on large data sets. At Present, an iterative algorithm is used for substructure discovery on partitioned graph using Map/Reduce paradigm that generates all substructures of increasing sizes (starting from substructure of size one that has one edge), eliminates duplicates if necessary, counts the number of identical (or similar) substructures, applies a metric (e.g. frequency) to rank the substructures. This process is repeated until a given substructure size is reached or there are no more substructures to generate. In each iteration, either all substructures or a subset of substructures (using ranking) is carried forward to restrict the expansion process.

This approach partitions a single large graph in smaller chunks and processes them in parallel. However, this thesis focuses on developing algorithms that works on a Multilayer graph. So the process of iteratively generating bigger substructures in each partition and computing identical substructures needs to be remodelled to work correctly on Multi-layer graphs, where processing in each layer should not be dependent on the other, as that beats the purpose of parallelization.

1.4 Thesis Contributions

The contributions of this thesis are:

- Advancing a scalable graph mining approach for substructure discovery in a homogeneous multi-layer network, where each layer is processed independently and in parallel.

- Correctly combining the edges in each layer to generate substructures across layers.
- Developing a Map/Reduce based substructure discovery algorithm on large single graphs to work correctly on multi layer graphs.
- Managing graph mining operations such as graph expansion, similarity computations in a Map reduce framework.
- Extensive experimental analysis using large scale real-world and synthetic datasets.

1.5 Road Map

Rest of the thesis is organized as follows:

- Chapter 2 surveys the related research work with respect to the motivation of this dissertation.
- Chapter 3 elaborates on the preliminaries for graph mining such as input graph representation, sub graph expansion, duplicate elimination, graph isomorphism, partition management in the Map/Reduce framework.
- Chapter 4 presents our partition based graph mining technique and algorithm for substructure discovery in a multi-layer network.
- Chapter 5 discusses the implementation details for all the components used for substructure discovery in a Multi-Layer Network.
- Chapter 6 provides extensive experimental analysis of several data sets along with drill-down analysis.
- Chapter 7 concludes the dissertation with directions for future work.

CHAPTER 2

RELATED WORK

As the focus of this thesis is mining interesting substructures in a Homogeneous Multilayer Network, we now present the relevant work on single and multilayered graphs.

2.1 Main Memory Approaches

One of the first prominent algorithm proposed for substructure discovery using a main memory approach was **SUBDUE** [6, 11–13]. The substructure discovery algorithm used by subdue is a computationally- constrained beam search where substructure are generated iteratively and evaluated using the MDL [14] metric. Input to the subdue algorithm is in the form of a labelled multigraph, where two vertices can have multiple edges between them. The algorithm begins with substructure of size one (one edge). Each iteration through the algorithm selects the best (or all) substructures and expands the instances of these substructures by one neighboring edge in all possible ways. The algorithm retains the best substructures in a list, which is returned when either all possible substructures have been considered or the total amount of computation exceeds a given limit. One of the interesting property of subdue is the use of background knowledge [15], it is used to guide the search towards more appropriate substructures. This background knowledge is encoded in the form of rules for evaluating substructures, and can represent domain independent or domain-dependent rules. Every time when a substructure is being evaluated, these

input rules are used to determine the value of the substructure under consideration. Because only the most-favored substructures are kept and expanded, these rules bias the discovery process of the system. Each rule is assigned a positive, negative or zero weight, that biases the procedure towards a type of substructure. Hence, the evaluation of each substructure is guided by the MDL principle and the background knowledge provided by the user.

Apriori based approach It is similar to frequent item set mining and searches for repetitive substructures and starts with graphs of small sizes and proceeds in a bottom-up approach. **AGM** [16] Apriori Graph mining algorithm generates candidate graphs, merges any two candidate graphs at an instant and checks whether the resultant graph is a sub graph in a given graph / graph database or not. Here, the size of a graph is denoted by the number of vertices present in that graph. Two graphs of size 'k' can be merged together to form a resultant graph of size 'k+1'.One or more resultant graphs of size 'k+1' is again fed into the apriori algorithm to obtain a resultant graph of size 'k+2'. So, in each and every iteration of this algorithm, two graphs (arbitrarily chosen from the candidate set) are merged together to form a resultant graph whose size is increased by one vertex.

2.2 Disk based approach

This approach [17] was developed to overcome the problem of storing the entire data set in main memory, where some portion of data is kept in memory and the rest on disk. Since random access to disk based graphs can be difficult and costly, indexing the graph seemed to be the optimal choice. Frequent subgraphs are ideal candidates for indexing since they are relatively stable to database updates, thereby

making incremental maintenance of index affordable. They also provide an efficient solution on index construction: we can first mine discriminative structures from a small portion of a large database, and then build the complete index based on these structures by scanning the whole database once.

2.3 Database approach

Disk-based algorithms solved the problem of keeping portions of the graph in memory for processing. However, these algorithms need to assemble data between external storage and main memory buffer which has to be coded into the algorithm. The performance of disk-based approaches can be very sensitive to optimal transfer of data between disk and memory, as well as buffer size, buffer management (or replacement policies) and hit ratios. An alternative approach [18, 19] was proposed to overcome this by efficient buffer management and query optimization, by mapping these graph mining algorithms to SQL. Although scalability was achieved to graph sizes over a million nodes and edges, use of joins for substructure expansion turned out to be computationally expensive as they involved joins of very large relations. Also, the removal of duplicate substructures required sorting columns (in row-based RDBMSs) making this component expensive as well in terms of the number of joins needed.

2.4 Partition based approach

As the size of the data is increasing rapidly, many problems are so large or complex that it is impractical or impossible to solve them on a single computer, especially with given limited memory. Scalable parallel computing algorithms holds the

key role for solving the problem in this context. This approach [7,8] was proposed to addresses the problem of substructure discovery by dividing the graph into smaller partitions and then combining the results across partitions effectively. A MapReduce based algorithm for horizontal scalabilty of substructure discovery that can work with any partitioning strategy.

As the goal of this approach is to partition the data among different processors, a substructure partition and a corresponding adjacency partition needs to be created to expand a substructure. Below are the two partitioning schemes proposed.

Range partitioning scheme: This strategy partitions the global adjacency list based on the range information to create adjacency list partitions as the global adjacency list cannot be loaded in memory. The range information is then used to determine the adjacency partition for a single vertex and directs a k-edge substructure to its appropriate adjacency partition. The range adjacency list partitions do not have any intersection of vertices among them (adjacency list of a vertex cannot be present in more than one partition).In this approach, adjacency list partitions are contiguous range of vertex ids. The number (and even the size) of partitions can tailored to match memory availability. Range partitioning can be done over a single pass of the graph data input. After all the substructures are routed to their appropriate partitions then it generates all substructures of increasing sizes (starting from substructure of size one that has one edge), counts the number of identical (or similar) substructures and applies a metric to rank the substructures. This process is repeated until a given substructure size is reached or there are no more substructures to generate.

Arbitrary partitioning: In this approach each substructure partition is paired with a corresponding adjacency partition. In every iteration, each substructure partition is expanded in parallel using the corresponding adjacency partition. But an

expansion of a substructure may add a vertex id whose adjacency list is not in the current adjacency partition which necessitates the update of adjacency partition by adding the adjacency list of the new vertex after every iteration. With graphs of bigger sizes, such an update phase has a huge bearing on the I/O. One way to eliminate the above overhead is to keep the adjacency partition fixed in each iteration like we did in Range partitioning but it also has a cost associated with it, routing the substructure to a processor holding the appropriate adjacency partition. Hence connectivity of the graph plays a crucial role, typically a worst case update in the arbitrary partitioning approach requires updating every adjacency partition making it very expensive, so with the graphs of increasing sizes, Range approach is believed to perform better.

2.5 Subgraph Mining in a Multilayer Network

All the previous approaches were for single graphs, but in many applications graphs are enriched by additional information, for example vertices in graphs can have multiple relationship between them, where the multiple edges can be modelled as different layers of the same underlying graph. As there exist multiple edges between vertices, this paper considers that edge labels represent characteristic's of the relationship.

The Algorithm [4] determines to find clusters in a multilayer graphs by partly using an efficient algorithm [20] which is used for finding quasi cliques and then find all one dimensional clusters in a single layer and the next step is to compose the resulting patterns to find multi-dimensional clusters and to limit the search space it removes the redundant clusters. So, the overall focus of this work is to find clusters of vertices that are densely connected by edges with similar edge labels in a subset of graph

layers. But being a main memory approach this technique cannot be used for large graphs which cannot be processed in the memory of a single processor.

All of the work discussed above are for substructure discovery in single graphs, however our problem is to find substructures in MLN, by processing the layers independently. Most of the work done in Multilayer Network, not particularly addresses substructure discovery. We take inspiration from the Substructure discovery algorithm in single graphs, decoupling based approach used in Multilayer Network and work on developing an algorithm for substructure discovery in a Homogeneous Multilayer Network.

CHAPTER 3

Preliminaries

Graph mining consists of algorithms applied to a graph data in order to discover interesting substructures as knowledge. A graph is a collection of objects. Each object in a graph is called a node (or vertex). Corresponding to the connections in a network, there are edges (or links) in a graph. Each edge in a graph joins two distinct nodes. In this chapter we present the preliminaries for graph mining, substructure discovery and various aspects involved in it.

3.1 Graphs

Graphs are used to represent many real-life applications such as networks. These networks may include paths in a city or a telephone network. Graphs are also used in social networks like LinkedIn, Facebook. For example, in Facebook, each person is represented with a vertex (or node). Each node is a structure and contains information like person id, name, gender etc, and the nodes are connected with edges which can represent the relationship between them.

3.1.1 Input Graph Representation

Graph representation is an important aspect in substructure discovery because it has direct and significant influence on memory usage as well as execution time of the algorithm. Various graph representation schemes are available, among which adjacency matrix, adjacency list, hash table are frequently used by the mining algorithms. Adjacency matrix representation is easier to implement, but there can be a

considerable waste of memory if the input graph is sparse whereas the adjacency list representation consumes less space compared to adjacency matrix [21]. In this paper, we focus on labeled graphs where node and edge labels are not assumed to be unique, however all the vertex ids are unique. Figure 3.1 shows a graph with vertex ids, node and edge labels.

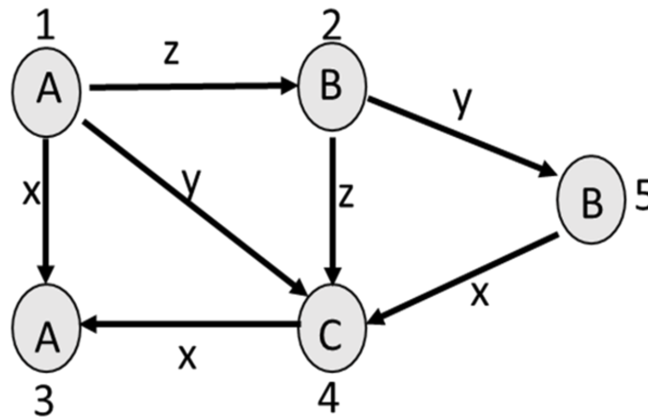

Figure 3.1: Input Graph

Our input graph is represented as a sequence of one edge substructures including its direction. Each edge is represented as a 5 element tuple *(edge label, source vertex id, source vertex label, destination vertex id, destination vertex label).* A graph is stored as a ascii file with a 1-edge substructure in each line and acts as an input to our algorithm.

3.1.2 **Adjacency list**

An adjacency list of a vertex id is the list of edges in which that vertex id appears. The number of edges in an adjacency list for vertex is the sum of in and out degrees of that vertex in the graph. We use the adjacency list of a vertex to expand

Edges
<z,1,A,2,B>
<x,1,A,3,A>
<y,1,A,4,C>
<z,2,B,4,C>
<x,4,C,3,A>
<y,2,B,5,B>
<x,5,B,4,C>

Vid	Adjacency List
1	[<z,1,A,2,B> <x,1,A,3,A> <y,1,A,4,C>]
2	[<z,1,A,2,B> <z,2,B,4,C> <y,2,B,5,B>]
3	[<x,1,A,3,A> <x,4,C,3,A>]
4	[<x,4,C,3,A> <z,2,B,4,C> <x,5,B,4,C>]
5	[<y,2,B,5,B> <x,5,B,4,C>]

Figure 3.2: Graph Input Along with Adjacency List

vertex id with an edge in which that vertex id appears by adding one edge to that instance from the adjacency list. Figure 3.2 shows the one edge substructures and the adjacency list associated with it.

3.2 Graph expansion

As we are interested in discovering substructures of any size and our input is a single edge substructure, graph expansion plays an important role in the process. Expanding each edge of the input by adding a single edge in every direction forms the base of our expansion, to ensure correctness the expansion process is unconstrained, that is each substructure independently grows into a number of larger substructures in each iteration. Such a expansion leads to **duplicates**, which must be dealt with (by removing them) to ensure correctness. Figure 3.3 shows an example of how duplicates are formed during substructures discovery. Note: The duplicates have the same vertex id and the same connectivity.

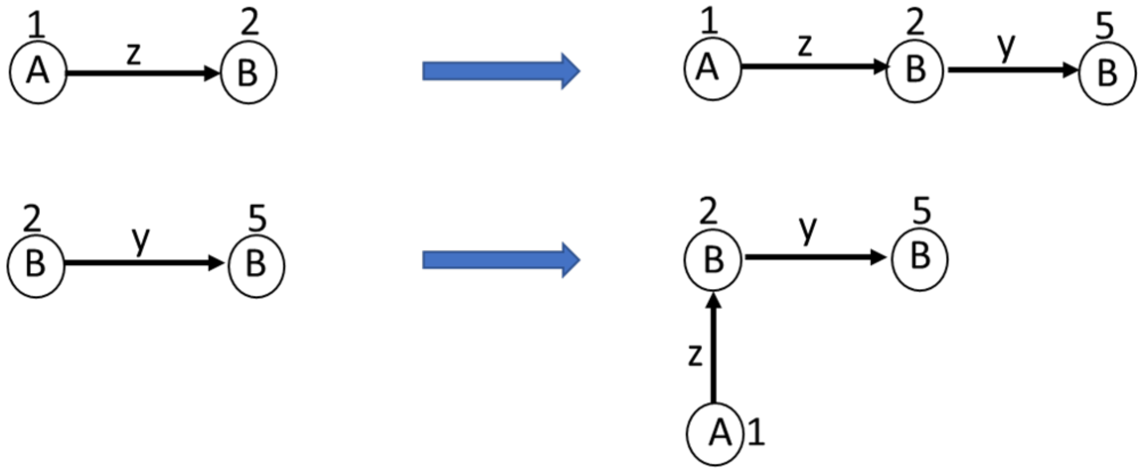

Figure 3.3: Duplicate generation

3.2.1 Canonical Instance

To identify duplicates, we employ a lexicographic ordering on edge label. If there are multiple edges with the same edge label in a substructure, they are ordered on the source vertex label. If source vertex label is also same, they are further ordered on the destination vertex label. If edge label, and vertex labels are also identical, then source and destination vertex ids are used for ordering. Hence, a substructure can be uniquely represented using the above lexicographic order of 1-edge components. We call this a canonical k-edge instance. Intuitively, two duplicate k-edge substructures must have the same ordering of vertex ids and thereby the same canonical k-edge instance. Figure 3.4 shows an example of duplicates and its instance and shows how duplicates will have the same canonical instance.

3.2.2 Graph Isomorphism

We use graph isomorphism to detect identical substructures in a graph. Isomorphs have the same vertex and edge labels but differ in vertex ids, and it is imperative to identify isomorphs so that we can count there occurrences. Intuitively,

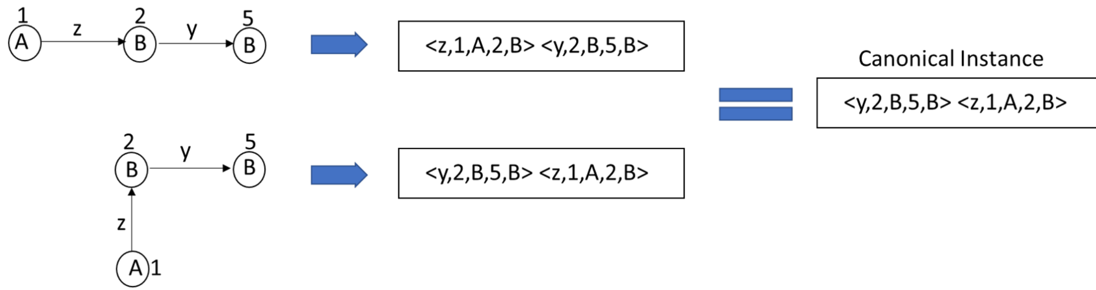

Figure 3.4: Canonical Instance

two isomorphic substructures shall have the same relative ordering of vertex ids. So, we use the canonical instance to derive canonical substructure from it by replacing each vertex id with their relative positions in the instance starting from one. The inclusion of these relative positions is critical for differentiating the connectivity of the instances.

Figure 3.5 shows an example of how canonical substructure is created from the canonical instance. We can see that the isomorphs have different canonical instances. However using the above technique, the relative positioning of Vertex id (2, 5, 4) for the canonical instance 1 and (7, 10, 9) for the canonical instance 2 boils down to (1, 2, 3). Hence we can identify isomorphs using the canonical substructure.

Canonical Labeling is one of the key operations in our approach as it enables us

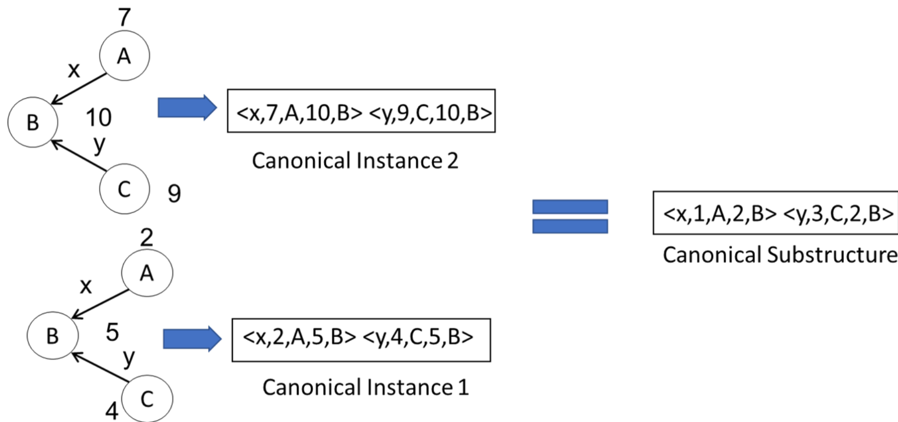

Figure 3.5: Graph Isomorphism

to determine duplicates and isomorphic instances and we will use the notion of a canonical instance and a canonical substructure throughout this paper to distinguish between duplicates and isomorphic substructures.

3.2.3 **Frequency Calculation**

Knowing (or counting) the number of isomorphic substructures is important for subgraph discovery. Depending on the metric used frequency or compressibility is used for identifying *interesting* substructures. Typically, each substructure occurs in the graph multiple times. Therefore, the importance of a substructure with respect to the graph can be measured using metrics based on the number of occurrences. This measure is typically used to rank the substructures. One of the commonly used ranking metrics for graph mining are: Minimum Description Length (MDL) [14] and frequency to rank the substructures. *Minimum Description Length* highlights the importance of substructure on how well it can compress the entire graph. Both the structure of the subgraph and the number of its instances have a bearing on compression. Frequency, on the other hand, determines the importance of a canonical substructure solely by the number of occurrences of its instances. Note that, counting the number of instances is common to both of these metrics.

Substructure discovery in large graphs becomes complicated as patterns in the graphs starts to overlap, we have used a MRN (Most Restrictive Node) [22,23] metric to count the non overlapping instances. But we also keep a track of the overlapping instances and use both to compute the frequency and MDL for a given substructure.

CHAPTER 4

DECOUPLING-BASED APPROACH

This chapter focuses on the design of the algorithm for a parallel processing environment for discovering substructures in a Multilayer Network. We are using an iterative algorithm that generates all substructures of increasing sizes in each iteration in each layer *(Intra-Layer)* and combines the expanded substructures of each layer to create *(Inter-Layer)* substructures, eliminates duplicates, counts the number of isomorphic substructures and applies a metric to rank them. This process is repeated until a given substructure size is reached or there are no more substructures to generate. For m layers, m k-edge substructures from each of m layers are used to generate k-edge substructures of m layers.

4.1 Decoupling-Based Approach

As we want to support processing each layer of the Multilayer Network independently and in parallel, we use a decoupling-based approach developed in [5] for substructure discovery. It uses the "divide and conquer" technique to generate substructures for individual layers and then combines substructures from individual layers to find substructures across MLN. We first find substructures of size k in each layer (starting with k value as 1) independently and then apply the composition function (which is the combining algorithm explained later) to find the substructures of size k that exists across all layers. In particular, we do not apply the composition function after finding substructures of all the sizes in each layer, but we do it after each iteration. For example, we start with a k-edge substructure in each layer (k is 1 for the

first iteration), and expand all the vertices in that substructure separately by adding one edge. We then apply the composition function on the expanded instances of each layer to find substructures that exist across layers and this process continues iteratively until a termination condition is applied. Figure 4.1 shows the overview of the decoupling-based approach for substructure discovery in a Homogeneous Multilayer Network.

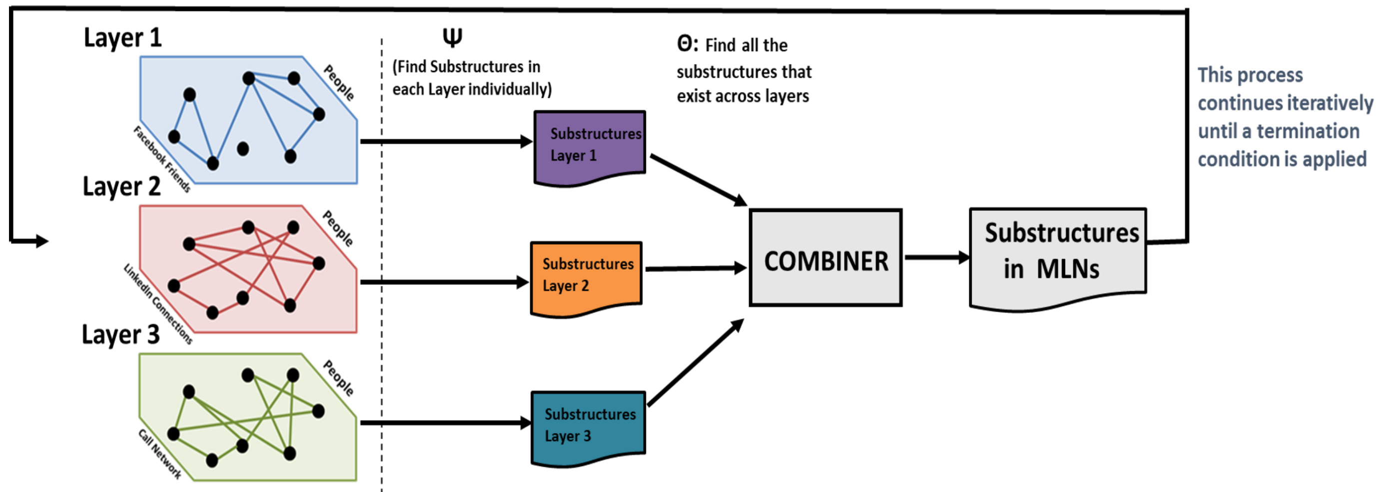

Figure 4.1: Overview of the Decoupling Approach for Substructure Discovery

Using a Decoupling approach instead of combining multiple layers as a single graphs has numerous advantages:

- It perseveres the MLN modelling and structure. But if we choose to combine the graphs, it is not possible to identify that the substructures generated are from which layer.
- It provides the flexibility to work on a subset of layers. Suppose we have 10 layers and we only want to find substructures in 6 Layers, then the decoupling approach provides a easy way to do that, whereas if we use the alternative approach of combining the graphs, we need to combine graphs for all the subset of layers we want to work on, which is ineffcient and expensive.

- Using a Decoupling approach provides parallelization opportunities where we can process each layer in parallel, which helps decrease the response time.

.

4.2 Combining Algorithm

As discussed above, finding substructures in each layer is not enough, as lot of substructures that exist across layers remains unexplored. So to ensure correctness, we need to develop an algorithm to combine the substructures from each layer to find substructures that exist across layers. We are dealing with homogeneous Multilayer network, where all the layers have the same set of nodes but the connectivity across the nodes in each layer is different. For any connected subgraph, there exists a common vertex or node between them which connects to the edges from different layers which forms the basis of our combining algorithm. So, if we group all the substructures generated from each layer based on their vertex id (node), it will bring all the edges across layers together and then we can combine the edges that share a common vertex id that exist in different layers.

As we use an iterative substructure discovery algorithm where we find substructures of increasing sizes in each layer, combine substructures from each layer after each iteration so we do not miss substructures of that size in the MLN. We use a recursive algorithm for combining substructures from m layers of size k using – edge substructures from each layer (size of the substructure remains same after combining). The motivation behind is to ensure consistency for the size of the substructures generated after each expansion and combining in each iteration. Using this approach we do not have to deal with substructures of varying sizes in each iteration.

Now, lets take a detailed look at the algorithm used for substructure discovery in MLNs. The first step of the algorithm is to group the instances based on their

Vertex id, then apply the Combine-MLN function which is a recursive function that checks for all the combinations in a k-edge substructure and combines the *Intralayer* substructures which share the same Vertex Id, this function is applied on all the vertices of the graph. Figure 4.2 Shows a 3-edge substructure from a MLN with 3 layers, grouped on Vertex id 1. Our goal is to find all possible 3-edges substructures across layers. We now apply the Combine-MLN function on all the edges grouped on this Vertex id, which has two parameters:

1. Size of the MLN (No of Layers)
2. Size of the input and combined substructures (they are same in our approach)

{(x,1,a,2,b ; y,1,a,3,c ; z,3,c,4,a) Layer 1
(x,1,a,5,c ; y,1,a,6,d ; z,6,d,4,a) Layer 2
(x,1,a,8,c ; y,1,a,9,b ; z,9,b,6,d) Layer 3}

Figure 4.2: Substructures grouped on Vertex id on 1

Figure 4.3 shows our Combine-MLN function, where the size of the MLN and substructure is 3, can be referred from the Figure 4.2. We can either choose 1-edge substructures from all the 3 layers or 2-edge substructure from 1 layer and the other 1-edge from either layer. The left branch of the tree with “L1=0” indicates that “0” edges are to be chosen from Layer1, and all the substructures are to be found from Layer2 and Layer3. All the substructures of this category are found at Step:1 and 2, which can be referred from Figure 4.4. The centre branch with “L1=1”, means 1-edge substructure from Layer1 and the other 2 from Layer2 and Layer3, which brings two conditions: 1-edge from both the layers or 2-edge from either layer. All the substructures of this combination are found at Step:3, 4 and 5. The rightmost

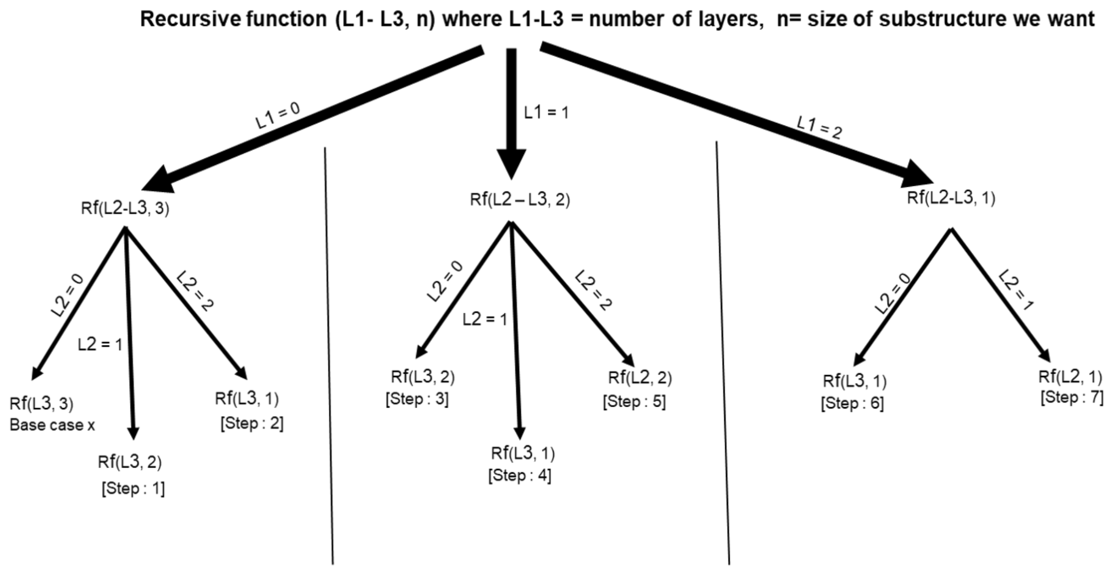

Figure 4.3: Example of the Recursive function used for Combining

branch with "L1=2", specifies that a 2-edge substructures is to be chosen from Layer1 and the other 1-edge from either Layer2 or 3. Substructures of this group are found at Step:6 and 7.

Figure 4.4 shows all the substructures across layers found by this function at each step (by step number. Similarly all the branches of the tree specifies a condition for the size of the substructure to be chosen and the Combine-MLN function is called at each step until the base case is reached which is, the required size of substructure reaches to zero or only one layer is left to process. The number combinations increases as the size of the substructures starts to increase in each iteration, contributing to width of the tree, whereas the number of Layers in the MLN is directly proportional to the height of the tree, which remains fixed across iteration.

Now, lets see Figure 4.5, which has two layers, and shows all the 2-edge substructures in each layer, now when we apply the Combine-MLN function on each Vertex id of the graph as shown above, we find all the substructures of size 2, which

(y,1,a,9,b ; z,9,b,6,d ; y,1,a,6,d) L3L2 – Step1
(y,1,a,9,b ; z,9,b,6,d ; x,1,a,5,c) L3L2 – Step1
(x,1,a,8,c ; y,1,a,9,b ; y,1,a,6,d) L3L2 – Step1
(x,1,a,8,c ; y,1,a,9,b ; x,1,a,5,c) L3L2 – Step1
(y,1,a,6,d ; z,6,d,4,a ; y,1,a,9,b) L2L3 –Step2
(y,1,a,6,d ; z,6,d,4,a ; x,1,a,8,c) L2L3 – Step2
(x,1,a,5,c ; y,1,a,6,d ; y,1,a,9,b) L2L3 – Step2
(x,1,a,5,c ; y,1,a,6,d ; x,1,a,8,c) L2L3 – Step2
(x,1,a,5,c ; y,1,a,6,d ; x,1,a,8,c) L2L3 – Step2
(y,1,a,9,b ; z,9,b,6,d ; x,1,a,2,b) L3L1 – Step3
(x,1,a,8,c ; y,1,a,9,b ; y,1,a,3,c) L3L1 – Step3
(x,1,a,8,c ; y,1,a,9,b ; x,1,a,2,b) L3L1 – Step3
(y,1,a,9,b ; z,9,b,6,d ; y,1,a,3,c) L3L1 – Step3
(x,1,a,2,b ; x,1,a,5,c ; x,1,a,8,c) L1L2L3 – Step4
(x,1,a,2,b ; x,1,a,5,c ; y,1,a,9,b) L1L2L3 – Step4
(y,1,a,3,c ; x,1,a,5,c ; x,1,a,8,c) L1L2L3 – Step4
(y,1,a,3,c ; y,1,a,6,d ; y,1,a,9,b) L1L2L3 – Step4
(y,1,a,3,c ; x,1,a,5,c ; y,1,a,9,b) L1L2L3 – Step4
(y,1,a,3,c ; y,1,a,6,d ; x,1,a,8,c) L1L2L3 – Step4
(x,1,a,5,c ; y,1,a,6,d ; x,1,a,2,b) L2L1 – Step5
(x,1,a,5,c ; y,1,a,6,d ; y,1,a,3,c) L2L1 – Step5
(y,1,a,6,d ; z,6,d,4,a ; x,1,a,2,b) L2L1 – Step5
(y,1,a,6,d ; z,6,d,4,a ; y,1,a,3,c) L2L1 – Step5
(x,1,a,2,b ; y,1,a,3,c ; x,1,a,8,c) L1L3 – Step6
(x,1,a,2,b ; y,1,a,3,c ; y,1,a,9b) L1L3 – Step6
(x,1,a,3,c ; z,3,c,4,a ; x,1,a,8,c) L1L3 – Step6
(x,1,a,3,c ; z,3,c,4,a ; y,1,a,9,b) L1L3 – Step6
(x,1,a,2,b ; y,1,a,3,c ; x,1,a,5,c) L1L2 – Step7
(y,1,a,3,c ; z,3,c,4,a ; x,1,a,5,c) L1L2 – Step7
(x,1,a,2,b ; y,1,a,3,c ; y,1,a,6,d) L1L2 – Step7
(x,1,a,3,c ; z,3,c,4,a ; y,1,a,6,d) L1L2 – Step7

Figure 4.4: 3-edge MLN Substructures Generated by Combine-MLN

exist across layers. The parameters of Combine-MLN function here is(2,2), which means there are 2 layers and the required size of the substructure is 2.
Together combined, are all the substructures in a Homogeneous MLN which would not have been possible to find if don't combine substructures across layers.

4.2.1 Challenges for Combining

As we are checking for all the combination in Intralayer edges, in some cases it can lead to generation of duplicates, when the same subset of a edge from the same layer is combined with the same edge from a different layer. Figure 4.6 shows how duplicates are generated. We need to remove them to ensure correctness. We can either remove it using a SET or before we add them to the list of local combined instances at each step, we can check the adjacent index's for duplicates and remove them. Moreover, our combining follows the canonical form so that we can identify the duplicates.

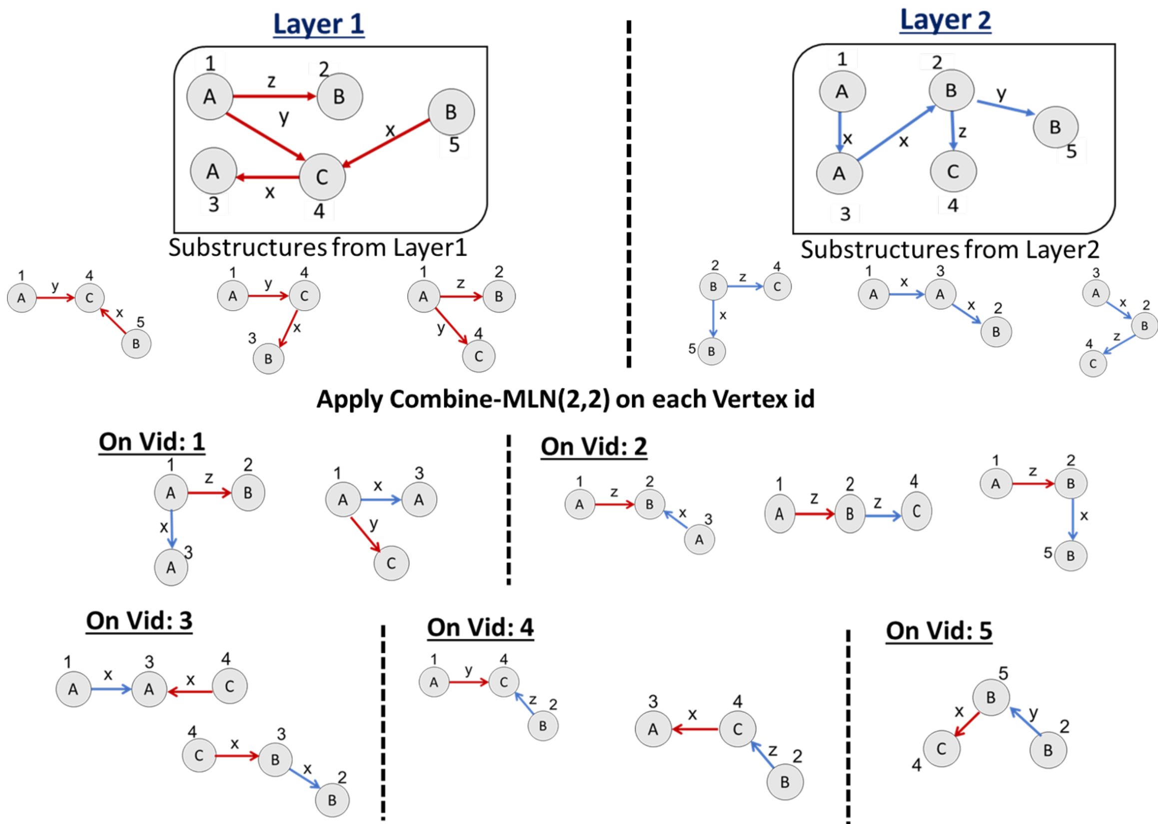

Figure 4.5: Combining Inter-Layer edges on Vertex id

Using this approach, we find all the substructures of all sizes that exist across layers in each iteration and don't miss out on any substructures.

4.3 Algorithmic Approach

In this section we will discuss our iterative algorithm used for finding substructures in a Homogeneous Multilayer Network

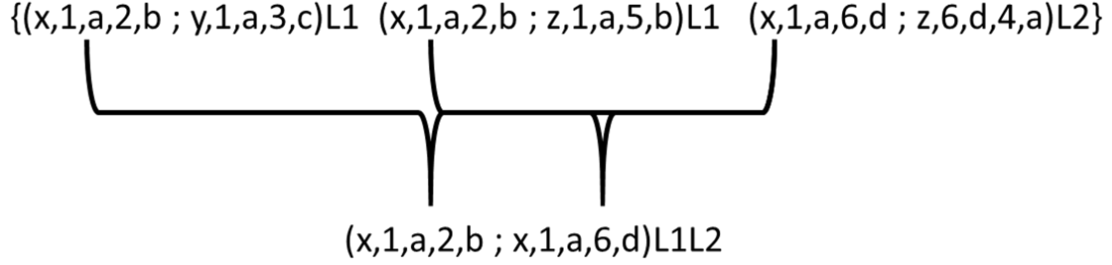

Figure 4.6: Example of Duplicates Generated during combining

Algorithm 1 Substructure Discovery in a Multi layer Network

Require: -

1: ***Input:*** Homogeneous Multilayer Network
2: ***Output:*** Top n-substructures in a Multilayer Network
3: **Load** Adjacency list for each layer
4: **for** each edge in MLN:
5: **Expand** k-edge substructure by one edge in all directions in each layer
6: **Eliminate Duplicates** using the Canonical representation in each layer
7: **end for**
8: **for** each expanded instance:
9: **Group** all the expanded instances from each layer based on their Vertex id
10: **Combine** all k-edge Intralayer edges sharing the same Vertex id and exist in different layers to form k-edge Interlayer edges
11: **Eliminate Duplicates** generated during Combining the Intralayer edges
12: **end for**

13: **for** all the canonical instances in the MLN:

14: **Count** the frequency of all the instances using Graph Isomorphism with or without Overlaps// Using relative canonical representation

15: **end for**

16: **Apply Metric** Frequency or MDL // With Overlaps or Ignoring Overlaps

17: **Apply Heuristic BEAM** to retain top-n substructures and their instances to be used the next iteration // Specify n depending on the need

18: **Increment k** by 1 for the next Iteration

19: **Begin** from *Step 4* for the next iteration

We now discuss the four major components of our Algorithm in detail

1. **Expansion:** Each layer is read as the input to our algorithm (one instance at a time), while the adjacency list is loaded and kept in memory for expansion (line 3). We expand each edge in the input by adding one edge in all directions using the adjacency list (line 4-5). As the expansion process is unconstrained to ensure correctness, some *Duplicates* are generated which needs to be removed to ensure correct count. As the expansion process follows the canonical ordering, we can identify duplicates and remove them (line 6) before sending to the next step .

2. **Combining:** Now that we have all the expanded substructures from each layer, we group them based on their vertex id(node) (line 9), the reason behind this is as we are dealing with a Homogeneous Multilayer network, where all the layers have the same set of underlying nodes. So it will bring all the edges with same

vertex id from each layer together and then we combine all the edges across layers that share a common vertex id and exist in different layer (line 10), but there are challenges associated with it which we have addressed in the above section. Now that we have all the edges *Inter and Intra* we move to the next step.

3. **Frequency Counting:** We use Graph Isomorphism to detect exact (identical) patterns in a graph. Intuitively, two isomorphic substructures will have the same relative ordering of the vertex ids and they have the same vertex and edge labels. Note: the canonical instances already follows the lexicographic ordering, hence it is easy to generate a k-edge canonical substructures, using the relative positioning of unique vertex id in the order of their appearance in the canonical instance. Now we group all the instances based on their canonical form and count their frequency (line 13-14).

4. **Apply Metric:** To restrict the future expansion to high quality substructures, we use a metric (line 16) *MDL* or *Frequency* to determine the importance of a particular substructure. We then apply a heuristic (BEAM) (line 17), whose value determines how many substructures are to be carried in the next iteration. So this will prune the unimportant candidates and will only use the high quality substructures for future expansion.

CHAPTER 5

Implementation Details

This chapter contains the implementation details of all the methods and procedures used for Substructure Discovery in a Homogeneous Multilayer Network using the Map/Reduce architecture and its open source implementation, Hadoop [24].

5.1 Layer Generation

A data set modeled as an HoMLN would have each layer as a graph and that would be the input to this substructure discovery algorithm. Since we are *not* using MLNs at present, we create MLNs from a large single attribute graph for the purposes of testing with large number of layers and control other graph characteristics, such as sparcity (% of edges as compared to a complete graph). We use this approach for empirical verification as well by embedding substructures of known frequency to make sure we get the *same substructures* when processed as a single graph and when processed as layers.

Hence, the first step of implementation is the generation of Homogeneous Layers from a single attribute graph. For this we use the synthetic graph generator Subgen. It allows us to generate graphs of different sizes and embed substructures with varying sizes and frequency. The graphs generated by Subgen is not in the form supported by our algorithm. We do not make any assumptions about the input representation of graphs. We need to modify the graph format to the one required by our algorithm.

We use a python script to modify the data generated by Subgen, into our input format. Our input graph is represented as a sequence of one edge substructures that

includes direction. Each edge is represented as a 5 element tuple *(edge label, source vertex id, source vertex label, destination vertex id, destination vertex label).*

5.1.1 Partitioning Schemes

Now that we have the graph in our format, we need to create Layers from it. With the help of a python script we start writing out edges from the input data into different files/layers. The function reads each edge from the input (single graph) and partitions it into different files, the number of files depends on the number of layers required. We have created multiple layers, depending on the need. Each edge is written into a single file only, i.e we don't have the same edge in multiple layers. We have implemented two partitioning schemes to generate our homogeneous layers, the intuition behind using two partitioning schemes is to verify that our algorithm works correctly irrespective of how the data is partitioned and does not depend on the connectivity of the graph. The other reason is also to investigate that whether the different partitioning schemes have a bearing on response time. Below we discuss the two partitioning schemes used:

1. **Random Partitioning:** as the name indicates, we partition the data arbitrarily in to k layers, such that same edge is not repeated across multiple layers and all the layers have the same set of nodes.
2. **Edge based Partitioning:** In this scheme, we partition the data based on Edge labels, such that the edges with same edge label remains in the same layer, needless to say we can have multiple edge labels in a single layer if the edge labels are more than the number of layers, but they will be distinct across layers.

5.2 Adjacency List Generator

One of the most significant part of the substructure discovery algorithm is the Adjacency list, which aids the expansion process. We have to generate the adjacency list for each layer, as the expansion process in each layer is done independently.

Input to create our adjacency list is the input graph we use for our algorithm, we initialize a Hash Map, and iterate on all the edges in the graph and assign the vertex id as the key which exist in a particular edge and add all the edges in the value list which have that vertex id. Note: Each layer will have its own adjacency list and the connectivity of the graphs for a particular vertex id will be different in each adjacency list. We use a python script to create our adjacency list, but the Map/Reduce framework can be used if the size of the graph is too large. The parameters passed to our adjacency list generator is the number of layers for which we need to create our adjacency list.

5.3 Algorithm in the Map/Reduce Framework

Finding a substructure that best compresses the graph consists of: generating substructures of all sizes (starting from a single edge), eliminating duplicates, combining substructures across layers and counting isomorphic (identical) substructures in each iteration. This strategy reveals two grouping criteria among substructures: instances across different layers must be grouped for the combining phase and then the isomorphic instances needs to be grouped based (on their canonical substructure form) for ranking. As these two process are sequential, we can count the isomorphic substructures only after we have all the substructures from each layer and across layers, this necessitates the use of a two chained Map/Reduce job in each iteration of our algorithm. The first Map/Reduce job aids expansion of a substructure in each

layer, combines the *intralayer* substructures and remove duplicates while the second Map/Reduce job finds the top-k isomorphic substructures across layers with the best rank to be used as candidates for the next iteration.

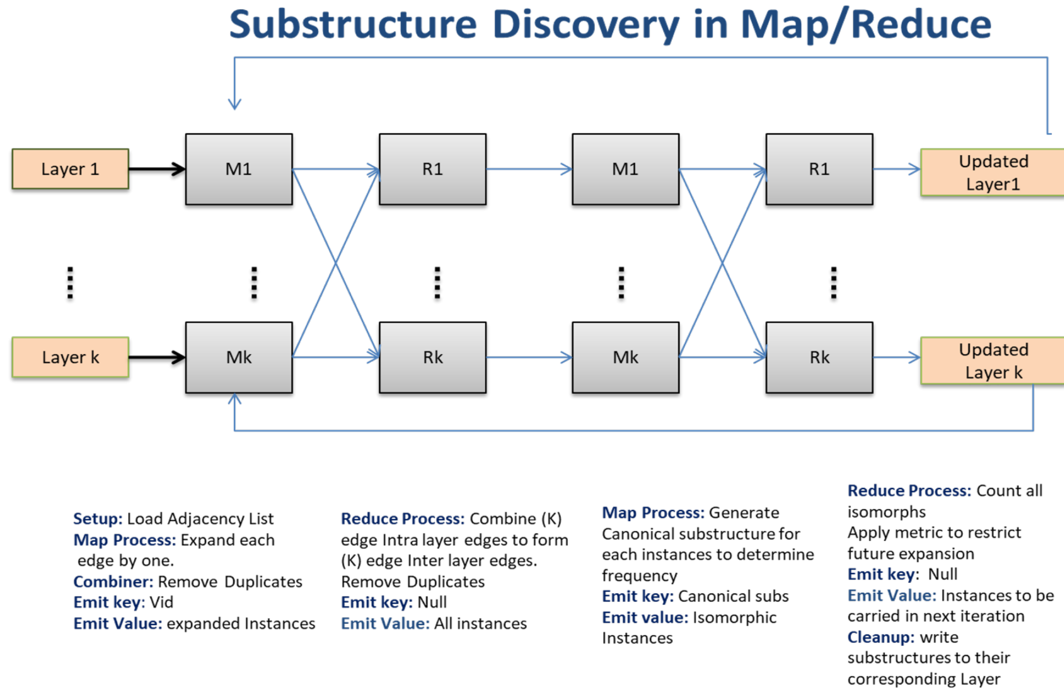

Figure 5.1: Workflow of Substructure Discovery in MLN

Figure 5.1 shows the overall flow of how substructure discovery in an MLN is done using the Map/Reduce framework. In Map/Reduce based substructure discovery, a mapper uses a single layer and its corresponding adjacency list to expand the substructures. The adjacency list of each layer is loaded in memory once in each mapper (using the setup method in Map/Reduce). Each mapper reads a single edge from the input layer and expands it by adding one edge in all possible ways as separate substructures. The expansion is unconstrained which leads to generation of duplicates which must be identified and removed. Typically the cost of removing

the duplicates is the same, whether it is done in the Mapper or the Reducer, but identifying and removing them in the Mapper decreases the amount of data shuffled (network cost) as those duplicates will not be sent to the Reducer. We remove the duplicates generated in the mapper using a combiner. After expansion and duplicate elimination, all the substructures are grouped in the reducer based on their vertex id for combining *intralayer* substructures to generate *interlayer* substructures. We use a Recursive (Combine-MLN) function, that checks for all the combinations and combines the instances across layers having a common vertex id. The reducer will emit all the k edge instances as values. The output of the first Map/Reduce job is written on to the disk which is used as the input to the second Map/Reduce job. In the intermediate Mapper, we generate canonical substructures from the canonical instances to determine graph isomorphism which are sent from mappers to the reducers, where all the instances from different mappers are grouped on their canonical substructure form to determine the frequency of exact substructures or isomorphs. A pruning metric like beam (or top-k) is then used to find the best substructures in that iteration, which will be used as candidates for the subsequent iteration.

Algorithm 2 First Map/Reduce Job for Substructure Discovery in HoMLN

Require: -

1: **Class Mapper**

2: **function** SETUP

3: Map = Load corresponding adjacency layer

4: **end function**

5: **function** MAP(key,value)

6: **for** each k-edge instance ks in value do:

7: **expand** ks to (ks+1) edge canonical instance

8: **for** each unique Vertex id in expanded instances do:

9: new key = unique vertex id

10: emit(new key, value = Expanded Instances) to combiner// to remove duplicates

11: **end for**

12: **end for**

13: **end function**

14: **Class Reducer**

15: **function Combine-MLN**(List of Layers L_s, i=Layer *id* to pick from each level, s= size of substructure, Vertex id)

16: **if** s == 0:

17: return Null

18: **if** L_s has one layer:

19: return all s size substructure from L_i

20: Merged-Instances = Global List to hold all combined substructures

21: L_{s1} = **Pop** Layer L_i from L_s to create new List of Layers

```
22:   for every size of substructure k← 0 to s:
23:     Intermediate-Instance-List = Combine-MLN(L_s1, i+1, s-k, v_id)
24:     New-Instance-List = Local List to hold all s size substructure at each step
25:     for all k size substructure S1 from Layer L_i:
26:       for all s-k size substructure S2 from Intermediate-Instance:
27:         if S1 and S2 has same Vertex id:
28:           Combine S1 and S2 to create New Substructure S_new
29:           if last added instance in New-Instance-List not same as S_new:
30:             add S_new to New-Instance-List
31:     Merged-Instances = Add all instances from New-Instance-List
32:   return Merged-Instances
33: function REDUCE(key, values) // key = vertex id, values = expanded instances
34:   L_s = Create list of layers from Value List
35:   k = Current size of Substructure in Value List
36:   Merged List = List to hold all combined substructures
37:   Call function Combine-MLN(L_s, 1, S=k, v) and return all instances to store
    in Merged List
38:   for each edge in Merged Lists:
39:     emit (key = Null, value = Combined instances)//(Interlayer)
40:   end for
41:   for each edge in Value list:
42:     emit (key = Null, value = k-edge instances) // Intralayer
43:   end for
44: end function
```

Expansion by Mapper: Algorithm 2 details our subgraph expansion routine in the Mapper. Each layer is read as Mapper input (one substructure instance at a time), while the adjacency list for the layer is loaded using the setup function and kept in memory. For the first iteration, the input key is the line no and the value is one edge instance. Similarly in the k-th iteration, the value is a k edge canonical instance. Line 3 loads the corresponding adjacency list for that layer. (line 6-7) expands each edge from the value by one edge and and (line 8-9) generates a new key (vertex id) from the expanded instances. Line 10 emits the expanded instances as values and vertex id as the key. As the expansion process is unconstrained we generate duplicates, which we remove using a combiner.

Combining by the Reducer: Each Reducer receives values list grouped on the Vertex id as the key. We generate a List of Layers, from the Value List at line 34 and use a Recursive utility, to combine all k-edge substructure to form a k-edge substructure. The utility will pick layer L_i for all s size substructure at each level and will find other size substructure from remaining layers recursively. The base case here is when we have only one layer left in list, in which case we return all substructures from that layer (line 18 and 19) or the size of substructure is zero, in which case we return empty list (line 16 and 17). Once the recursive call returns other size substructures (line 23), we combine them with all s size substructures from the picked layer (25 to 31). Here combining will be done only if both substructure have common vertex ids as key. The combined instances of all sizes will be then returned to parent call (line 32) and hence, all k size merges from all layers will be returned to the root call (line 37). All the Combined and IntraLayer substructures are then emitted to the Intermediate Mapper (line 38-42) for generating Canonical substructure to determine

graph isomorphism. The key is null and the values are all the substructures in the HoMLN.

Algorithm 3 Second Map/Reduce Job for Substructure Discovery in HoMLN

```
1:  Class Mapper
2:  function MAP(key,value) // k= line no, val= k-edge instances(Inter and Intra)
3:    vMap = hashtable to hold unique vertex info
4:    for each vertex id v in value do:
5:      vMap.put(v,null)
6:    end for
7:    update vMap positions from the value
8:      new key = generate Canonical substructures from value
9:      value = Canonical instance
10:     emit(new key, value)
11: end function
12: Class Reducer
13: function SETUP()
14:   Beam Map = Null // To store best substructures
15: end function
16: function REDUCE(key,values)//k= substructures, val = Isomorphic Instances
17:   C = Count(Instances in Values)
18:   MDL = Calculate mdl of each key using count of Instances
19:   Update Beam Map with MDL
20: end function
21: function Cleanup()
22:   for each instance in Values do:
23:     if instance has a single layer id: // Intra Layer
24:         key = null
25:         emit(null, Instance)
26:     end for
27:   end function
```

Algorithm 3 is our 2nd Map/Reduce job where we create Canonical substructures from the instances in the Mapper. Count their frequency and apply a metric to restrict future expansion in the Reducer.

Identifying Isomorphs in the Mapper:Creating Canonical substructures from instances requires a hash table to identify relative positioning of unique vertices (line 3). The Mapper receives all the instances as value and null as the key (line 2), We create the Canonical substructure for the Canonical instances in values and make that the key (line 4-8). Finally the mapper emits the canonical substructures as the key and the corresponding instance as value.

Frequency Counting by the Reducer: The reducer receives instances across mappers grouped on canonical substructure.We will use a notion of beam, which will be specified by the user to store the best instances, line 14 allocates a beam of size k as a hash map to hold MDL values and all the instances with that Value. Line 17-19 counts and finds the instances with top k MDL values to restrict the future expansion to high quality substructures in next iteration. The reducers emits the instances form each layer using the Layer id as values (line 22-25) to be used in the next iteration. .

5.4 Resource Utilization

As the goal of this thesis is to find interesting substructures in a Homogeneous Multilayer Network with a decent response time. In the ideal case we would want to process each layer on separate Mappers and Reducers to achieve best parallelism. But not always the number of Mappers equals the number of Layers in an MLN. Consider a scenario where we have a 60 Layer MLN, but only 10 processors (Mappers Reducers) available in the cluster. So the input to our algorithm will be 60 Layer/ Map tasks, and they will be processed on the 10 available mappers. So, 10 Map

tasks/Layers will be processed in parallel one at a time. Similarly the remaining 50 Map task will be processed in the same fashion. So on average, each Mapper will process 6 Layers. This approach would give us the correct output as Hadoop allows to process more Map tasks in a single Mapper one at a time, but this would hamper the response time, as each Mapper and Reducer will end up processing more data.

5.5 Input Parameters File

The input parameters file used in our implementation, defines our system and sets the condition of its operation. It has all the details required for the execution of our program. Following are the parameters used in the input parameters file.

- Input Path: Here we specify the HDFS path where the input graph is loaded.
- Graph Size: We specify the number of vertex and edges in the graph, this information is mainly used for MDL calculation.
- Max size: This specifies the maximum size of the substructure we want to generate, in general the number of iterations. In each iteration a substructure grows by one edge.
- Metric: We state the pruning metric here, which can be "Minimum Description Length" or "Frequency".
- Beam Size: As we are interested in finding frequent pattern in a graph, this parameter restricts the search space and specifies the number of frequent substructure to be used in the next iteration.
- Top N substructure: As we are interested in finding substructures of all sizes that best compresses the graph, it is imperative to keep a track of them re-

gardless of the iteration. As every iteration generate frequent substructures of respective sizes, this parameter states to keep number of top substructures after all the iteration. For example, if the parameter states to keep top "5" substructures, then in the first iteration it will write the top 5 substructures with best MDL/Frequency value in the file, and in the next iteration if any substructure has better MDL/Frequency value then currently in the file, it will overwrite them. So regardless of the iterations we run, the top N substructure file will always have the best substructure that best compresses the graph.

- Analysis: As we run multiple jobs with different configurations and want to keep the track of all the details of the job, this parameter specifies the path to store the Analysis file for each job.
- No of Reducers: This parameter states how many reducers are to be used for the job.

5.6 Controlling the Number of Mappers

Typically Map/Reduce is used for scalability problems, where the data is too large to be stored in the memory of a single processor. The performance of the MapReduce jobs depends on a lot of factors which needs to looked into detail as we are interested in processing large data sets with a decent response time. The relationship of MapReduce input split sizes and the input data is very important and they can work together to help (or hurt) job execution time. The default input split size is 128 MB which can be altered. So, for instance lets say we have a cluster with 5 processors which can be processed in parallel and the input data of 220 MB. If we 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 processors. But say you want to utilize all of your resources 5 processors to decrease your response time. Then we

can change the input split size to 45 MB, and it would create 5 Partitions (5 Map tasks) which can be processed in parallel and it will decrease the response time.
On Comet we specify the number of processors used as nodes and each node has 2 processors, so if we want 10 Mappers/Reducers we specify 5 nodes. But the map tasks should be same 10 in this case. If less than 10, 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.

5.6.1 Implementing Job Counters

We have used the built in counters and implemented few custom counters (user-defined) in each component of our Map/Reduce program, to measure the progress or the number of operations that occur within our Map/Reduce job. Counters in Hadoop MapReduce are a useful channel for gathering statistics about the MapReduce job: for quality control and to analyse the cost and space. Below is the list of all Job counters used.

- Map Setup time: Setup method as the name indicates is used to setup the map task and is used only once at the beginning of the task. All the logic needed to run the task is initialized in this method. Implementing a counter for setup time helps to analyse the setup cost for a Mapper/Reducer. As we have a pair of two Map/Reduce jobs, we can analyse the setup time for each Mapper/Reducer. The main cost of setup is incurred in the first Mapper, as we load the adjacency list for a layer whereas the setup time for the second method is negligible as it only requires to declare the data structures used in following Map task.
- Map time: The Map method is the most important method and is called once for every key/value pair in the input to the mapper task. Implementing the counter here will help to inspect the cost associated with our Map method. The

Map time associated with our algorithm comprises of expanding substructures in each layer and creating Canonical substructures from the all the instances. As we increase the number of mappers for the same graph size, we shall observe a decrease in the Map time as less data will be processed by each mapper. The Map time is calculated by taking maximum of all Mappers Map time.

- Shuffle time: In the shuffle phase data from the mapper are grouped by the key, split among reducers and sorted by the key. Every reducer obtains all values associated with the same key. Shuffle and sort phase in Hadoop occur simultaneously. We have implemented a counter here to examine the total time taken by the shuffle phase to transmit the data from the Mapper to Reducer.
- Reduce time: The Reduce method is called for each $< key, (listofvalues) >$. This counter details the total time take by our Reduce method. Reducer time comprises of Combining substructures across layer, counting the isomorphic instances and applying metric to restrict future expansion, using a counter here is imperative as most of the work of our algorithm is carried out in the reduce phase.
- Cleanup time: Cleanup is the method which is called only once at the end of each task. This method is responsible for the cleanup of the task residues if any.

The next chapter discusses the experimental analysis of our algorithm for correctness and scalability.

CHAPTER 6

EXPERIMENTAL ANALYSIS

In this section we experimentally evaluate our algorithm. All the experiments are performed using Java with Hadoop on Comet cluster at SDSC (San Diego Supercomputer Center). The Comet Cluster has 1944 nodes and each node has 24 cores (built on two 12-core Intel Xeon E5 2.5 GHz processors) with 128 GB memory, and 320GB SSD for local scratch space.

6.1 Experimental Setup

We experiment on several real world and synthetic datasets to establish the effectiveness, correctness, speedup with respect to different configurations of mappers and reducers, and scalability of our approach. Table 6.1 shows the data sets we have used for our experiments. We have used **SUBGEN** an synthetic graph generator for a few of our experiments as it allows to embed small graphs with user defined frequency in a single graph. Subgen also provides us with better control in generating graphs of different sizes and characteristics using parameters.

6.2 Empirical Correctness

The correctness and efficiency of the algorithm is verified by running SUBDUE [6], a main memory approach and our algorithm on the same dataset with multiple layers. After running our algorithm and SUBDUE on a graph with 10KV_20KE,

What For	Dataset	#Nodes	#Edges
Correctness	Synthetic (SUBGEN)	100KV	800KE
Multiple partitioning	Synthetic (SUBGEN)	400KV	1.2ME
Scalability	LiveJournal	3.9MV	34.8ME
Scalability	Orkut	3.87MV	114.8ME
Varying Density	Synthetic(SUBGEN)	2KV	1ME, 1.9ME, 2.9ME,3.9ME

Table 6.1: Data Sets Used in Experiments and their Sizes

we discovered the same set of substructures. But SUBDUE being a Main memory approach, failed to compile with increasing sizes of graphs of more than 100KV_800KE. So to verify the correctness on large graphs we embed substructures with a user defined frequency, and the goal is to find the same substructures.

Figure 6.1 shows the 5-edge embedded substructures with a frequency of 1000 in a graph of size 100KV_800KE. We found the exact substructure after running our algorithm with a frequency of 976, when we ignored the overlaps and a frequency of 1000 after counting the overlaps which validated the correctness of our algorithm empirically for this dataset.

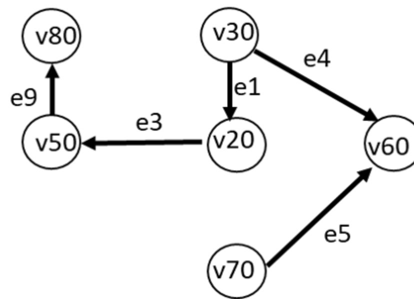

Figure 6.1: 5-edge Embedded Substructure

6.3 Multiple Partitioning Schemes

This set of experiments is performed using a synthetic dataset of size 400KV_1.2ME with embedded substructures. We used two partitioning schemes that were discussed earlier, to verify the correctness of our algorithm and to monitor the bearing of partitioning schemes on response time.

Layers	**Random**	**Edge-Based**
Layer1	399903	351360
Layer2	399730	478441
Layer3	400637	370469

Table 6.2: Edge Distribution

Table 6.2 shows the edge distribution for both the partitioning schemes, we see that the distribution remains even for Random partitioning, but becomes skewed for Edge based as their can be edge labels with higher frequency which goes into a single layer making it uneven. So now lets have a look at the response time. Figure 6.2 shows the total time taken by both the partitioning schemes. We don't see a substantial difference in the total response time, it more or less remains the same, so we inspect the breakup analysis of the Map and Reduce time to understand it clearly.

As we see in Figure 6.3, the Map time for edge based partitioning is significantly higher when compared to the Map time for random partitioning. The reason being, as the edge distribution is skewed, the Mapper ends up processing more data for edge based partitioning, contributing to more computation time. On the other hand, response time for reducer doesn't change as the data processed by each reducer remains the same because all the substructures in the reducer are grouped based on their Vertex id and edge labels have no effect on them.

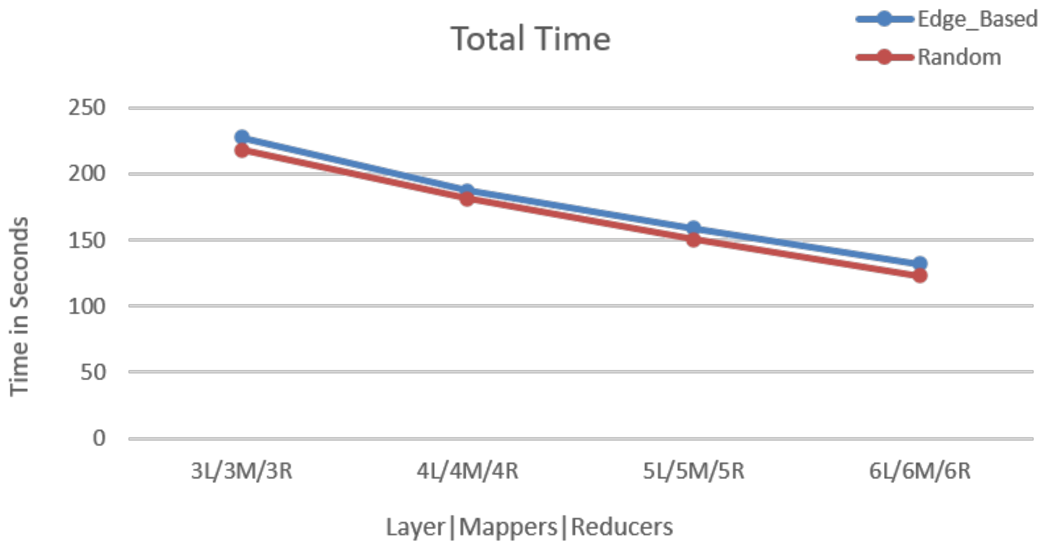

Figure 6.2: Total response time using different partitioning schemes

We found the same embedded substructure for both the partitioning schemes which indicates that our algorithm doesn't depends on the connectivity of the graph.

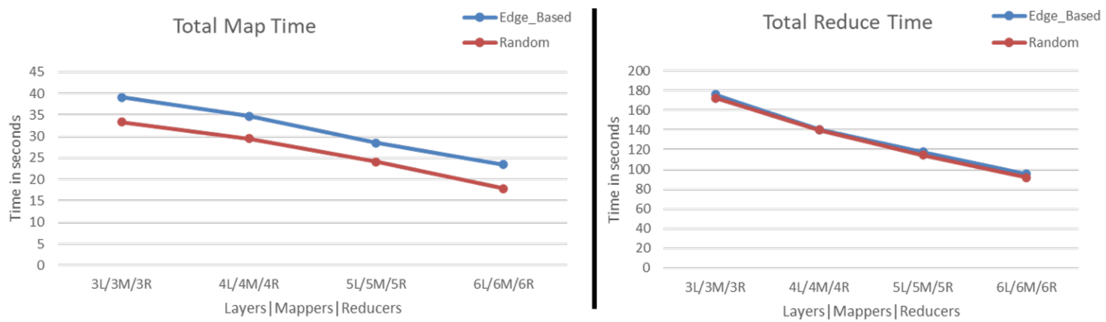

Figure 6.3: Map/Reduce time using multiple partitioning schemes

6.4 Scalability

Without altering the graph size, an increase in the number of processors is typically beneficial for mining. So, this set of experiments were performed on Live-

Journal [25] and Orkut [26] data to determine the speed up and effectiveness of our algorithm as we increase the number of processors. We further divide this section into 3 scenarios.

6.4.1 **Same number of Mappers and Reducers as Layers**

For this scenario, all the layers are processed in parallel with the same number of Mappers and Reducers. Our results in Figure 6.4 shows a speed up of 37.4% when we increase the increase the number of layers and mappers/reducers from 8 to 16 and 35.6% when we further increase them from 16 to 32. The reason for the speedup is, as the same data set is partitioned into more layers and is processed by more number of processors, leading to smaller sized partitions and less computation in each processor. This reduction in computation cost contributes to the speedup achieved. We see that in Figure 6.5, that the amount of speed up achieved by the mapper is more than the reducer when we double the processors for both, because the amount of data processed in the 2nd reduce job, is not necessarily halved. As each reducer emits the instances with local top-k MDL values to find the instances with global top-k best MDL values, so as we double the reducers the mining time for the reducer is not exactly halved. Moreover, as we double the number of Layers and reducers, the data received by each reducer gets halved, but as the number of layers increase the mining cost in the 1st reduce job also increases as the height of the tree grows leading to more number of possible *Interlayer* combinations.

Our results, in Figure 6.4 shows that the total time taken by beam size 10 is more as compared to beam size 6, because as we increase the beam, the number of substructures carried in the each iteration increases, which can be seen in Table 6.3

resulting in more data to be processed by each processor contributing to more computation cost, hence more response time.

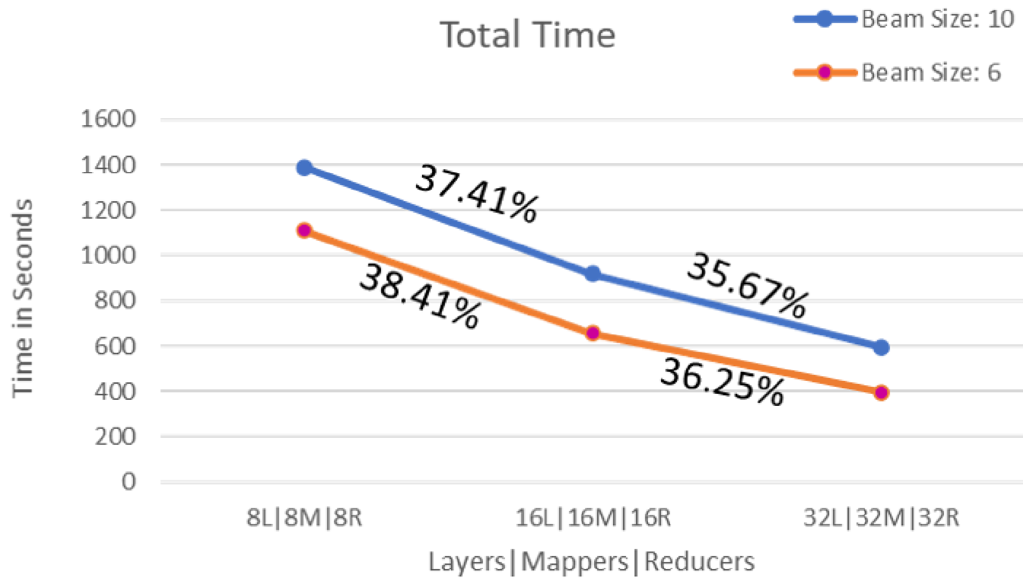

Figure 6.4: Speed up achieved on LiveJournal Data

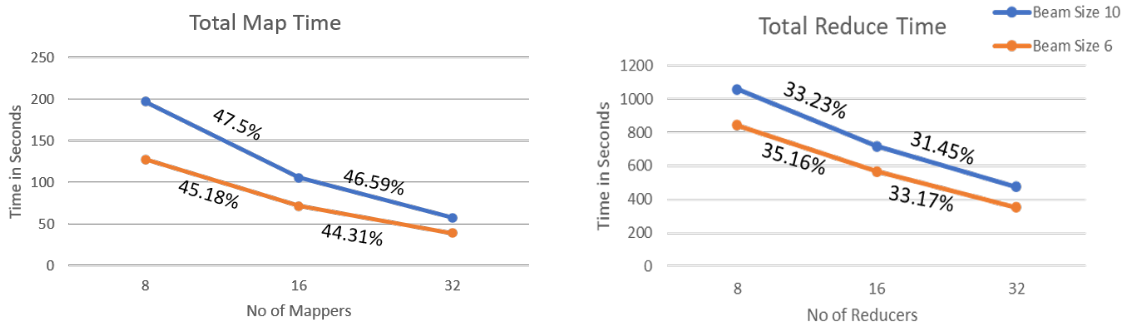

Figure 6.5: Map and Reducer time for LiveJournal

Iterations	Beam 10	Beam 6
1	2836928	2836928
2	3929240	3172184
3	4593864	3628496
4	5249968	3849008
5	5346152	4004984
6	5716984	4128496

Table 6.3: No of substructures generated in each iteration

6.4.2 Effect on Response Time With change in Reducers

This experiment was performed on LiveJournal and Orkut dataset with 32 Layers. In this scenario, we keep the number of mappers same and change the number of reducers to see the effect of reducers on response time.

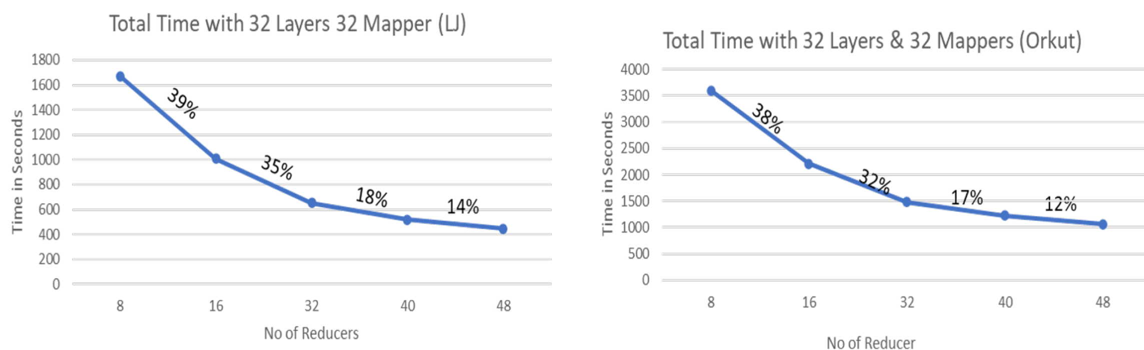

Figure 6.6: Total Time with change in Reducers

Our results in Figure 6.6 shows the speed up of more than 37% for both Live-Journal and Orkut when we increase the reducers from 8 to 16 and more than 32% from 16 to 32. But after that the amount of speed up achieved starts to decrease. Though, it does gives us a speedup of around 17% when we increase the reducers from 32 to 40, but again decreases to 14% for 48 reducers. So giving more number

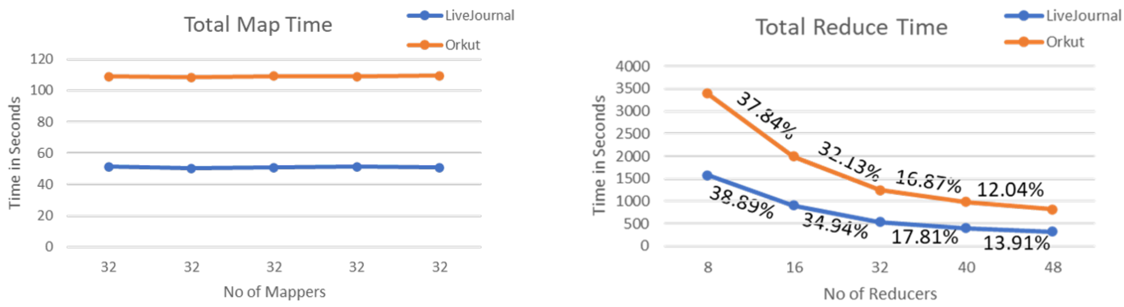

Figure 6.7: Map/Reduce time with change in Reducers

of reducers does not always guarantee decent speedup, as law of diminishing returns and adding additional factor (Reducers) will result in smaller decrease in response time. Orkut takes more absolute time in general as it is has more than 3 times the number of edges for almost the same number of vertices.

We see in Figure 6.7, that response time for the mappers remains constant for both Orkut and LiveJournal beacause the same amount of data is processed by the mappers as they remain the same. But the reduce time changes for both the data sets which corresponds to the change in total time as shown in Figure 6.6

6.4.3 Effect on Response time With change in Mappers

This experiment was performed on LiveJournal and Orkut dataset with 32 Layers. We keep the number of reducers same in this scenario but change the number of mappers to inspect the effect of mappers on the response time.

Results in Figure 6.8 demonstrates that increasing the number of mappers doesn't contribute to much speedup, as most of the work is done by the reducer in our algorithm. However if we just analyse the Map time for this experiment shown in Figure 6.9 we see a significant speedup , but when combined together to calculate overall time, the performance is not significant as the reducers remain the same 32.

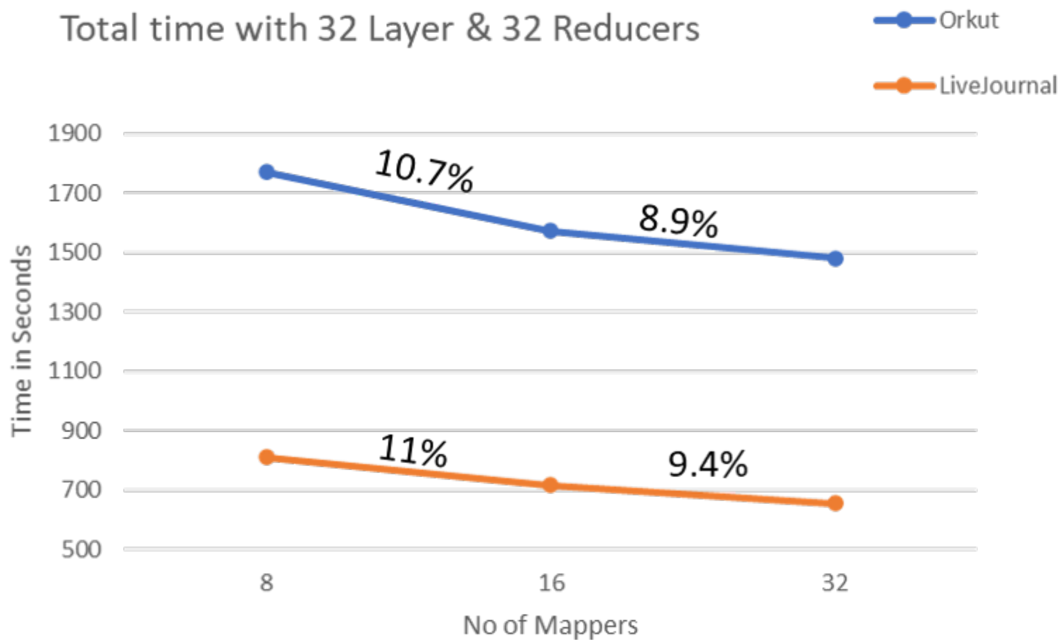

Figure 6.8: Total time with w.r.t change in Mappers

If we increase the number of mappers more than 32, it would not show any speed up, as they would not utilized by the framework because our algorithm does not partitions individual layers. So the maximum number of mappers used should not be more than the number of layers in an MLN.

Orkut takes more absolute time in general as it is has more than 3 times the number of edges for almost the same number of vertices, which leads to more expansion in the mapper and *Intralyer* combining in the reducer contributing to more mining time.

6.5 Effect of Graph Connectivity

Connectivity of graphs also influences the performance of mining algorithms. We categorize graphs as dense to sparse, where the number of vertices are fixed but the number of edges vary in the spectrum ranging from a completely connected graph

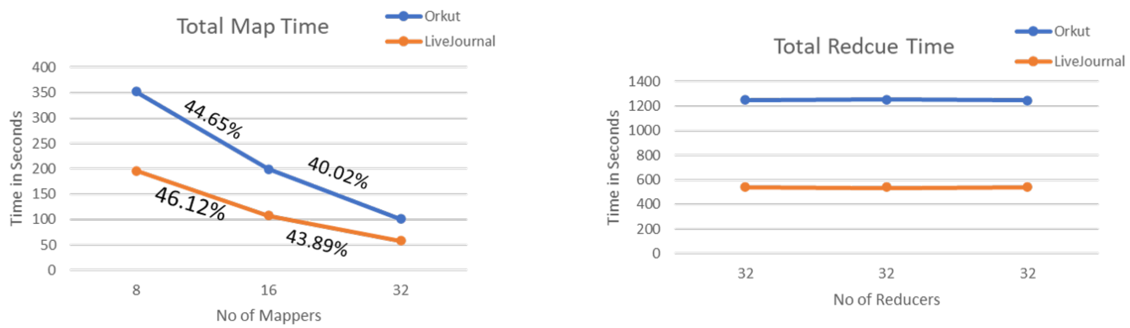

Figure 6.9: Map/Reduce time with w.r.t change in Mappers

to a very sparsely connected line graph. We performed this experiment on 4 Layers using 4 mappers and 4 reducers to see the effect of connectivity of graphs on response time. Our results in Figure 6.10 shows that with dense graphs, where each vertex is connected to more vertices on the average, the computation cost increases as there are more expansions which leads to more map time and more combinations of *intralayer* edges in the reducer contributing to more reduce time.

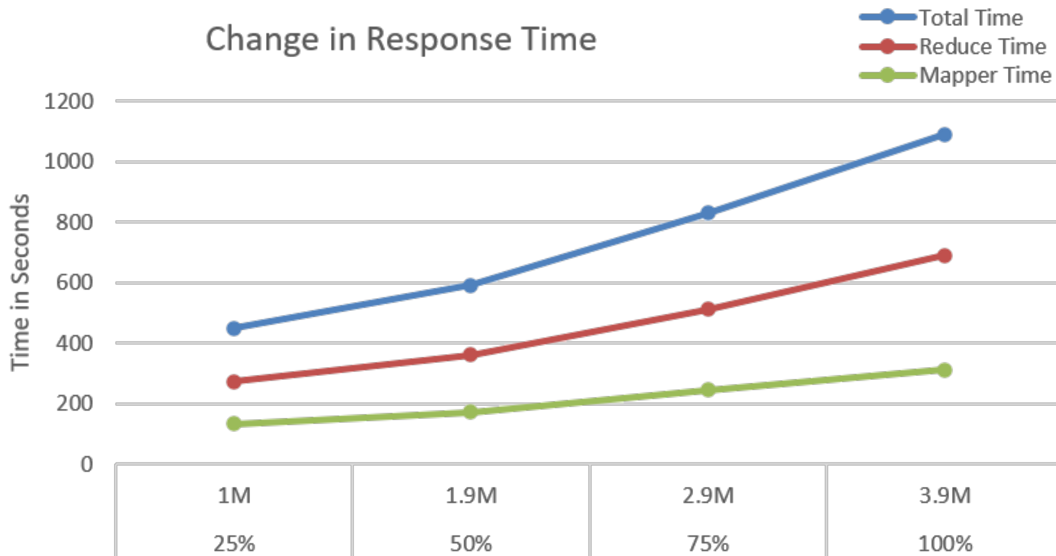

Figure 6.10: Effect of density on Response time

CHAPTER 7

CONCLUSION AND FUTURE WORK

This thesis proposes a scalable substructure discovery in a Homogeneous Multilayer Network using the decoupling based strategy. We identified a set of problems and addressed it as a part of this research. This thesis introduced generic Map/Reduce based algorithm for horizontal scalability of substructure discovery that works over a HoMLN. The basic components of graph mining - subgraph generation, combining substructures in each layer to generate substructures across layers, duplicate elimination and counting of isomorphic substructures were incorporated into the algorithms for the Map/Reduce paradigm. Experiments validated the benefits of using Map/Reduce based substructure discovery to scale to large graphs with multiple layers.

Partitioned graph mining is a promising research field and provides exciting future directions. Our decoupling approach can be extended to a Heterogeneous MLN and can be modelled to work on a different distributed environment such as Spark [27]. Partitioning the layers of MLN can be addressed further to accommodate the increasing sizes of graphs.

REFERENCES

[1] R. Agrawal, R. Srikant, *et al.*, "Fast algorithms for mining association rules," in *Proc. 20th int. conf. very large data bases, VLDB*, vol. 1215, 1994, pp. 487–499.

[2] S. Boccaletti, G. Bianconi, R. Criado, C. del Genio, J. Gómez-Gardeñes, M. Romance, I. Sendiña-Nadal, Z. Wang, and M. Zanin, "The structure and dynamics of multilayer networks," *Physics Reports*, vol. 544, no. 1, pp. 1 – 122, 2014.

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

[4] B. Boden, S. Günnemann, H. Hoffmann, and T. Seidl, "Mining coherent subgraphs in multi-layer graphs with edge labels," in *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '12. ACM, 2012, pp. 1258–1266. [Online]. Available: http://doi.acm.org/10.1145/2339530.2339726

[5] A. Santra, S. Bhowmick, and S. Chakravarthy, "Efficient community re-creation in multilayer networks using boolean operations," *Procedia Computer Science*, vol. 108, pp. 58–67, 2017.

[6] N. S. Ketkar, L. B. Holder, and D. J. Cook, "Subdue: Compression-based frequent pattern discovery in graph data," in *Proceedings of the 1st international workshop on open source data mining: frequent pattern mining implementations*, 2005, pp. 71–76.

[7] S. Das and S. Chakravarthy, "Partition and conquer: map/reduce way of substructure discovery," in *International Conference on Big Data Analytics and Knowledge Discovery*. Springer, 2015, pp. 365–378.

[8] S. Das *et al.*, "Divide and conquer approach to scalable substructure discovery: Partitioning schemes, algorithms, optimization and performance analysis using map/reduce paradigm," Ph.D. dissertation, 2017.

[9] W. Lin, X. Xiao, and G. Ghinita, "Large-scale frequent subgraph mining in mapreduce," in *2014 IEEE 30th International Conference on Data Engineering*. IEEE, 2014, pp. 844–855.

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

[11] R. N. Chittimoori, L. B. Holder, and D. J. Cook, "Applying the subdue substructure discovery system to the chemical toxicity domain." in *FLAIRS Conference*, 1999, pp. 90–94.

[12] S. Djoko, D. J. Cook, and L. B. Holder, "An empirical study of domain knowledge and its benefits to substructure discovery," *IEEE Transactions on Knowledge and Data Engineering*, vol. 9, no. 4, pp. 575–586, 1997.

[13] L. B. Holder, "Empirical substructure discovery," in *Proceedings of the sixth international workshop on Machine learning*. Elsevier, 1989, pp. 133–136.

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

[15] D. J. Cook and L. B. Holder, "Substructure discovery using minimum description length and background knowledge," *Journal of Artificial Intelligence Research*, vol. 1, pp. 231–255, 1993.

[16] A. Inokuchi, T. Washio, and H. Motoda, "An apriori-based algorithm for mining frequent substructures from graph data," in *European conference on principles of data mining and knowledge discovery*. Springer, 2000, pp. 13–23.

[17] W. Wang, C. Wang, Y. Zhu, B. Shi, J. Pei, X. Yan, and J. Han, "Graphminer: a structural pattern-mining system for large disk-based graph databases and its applications," in *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, 2005, pp. 879–881.

[18] S. Chakravarthy, R. Beera, and R. Balachandran, "Db-subdue: Database approach to graph mining," in *Pacific-Asia Conference on Knowledge Discovery and Data Mining*. Springer, 2004, pp. 341–350.

[19] S. Padmanabhan and S. Chakravarthy, "Hdb-subdue: A scalable approach to graph mining," in *International Conference on Data Warehousing and Knowledge Discovery*. Springer, 2009, pp. 325–338.

[20] G. Liu and L. Wong, "Effective pruning techniques for mining quasi-cliques," in *Joint European conference on machine learning and knowledge discovery in databases*. Springer, 2008, pp. 33–49.

[21] 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.

[22] 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.

[23] 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.

[24] Hadoop. [Online]. Available: http://hadoop.apache.org.

[25] Live-journal. [Online]. Available: http://snap.stanford.edu/data/com-LiveJournal.html.

[26] Orkut. [Online]. Available: http://snap.stanford.edu/data/com-Orkut.html.

[27] A. Spark, "Apache spark," *Retrieved January*, vol. 17, p. 2018, 2018.

BIOGRAPHICAL STATEMENT

Anish Rai was born in Dhanbad, Jharkhand, India in 1995. He received his Bachelor of Technology degree in Information Technology from Dehradun Institute of Technology, India in May 2017. His interest in Big Data led him to pursue Master's in Computer Science at the University of Texas at Arlington in Fall 2018. He has served as a Graduate Teaching Assistant at the University of Texas at Arlington from August 2019 to May 2020. He received his Master of Science in Computer Science from the University of Texas at Arlington in May 2020. His research interest includes Machine Learning and Big Data Engineering and Analytics.