DIVIDE AND CONQUER APPROACH TO SCALABLE SUBSTRUCTURE DISCOVERY: PARTITIONING SCHEMES, ALGORITHMS, OPTIMIZATION AND PERFORMANCE ANALYSIS USING MAP/REDUCE PARADIGM

by

SOUMYAVA DAS

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

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT ARLINGTON

May 2017

To my parents who set the example and who made me who I am.

ACKNOWLEDGEMENTS

I would like to express sincere gratitude to my advisor Dr. Sharma Chakravarthy for providing valuable guidance and support during the course of my doctoral work. His keen interest and enthusiasm in teaching and his pedagogical methods truly inspired me. It is the same type of enthusiasm that he devoted into advising that helped in making me a true researcher. He has spent a tremendous amount of time training me in research, writing, and giving professional talks. He has been brilliant and insightful in research discussions. His dedication, persistence, and sincerity in research deeply impressed me and have set up the high standards that I want to maintain in my future career.

I am also extremely grateful to David Levine. He has been a great source of encouragement during the last five years. With his teaching methodologies, care for students and everlasting smile he has become an immediate role model for me. I want to thank my thesis committee members Dr. Gautam Das and Dr. Leonidas Fegaras for their interest in my research, for taking time to serve in my dissertation committee and for their comments, suggestions, guidance and help at the time of need.

I would also like to thank Dr. Ramez Elmasri and Dr. Bahram Khalili for all their support, encouragement and guidance during the years I have been in UTA as a graduate teaching assistant and an instructor. In addition, I would like to acknowledge the assistance I received, from Pamela McBride, Irie Bito, Sherri Gotcher as well as the entire staff of the Computer Science and Engineering Department at UTA, during the course of my doctoral work.

I wish to thank all my colleagues (past and present) at Information Technology Lab (ITLab) for their support and encouragement and for making the stay at ITLab over the last five years thoroughly exciting. I am also grateful to my friends for their interest in my research and for their helpful discussions and invaluable comments. My appreciation especially goes to Sujoy, Sarvani, Yuanzhe, Praveen (and his family), Mahashweta, Puloma, Subhash, Manish, Chetan, Swanand, Abhishek, Souvik, Arka, Sourav and Daipayan for providing me with generous support and countless fun throughout the ups and downs of graduate life at UTA. My appreciation also goes out to my childhood friends Partha, Dhrubojyoti, Ushnish and Sameek for being there for me. I am also grateful to all the professors and mentors who have helped me throughout my career, both in India and United States.

Finally, I am grateful to my parents Dr Pranati Das and Dr. Rabindra Nath Das for always inspiring me and inculcating in me a love for academics. A huge vote of thanks to my amazing cousins and my other family members for their endless love and support. Without their encouragement and endurance, this work would not have been possible. Last but not the least, I express my appreciation for the goodness of an overriding providence in my life.

April 24, 2017

ABSTRACT

DIVIDE AND CONQUER APPROACH TO SCALABLE SUBSTRUCTURE DISCOVERY: PARTITIONING SCHEMES, ALGORITHMS, OPTIMIZATION AND PERFORMANCE ANALYSIS USING MAP/REDUCE PARADIGM

SOUMYAVA DAS, Ph.D.

The University of Texas at Arlington, 2017

Supervising Professor: Sharma Chakravarthy

With the proliferation of applications rich in relationships, graphs are becoming the preferred choice of data model for representing/storing data with relationships. The notion of "information retrieval" and "information discovery" in graphs has acquired a completely new connotation and are currently being applied to a wide range of contexts ranging from social networks, chemical compounds, telephone networks to transactional networks. From the point of view of an end user, one of the most important aspects on graphs is to discover recurrent patterns following user-defined parameters. Finding frequent patterns play an important role in mining associations, correlations and many other interesting aspects among data.

The evolution of web 2.0 has propelled growth of graphs at an unprecedented rate. These graphs have hundreds of millions of entities and their interactions needing tens to hundreds of GBs of storage. Data processing for finding frequent patterns on these graphs generate huge intermediate result sets. Neither these graphs nor the intermediate results can be materialized on a single machine. Even if we have access to

powerful machines to scale vertically, state-of-the art methods for frequent subgraph mining requires enormous amount of computing resources causing even powerful machines to crash at times. Therefore, development of techniques that scale horizontally and effectively with increasing graph sizes is necessary. This dissertation addresses research in that direction by designing scalable graph mining techniques. Although scalability has been the main concern while developing approaches and algorithms, their correctness, elegance, and efficiency on large-scale graphs is maintained.

Until now, graph mining has been addressed using main memory, disk-based as well as database-oriented approaches to deal with progressively large-sizes of applications. This dissertation starts with the problem of substructure discovery by dividing the graph into smaller partitions and then combining the results across partitions effectively. Two algorithms, based on two partitioning strategies are introduced which cast a main memory approach (Subdue [1]), along with its nuances into a distributed framework. Map/Reduce has been used as a distributed paradigm here. The basics of graph mining such as systematic expansion and computing graph similarity have been elegantly translated to the Map/Reduce paradigm. The overall focus is to address scalable mining techniques on partitioned graph using a cluster of (heterogeneous) commodity machines.

In the process of mapping the mining algorithm to a distributed environment, some of the nuances of the existing algorithm are propagated to the distributed paradigm. For example, now intermediate results are generated within each partition which need to be handled across partitions. A lot of these intermediate results are duplicates when different graphs grow into multiple copies of the same bigger graph. Elimination of duplicates is critical not only for correctness but also for reducing the mining cost (i.e., performance and speedup.) The next part of the dissertation introduces a set of optimizations over the existing iterative algorithm (both in single

machine and map/reduce environment.) These optimizations aim to reduce duplicate generation by introducing heuristics based on graph characteristics. Irrespective of the choice of the heuristics, these optimizations improve response time and storage cost of graph mining.

The dissertation finally continues to examining different paradigm specific costs for our partition-based graph mining. Graph partitioning is a widely researched problem where the motive is to minimize inter partition connectivity. However graph partitioning has been predominantly used for query answering purpose in graphs. This dissertation outlines the limitations of a state of the art partitioning scheme for substructure discovery and introduces two new partitioning strategies for graph mining. Mining in the presence of partitions incurs computation cost in each partition and network cost of grouping results across partitions. We analyze the cost of partition-based graph mining for our partitioning schemes thereby leading to evaluating the choice of initial partitioning for substructure discovery. Usability of these partitioning techniques for three different classes of graph mining problems (non iterative, fixed cost iterative and variable cost iterative) have been provided to postulate our observation that one partitioning scheme does not fit all. Finally, the dissertation elaborates the applicability of our partition-based techniques on a different paradigm (Spark) to justify the benefit of our algorithm design over any distributed paradigm.

The effectiveness of the techniques proposed in this dissertation are validated with extensive experimental analysis on 2 real world graphs (Live Journal [2] and Orkut [3]) and 2 synthetic graphs and their variations(generated using the Subgen [4] and RMAT [5] artificial graph generators.)

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

Data mining has been an active topic of research for quite some time with most effort being focused on discovering association rules from transactional data. A popular example is the *market-basket analysis* where the goal is to discover interesting and repetitive patterns (or frequently purchased items) in transactional data. However traditional transactional mining techniques are rendered ineffective in data with inherent relationships. Such data are typically modeled as graphs. The ability to mine over graphs is important as graphs are capable of elegantly capturing complex relationships.

Graphs have become increasingly important in modeling sophisticated structures and their interactions, with broad applications including chemical informatics, bio-informatics, computer vision, video indexing, text retrieval and web analysis. A common thread that binds these applications together is the necessity to discover repetitive and interesting patterns (referred to as substructure discovery or graph mining.) Although many approaches for graph mining have been proposed in literature, advancing the frontier of graph mining needs to concentrate on:

1. Appropriate techniques for substructure discovery on graphs that scale with increasing graph sizes to any arbitrary graph size
2. Analysis of distributed paradigm for scaling horizontally and developing techniques to port mining algorithms to chosen one and other distributed paradigms
3. Analysis of partitioning schemes for graphs and detailed analysis of components costs of graph mining for a partitioning scheme over a distributed paradigms

4. Suitable optimizations and establish their correctness to improve response time, work done targeting all components costs including the storage cost.
5. Where possible, derive generalizations and provide heuristic-based guidelines (based on broader graph characteristics) for choosing partitioning schemes, best optimization option, and storage alternatives

This dissertation advances research in the direction of scalable substructure discovery (also called graph mining) by answering the above challenges.

1.1 Substructure Discovery in Graphs

Frequent patterns are itemsets, subsequences, or substructures that appear in a data set with no less than a user-specified criteria. For example, a set of items, such as milk and bread, that appear frequently together in a transaction data set, is a frequent itemset. Similar to an itemset, a substructure in a graph can refer to different structural forms, such as subgraphs, subtrees, or sub lattices. If a substructure occurs frequently in a graph database, it is called a (frequent) structural pattern.

Substructure discovery is the process of frequent pattern mining in graphs (or a forest). Finding frequent patterns in graphs plays an essential role in mining associations, correlations, and many other interesting relationships among data. The frequent subgraphs help in graph indexing, classification, clustering, and other data mining tasks on graphs as well. Thus, frequent pattern mining has become a fundamental data mining task and a focused theme in graph mining research.

Recurrent patterns also separate anomalies from interesting data. For example, repetitive patterns in a transactional network (banking) can be used to separate frequent transactions from fraudulent ones. Frequent patterns in social networks can be used to help publishers in selective marketing and advertising thereby increasing revenue. In biological networks and chemical compounds frequent patterns help validate

the approved diversity of chemicals (or drugs.) In telephone networks, recurrent calling patterns can be used to introduce customized calling plans and also improve user experience. Figure 1.1 shows an example graph along with the repetitive patterns.

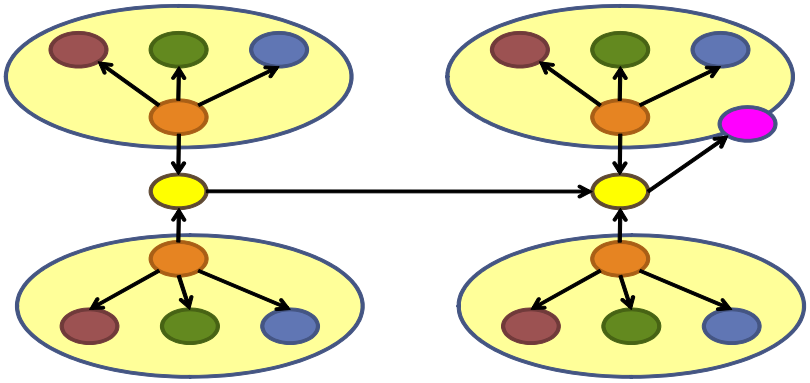
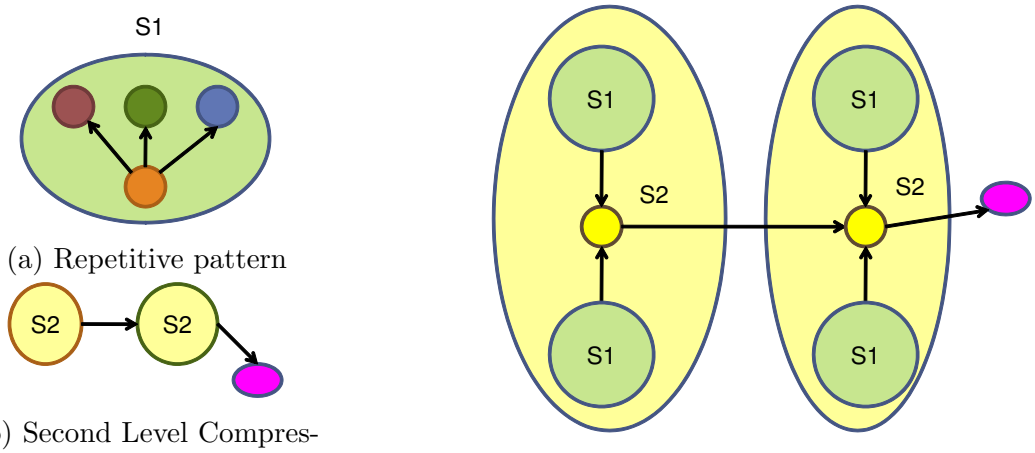


Figure 1.1: An example graph

An Interesting Anecdote: The frequent patterns can also be used to compress a graph and save storage space. Big graphs can be coarsened to small graphs using these compression technique and main memory based mining techniques can be applied on the compressed graph to discover progressively bigger (or even hierarchical) patterns. Figure 1.2 shows how the frequent pattern in Figure 1.2a can be used to compress the graph in Figure 1.1 to Figure 1.2c. The compressed graph can be again compressed hierarchically to Figure 1.2b. Graphs larger than main memory can be compressed using this technique to be loaded in main memory when conventional main memory based techniques can be used for mining.

Existing literature considers two kinds of graph databases: transactional and single graphs. The *transactional* case assumes a large number of relatively smaller sized graphs. An example being a database of chemical compounds, where each graph represents a connection of hundreds on elements. A substructure is called frequent if it appears in more than τ number of graphs where τ is a user-defined parameter.



(a) Repetitive pattern

(b) Second Level Compression Using S2

(c) Compressed Original Graph Using S1

Figure 1.2: Frequent pattern in graphs aiding in compression

The single graph is a generalization of the transactional approach since a large graph can also be considered as an agglomeration of smaller independent components. This dissertation focuses on mining on single graphs. Substructure discovery in a large single graph is even more complicated as patterns might overlap with each other. Also, the complexity is exponential in graph size making things even more challenging.

With the advent of Web 2.0 and a worldwide reach of hundreds of millions of users, graphs are growing at an astonishing pace. For example LiveJournal [2] has 4 Million users and 34.68 Million interactions among them and takes 2.5GB of disk space. Another social media graph, Orkut [3] has 4.03 Million users and 117 Million interactions among them using approximately 8GB of disk space. Other big-scale networks such as Facebook [6] and LinkedIn [7] already have close to billions of users and consume TBs of disk space. Graph mining on such big graphs face the following issues:

- The graphs are too large to be loaded into main memory of a single machine
- Operations on graphs generate huge intermediate result set easily overflowing main memory.

Even if we have access to powerful machines, literature [8] shows that a state-of-the art method for frequent subgraph mining crashes after a day consuming $192GB$ for an input graph of only $100K$ nodes and $1M$ edges. Therefore the development of techniques that can mine large graphs even with limited computational capability is very crucial.

1.2 Scalability of Substructure Discovery

Substructure discovery is already a widely researched area. However discovery of repetitive substructure patterns has become difficult with increasing graph sizes which renders earlier approaches (from main memory [1,9–12] to disk-based [13–15] to database-oriented approaches [16, 17]) ineffective. Hence there exists a clear need for developing novel techniques for substructure discovery in big structural datasets. In order to analyze graphs of increasing sizes and still discover interesting substructures within a meaningful response time, there is a need for mining using a paradigm that scales with increasing graph sizes.

An intuitive approach is to adapt the time-tested *divide and conquer* technique for graph mining. The idea is to partition the graph into smaller chunks, mine across partitions and then combine findings across partitions. There have been some effort at partitioned approaches to substructure discovery [18, 19] but they do not discuss expansion of substructures across partitions hence compromising accuracy (or get approximate results.) Preserving accuracy in the presence of partitions entails handling substructures that cross partition boundaries necessitating exchange of information across processors. The focus therefore shifts to incorporate correctness into the chosen distributed paradigm to address graph mining.

Map/Reduce is one distributed paradigm that has been used to scale computations horizontally to accommodate very large data sizes. This paradigm also handles

heterogeneous as well as different generation computing devices. However, we have to work within the specified functionality provided by the paradigm, such as partition, compute, merge, and synchronize computations using a programming framework introduced by Google [20]. Map/Reduce, with its open source implementation Hadoop [21] provides a powerful programming framework to process large data sets. Hadoop separates the underlying complexities from the user by providing a couple of methods (or functions) to interact with the system. It also automatically handles scheduling and failure thus allowing us to focus only on using the functions to our advantage. However, for using Hadoop, mining algorithms have to be cast efficiently into this paradigm.

Currently, an iterative algorithm is used for substructure discovery 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 the rank) is carried forward to constrain the expansion process. The steps of iteratively generating bigger substructures (graph expansion) and counting identical substructures (graph similarity) needs to be remodeled to work correctly on partitioned graphs. Ideally, each computation in each partition should not be dependent on other graph partitions, as that beats the purpose of parallelization.

We draw inspiration from Subdue [1], one of the earliest graph mining technique for designing our mining algorithms on the distributed paradigm. Subdue can handle graphs with loops, cycles, multiple edges, directed and undirected edges making it one of the best general purpose substructure discovery techniques. To make graph mining scalable, in this dissertation, we design mining techniques (inspired from Subdue)

that works with graph partitions. Expansion of graphs occur in individual partitions *independently* achieving parallelism. The expanded substructures across partitions are then grouped across partitions for counting. Since mining is an iterative process, partitions may have to be updated for subsequent iterations. We introduce two partitioning techniques and corresponding Map/Reduce based algorithms for graph mining. Experiments indicate that instead of using high computation intensive super computers, graph mining can be done with a cluster of commodity machines and yet scale to graphs with millions of nodes and hundred of millions of edges.

Note that these partition-based algorithms still generate a huge set of intermediate results. Following user-defined pruning parameters a subset of the intermediate result is pruned *after* generation needing additional pruning cost. Hence any improvement that reduces generation of the intermediate result will result in improvement of the mining method. Therefore investigation of optimization techniques for substructure discovery is crucial.

1.3 Optimizations on Substructure Discovery

The expansion process used in the earlier technique expands a subgraph in all possible ways by adding an edge (incoming or outgoing if a directed graph) on any of its vertices. Such an expansion is complete as it guarantees generation of all possible substructures in every iteration. However, as a byproduct duplicates are generated when different substructures in the same iteration, expand into multiple copies of the same (exact) bigger substructure. The number of duplicates, thus generated, typically depend on graph characteristics.

Following expansion, duplicates impact the computation as well as the storage costs. First, generation of duplicates incurs extra computing cost. Second, the expanded substructures along with duplicates may need to be persisted before elim-

inating duplicates requiring additional storage. The duplicates finally need to be removed to *ensure correctness* requiring additional pruning cost. In the presence of partitions, the number of duplicates generated in individual partitions affect the network cost as these duplicates need to be brought together from different partitions for elimination. If there is a way to reduce (if not completely eliminate) duplicates, performance can be significantly improved across all components of *any* graph mining (storage, I/O, network and computation cost) technique.

In this dissertation we augment the previous expansion strategy by introducing three heuristics, each capable of reducing the number of duplicates generated during graph expansion. Our heuristics can be seamlessly integrated into most of the graph representations used for the iterative algorithm. In addition to establishing the theoretical correctness of each constraint, we investigate the composition of heuristics to achieve greater benefit. For scalability, we validate the benefits of using our heuristics on partition-based graphs. Finally, we present an extensive comparative analysis of the constraints with respect to the graph characteristics to help choose the best heuristic for an arbitrary graph.

1.4 Component Cost analysis

For any algorithm, cost analysis is critical for identifying places to improve the performance. For the Map/Reduce framework used in this dissertation, there are a number of costs associated with an algorithm. In main-memory based methods, the mining cost was mainly computed as main-memory access and computing costs. As disk-based mining came into existence where some part of data was in memory and the rest in disk, I/O cost based on the buffer sizes and hit ratios had an impact on mining. The entire mining cost was a cumulative of the main memory cost and the

disk access cost. With the adaptation of newer paradigms to graph mining, a more detailed component cost analysis of graph mining is warranted.

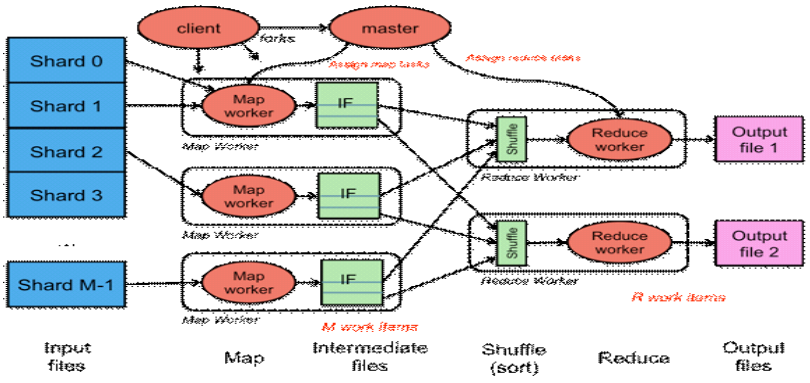


Figure 1.3: Map/Reduce paradigm

Figure 1.3 shows the costs that are specific to the Map/Reduce paradigm. A distributed paradigm comes with a distributed file system (DFS.) The initial partitioning strategy determines how the input file in broken into shards and kept in the DFS. The computation cost is specific to the mapper which involves writing intermediate files to local disk involving disk I/O. Sending intermediate files from mappers to reducers involve network cost. Finally after any computation cost incurred by reducers, the outputs are again materialized into the DFS. In case of an iterative task like graph mining, the output of a reducer is fed as input to a next set of mappers for subsequent iteration.

The choice of initial partitioning strategy and the underlying costs specific to the Map/Reduce paradigm bears an effect on graph mining. Hence this dissertation analyzes the cost of graph mining with different partitioning strategies and analyzes the role of initial partitioning for partition-based graph mining. A state-of-the-art graph partitioning technique METIS is initially used for graph mining and based on our analysis of its limitations, two new partitioning strategies have been developed for

substructure discovery. The cost analysis helps to evaluate the partitioning strategy for graph mining by discussing the trade-off between different components of the algorithms using these partitioning schemes. Since the number of map workers and reduce workers can be specified by users, the dissertation also investigates the effect of different numbers of Map/Reduce parameters on the graph mining cost.

Finally, based on the insights gained during the development and analysis of this algorithm and its evaluation, we try to characterize algorithmic classes for the Map/Reduce framework. The goal is to understand the performance issues for different classes of algorithm with different requirements. Specifically, we analyze the following three classes of algorithms: (i) non-iterative algorithms (such as joins), (ii) iterative algorithms (such as PageRank [22]) with no need for persistent data across iterations, and (iii) iterative algorithms (such as Substructure Discovery) with need for persistent data across iterations. The purpose of this comparison is to make it easier to develop and fine-tune algorithms pertaining to different classes based on their performance bottlenecks.

1.5 Contributions

The contributions of this dissertation are:

- Scaling graph mining to arbitrary-size graphs with commodity machines and partitions schemes using the Map/Reduce paradigm
  - Two Map/Reduce based algorithms for substructure discovery
  - Handling basic graph mining operations like subgraph expansion and similarity computation in the Map/Reduce framework
  - Extensive experimental analysis on graphs with varying characteristics showing scalability
- Optimizations over graph mining to improve the overall mining cost

  - Identified several constraint-based heuristics to reduce the number of duplicates generated along with their theoretical correctness
  - Analysis of two pruning strategies with respect to soundness and completeness using the proposed heuristics
  - Comprehensive analysis of the algorithms on single and partitioned graphs with diverse characteristics using the heuristics and pruning properties
  - Back-of-the-envelope analysis of the choice of heuristics to be used for an arbitrary graph
- Analysis of costs of partition-based graph mining using different partitioning schemes on the Map/Reduce paradigm
  - Analysis of existing partitioning strategies for graph mining and two new partitioning schemes to overcome their limitations
  - A detailed analysis of the costs associated with each component of graph mining with Map/Reduce with varying user-defined parameters using large scale real-world graph databases and synthetic datasets
- Back of the envelop comparison of several classes of Map/reduce algorithms from a performance perspective.

1.6 RoadMap

The rest of the chapters are organized as follows:

- In Chapter 2 we present the partition-based graph mining technique and algorithms using Map/Reduce and then evaluate our approach on large scale graphs with varying graph characteristics
- Chapter 3, extends our partition-based approach to work with various partitioning schemes and in the process discuss the cost of graph mining on various partitioning techniques.

- In Chapter 4 we elaborate on optimizations on graph mining, present the heuristics with associated conditions and discuss the results of experimental evaluation on the same with respect to graphs with varying characteristics
- Chapter 5 surveys the related work with respect to the motivation of this dissertation while Chapter 6 concludes the dissertation with directions for future work.

CHAPTER 2

SCALABLE GRAPH MINING

This chapter focuses on the problem of subgraph mining in the context of big data analytics. Analysis of fraud detection, finding friendships and other characteristics in social networks require that graph mining be done on very large graphs that go beyond the capabilities of previous approaches. Our goal is to initially use the Map/Reduce paradigm for mining interesting and repetitive patterns from very large graphs. As graph databases with structural information (e.g., FreeBase) is becoming prevalent, even graph queries can be processed as substructure matching in very large graph databases. Hence, it is imperative that we map mining algorithms using massively parallel paradigms to be able to process very large graphs and to obtain best possible speed-up to keep the response time meaningful. In this chapter, we discuss our approaches to substructure discovery using the Map/Reduce architecture and its open source implementation, Hadoop.

2.1 Introduction

Substructure discovery is the process of discovering substructure(s) as a connected subgraph in a graph (or a forest) that best characterizes a concept based on some criterion. Due to increasing size of graphs, there has been some effort on partitioned approach to substructure discovery as in [18, 19], but they do not discuss expansion of substructures across partitions. In order to analyze large graphs and discover interesting substructures within a meaningful response time, the input data needs to be processed using a paradigm that can leverage partitioning and distribu-

tion of data on a large scale. Map/Reduce provides a powerful parallel and scalable programming framework to process large data sets.

Briefly, in order to detect interesting concepts in a graph (or a forest), an iterative algorithm is used that: 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. Although main memory based data mining algorithms exist, they typically face two problems with respect to scalability:(i) storing the entire graph (or its adjacency matrix) in main memory may not be possible and (ii) the computational space requirements of the algorithm may exceed available main memory. Hence it is important to partition the graph such that the individual partitions do not encounter these two problems. Mining using these partitions is tricky and needs effective partition management strategies. Moreover each parallel approach requires synchronization of computations across partitions and proper key/value pairs need to be formulated in the Map/Reduce paradigm to accomplish proper synchronization and lossless computation. The contributions of this work are:

- Two approaches to partition management for graph mining and their analysis
- Corresponding Map/Reduce algorithms
- Formulation of key/value pairs for isomorphism detection and duplicate removal
- Extensive experimental analysis of algorithms using diverse data sets to validate intuitive conjectures for scalability, I/O usage and response time behavior.

Roadmap: The rest of this chapter is organized as follows: Section 2.2 presents the problem definition and a brief overview on graph mining. Section 2.3 and Section 2.4 detail our algorithmic approach and problem solving methodologies. Section 2.5 discusses experiments and their analysis while Section 2.6 concludes the chapter.

2.2 Preliminaries and Problem Definition

In this paper, we focus on labeled graphs where node and edge labels are not assumed to be unique. We introduce a few definitions used in the rest of the paper.

Graph: A graph $G = (V, E, V_L, E_L)$ consists of a set of V vertices and E edges. Each vertex has a vertex label belonging to the set V_L and a set of edge labels in E_L. The graph can be either directed or undirected. Figure 2.1 shows an example of a graph in relevance to this thesis.

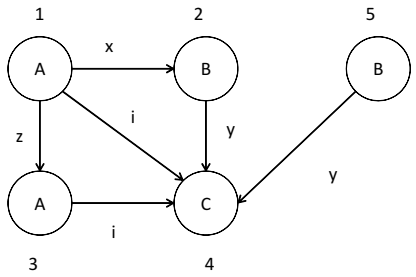


Figure 2.1: A Graph Example

Graph Isomorphism: *Formally, graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are **isomorphic** if (i) $|V_1| = |V_2|$ and $|E_1| = |E_2|$, (ii) there is a bijection (one to one correspondence) f from V_1 to V_2, (iii) there is a bijection g from E_1 to E_2 that maps each edge (u,v) in E_1 to (f(u), f(v)) in E_2.* We use isomorphism to detect identical (or exact) substructure patterns.We use isomorphism to detect identical (or exact) substructure patterns. Figure 2.2 includes a few isomorphic graphs present in Figure 2.1.

Minimum Description Length (MDL): Minimum description length (MDL) is an information theoretic metric that has been shown to be domain independent and highlights the importance of a substructure on how well it can compress an entire graph. Description length of a graph G is essentially the number of bits required to

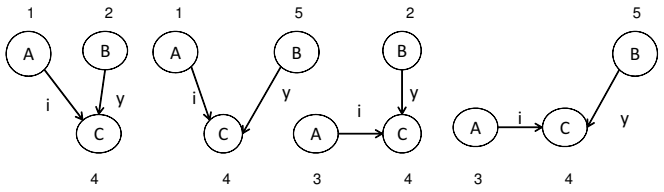


Figure 2.2: Graph Isomorphism Examples

encode the graph. For a substructure S which occurs in graph G, MDL is calculated using the formula $MDL = (DL(S) + DL(G|S))/DL(G)$ where DL(S) is the description length of the substructure being evaluated, $DL(G|S)$ is the description length of the graph as compressed by representing each instance of the substructure as a node, and DL(G) is the description length of the original graph. The substructure that compresses a graph best (i.e., minimizes $DL(S) + DL(G|S)$) is considered the best theory/concept in the graph. Both frequency of the subgraph and its structure has a bearing on compression. We compute this value by using the number of vertices and edges instead of bits. MDL is detailed in [1, 16].

Problem Definition: *Given a labeled graph as input, the general problem is to find the substructure(s) that reduces the graph best using the MDL or frequency principle. The subgraph size can be optionally specified to find the best substructure up to that size.* The goal is to develop algorithms that are amenable to partitioning the graph to satisfy the number and type of processors. For the current work, we have chosen the Map/Reduce paradigm.

2.3 Methodology

We shall first discuss the representation of input graph followed by strategies to manage partitions for parallel processing.

2.3.1 Input Graph Representation

Both undirected and directed graphs can be used for our approach but we have considered only directed graphs in this paper as it has relations explicitly represented using directions . Our input graph is represented as a sequence of unordered edges (or 1-edge substructure) including its direction. Each edge is completely represented by a 5 element tuple <edge label, source vertex id, source vertex label,destination vertex id, destination vertex label>. Table 2.1 shows the 1-edge substructures of Figure 2.1. The size of an edge is the sum of components in it and is assumed to be a constant. Our method can be easily extended for undirected graphs. Since undirected graphs do not have an explicit source-destination relationship, it can be converted into a directed graph by replacing an edge with two directed edges but giving rise to loops. Loops can be handled by keeping track of visited nodes/edges during graph traversals. For graphs where edges can be both directed and undirected, a sixth element needs to be introduced into the current edge representation indicating the nature of the edge (directed or undirected). Our representation is general and can be extended for multiple edges as well by using an edge identifier in the edge representation to demarcate multiple edges between same two nodes. Following our input representation[1] for a directed edge, a k-edge substructure (a connected graph of k edges) is represented by a collection of k 1-edge substructures. A graph is stored as a flat file with a 1-edge substructure in *each* line and acts as an input to our algorithm.

[1]Any other graph representation from which our representation can be inferred can also be used for our approach

Table 2.1: 1-edge substructures of Figure 2.1 (top) AND Generated Adjacency List (bottom)

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

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

2.3.2 Partition Management

As we are considering large graphs in this paper we assume that the entire information needed (adjacency lists of graph etc.) cannot be stored in the memory of a single machine. We are not considering disk-based alternatives to store the graph as iteration over disk-based graphs will incur significant I/O in each iteration. Hence partitioning the graph among multiple processors is our goal where each partition can be held in the memory of a single processor. Of course, if the number of processors and their main memory is not sufficient, either disk-based approaches or subdividing a partition based on memory availability and processing them sequentially can be used.

Consider a graph with V vertices and E edges. Since we partition the graph across multiple machines, we need to create a substructure partition consisting initially of 1-edge substructures. Expansion of a substructure in the substructure partition requires a corresponding adjacency list partition that contains adjacency list of each vertex present in the substructure partition. An adjacency list of a vertex id is the list of edges (not necessarily ordered) 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. Hence, the size of the adjacency partition will be much larger than the size of the substructure partition (at least initially) for each partition.

The initial representation of a graph can vary depending on how it is generated/traversed. We do not make any assumption on the input graph representation. For example, the input generated by SUBDUE graph is an unordered sequence of edges (1-edge substructures) that contains all relevant information about the edge (node labels, edge label and direction.) From this we need to create both our substructure and adjacency partition for our requirement. We also assume that we can infer the processing and memory requirement of each processor as well as the number of processors available for computation. We are not assuming homogeneous processors and hence the size of individual partitions may vary.

For a graph, creation of substructure partitions can be done in many ways. The adjacency partition depends on the substructure partition. Since these graphs are large, multiple passes on the initial input representation to generate these partitions need to be avoided. Although the substructure partitions are disjoint (same edge is not repeated across multiple partitions), a vertex can still be repeated among substructure partitions. For example, if the first 3 edges in Table 2.1 are in say, partition 1 and the rest in partition 2, vertices 1, 2, 3 and 4 are in the first substructure partition while vertices 2, 3, 4 and 5 are in the second substructure partition leading to some vertices (here, 2, 3 and 4) to be present across multiple substructure partitions. The number of times a vertex id is repeated across substructure partition determines the number of times the adjacency list needs to be replicated for that vertex in the adjacency partitions. Intuitively, the lesser is the number of repeated vertex ids, better is the quality of the initial partitioning.

Note that the above discussion is only for initial generation of substructure and adjacency partitions. After the first iteration, when new edges are added to a

substructure due to expansion, new vertices are added to the existing substructures in a substructure partition. The previously generated adjacency partition may not contain the adjacency list of these new vertices. Continuing from our last example, in Figure 2.1, when the substructure $< i, 1, A, 4, C >$ is expanded on vertex 4 in partition 1, a new vertex (here vertex id 5) is introduced in the substructure partition. This substructure cannot be expanded in the next iteration unless the adjacency list of vertex id 5 is inserted into the current adjacency partition. Hence the adjacency partition needs to be updated after each iteration to include the needed adjacency lists from other partitions. This is expensive as, in the worst case, all adjacency partitions may have to be traversed to update a single adjacency partition incurring significant i/o and processing.

The above overhead of updating adjacency partitions after each iteration can be avoided if we can keep the adjacency partition fixed to its initial set of vertices and substructures needing adjacency list for expansion are brought to the processor holding the proper adjacency partition. This will require some mechanism to quickly determine which adjacency partitions are needed to expand a given substructure. This will reduce the adjacency list update overhead but will incur additional processing for routing substructures to the processor with the right adjacency partition.

Below we present a technique that works on any partitioning scheme and address the nuances of the existing algorithm while moving to partitions.

2.3.2.1 Arbitrary Partitioning

As we are interested proposing parallel algorithms that are scalable and exhibit acceptable speedup we want to analyze our algorithm both for good and random partitioning schemes as this will provide the range achievable for speedup. We also know that keeping connected components in each substructure partition with min-

imum amount of repeated vertices across other partitions will give good results as the number of substructures crossing the partition, and hence the number of updates to the adjacency partitions in the later iterations will be minimized. On the other hand disconnected components in a substructure partition do not share the above and require more updates to the adjacency list.

In this approach each substructure partition E_i is paired with a corresponding adjacency partition AL_i. Since a single vertex may be repeated across multiple substructure partitions, the adjacency list of *that* vertex is also replicated across multiple adjacency partitions as well. Continuing from our example with substructure partitions, the adjacency list of the repeated vertices (2, 3 and 4) needs to be replicated across both the adjacency partitions. If $Repeat(AL_i, AL_j)$ is the set of repeated vertices across AL_i and AL_j where $i \neq j$ and $i, j \leq p$ then $\sum_{i \neq j}^{p} Repeat(AL_i, AL_j) \geq 0$. The lesser the value of $Repeat(AL_i, AL_j)$ the better is the partition.

In one pass of the graph, the edges of a graph are grouped into required number of non overlapping partitions. Each substructure partition needs a subset of the adjacency list of the entire graph (or an adjacency partition) which only contains the adjacency list of all nodes present in that substructure partition. Any partitioning scheme can be used for this purpose. Example of an arbitrary partitioning is shown in Figure 2.3. Each substructure partition is expanded in parallel using the corresponding adjacency partition. Alternatives to using adjacency partition (Schimmy [22]) require shuffling the entire graph structure in each iteration and have been shown to be inefficient in the literature. Although substructure partitions are disjoint, adjacency partitions are not.

In every iteration, newer substructure partitions are formed by grouping isomorphic substructures across existing substructure partitions. These isomorphic substructures have little interconnection among each other, as they initially belong to

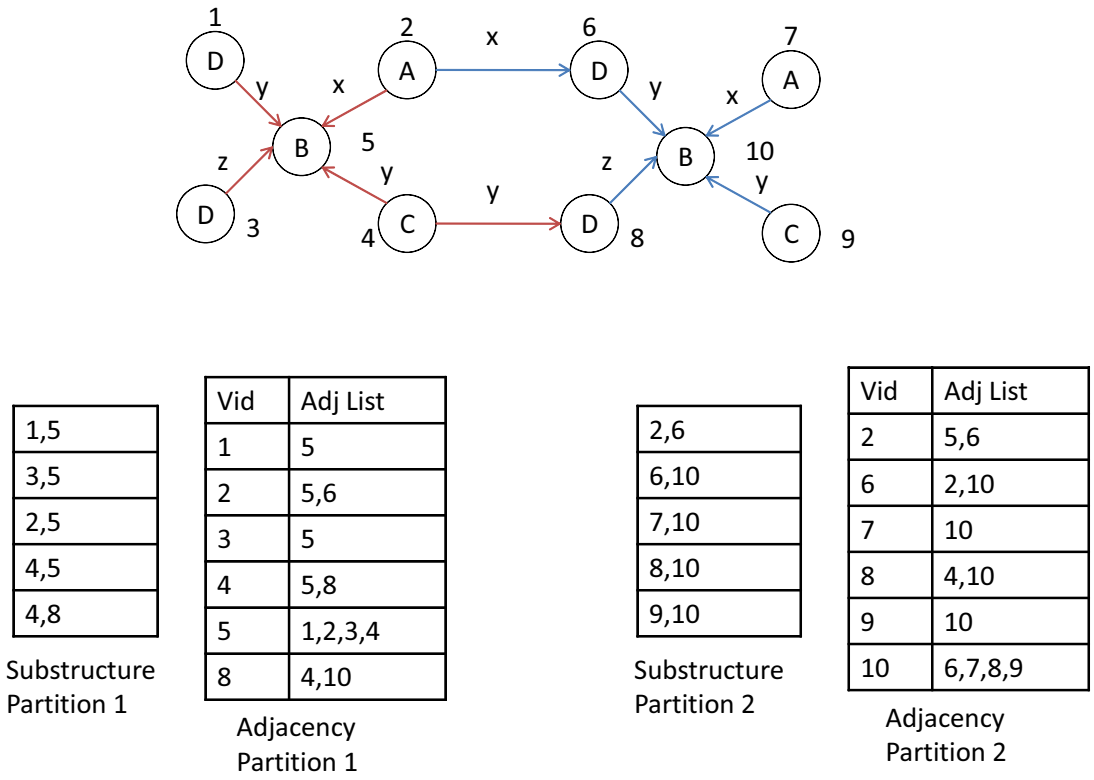


Figure 2.3: Arbitrary Partitioning

different substructure partitions. Correspondingly, a new substructure partition, even in the first iteration, has low intra partition connectivity. Hence we do not use a good quality initial partitioning scheme (like METIS [23]) which focuses on maintaining high intra partition connectivity and hence needs more computing effort. Needless to say, duplicate elimination and isomorphism check form the crux of our mining algorithm and we shall discuss how to handle them in detail in the next section.

Need for partition update during any iteration: Irrespective of how partitions are made on 1-edge substructures, the adjacency partitions may not be adequate beyond the first iteration. An expansion of a substructure may add a vertex id whose adjacency list is not in the current adjacency partition. Expansion by adding an edge (and hence new vertices) may grow a substructure across partitions. Consider the

graph in Figure 2.3. The red edges are are one partition while the blue ones are in another partition. An expansion on the edge $(2, 5)$ on vertex id 2, inserts a new edge and a new vertex (here 6) which lies in another partition. This substructure cannot be expanded in the next iteration unless the adjacency list of 6 is incorporated in the current adjacency partition. This can be shown true for any partition in any iteration (unless the partitions either contain the entire graph or correspond to disconnected parts of the graph). This necessitates updating the current partition by adding the adjacency list of the new vertex.

Moreover, grouping brings similar substructures across multiple substructure partitions hence creating newer substructure partitions. See in Figure 2.4, the two isomorphic structures occurring in different partitions are grouped together in a new partition. The vertices of these isomorphic substructures, if not present in the adjacency partition, needs to be added along with their adjacency list. This calls for an update of the corresponding adjacency partitions in every iteration to ensure independent substructure expansion in the subsequent iteration.

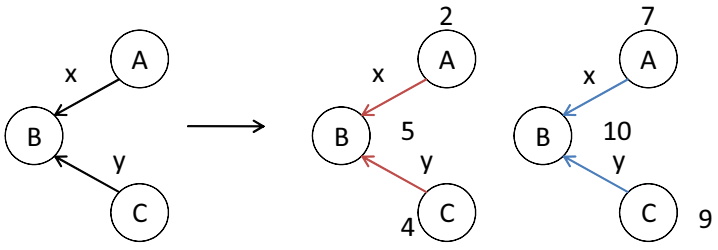


Figure 2.4: Necessity for Update

Considering p adjacency partitions, updating an adjacency partition requires loading and searching the other partitions for required adjacency lists in some order. The update stops when adjacency lists of all required vertices have been added using

one or more adjacency partitions (1 in the best case, p-1 in the worst case and p/2 in the average case.) Updates occur in parallel, making the update process as fast as the longest update across all the partitions. With graphs of bigger sizes, such an update phase has a huge bearing on the I/O.

One way to eliminate the overhead, incurred after each iteration, is to avoid updating adjacency partitions. We still have to expand substructures on all nodes. An alternative is to keep adjacency partitions fixed and expand a substructure in multiple substructure partitions using a fixed adjacency partition. This will avoid the I/O intensive adjacency partition update after each iteration but will require grouping substructures differently and sending them to appropriate partitions for expansion. This motivates our second approach to partitioning.

2.3.2.2 Range-Based Partitioning

If we want to keep the adjacency partitions in approach 1 (generated for the corresponding 1-edge substructure partitions) fixed across iterations, the major challenge is to direct a substructure to a processor holding the appropriate adjacency partition. To direct a k-edge substructure to appropriate adjacency partition(s), we need to quickly determine to which partition(s) the adjacency list of each vertex in the k-edge substructure belongs. Without an order among vertex ids across the adjacency partitions, this operation requires the complete information of all vertices and their adjacency partitions. This information of all vertices will be large and can not be stored in the memory of a single processor. This necessitates a partitioning scheme on global adjacency list, where the operation to direct a k-edge substructure can be done with information that can easily be accommodated in memory of a processor.

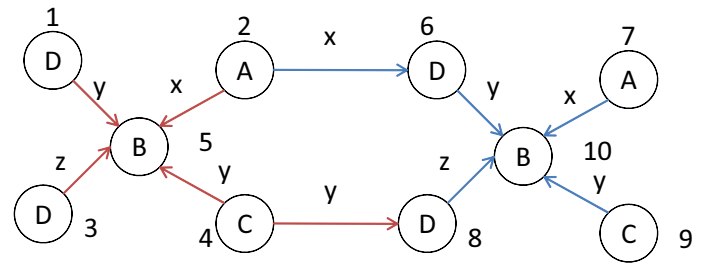
We propose range partitioning scheme on the global adjacency list for creating adjacency partitions. The range information is used to determine the adjacency

partition for a single vertex. This partitioning scheme removes the mapping between a substructure partition and the adjacency partition and requires range information to direct a substructure to a processor holding the appropriate adjacency partition. The range adjacency partitions do not have any intersection of vertices among them (unlike previous approach where adjacency list of vertices are replicated across adjacency partitions.) If AL is the global adjacency list then $\sum_{i=1}^{p} |AL_i| = |AL|$ where $AL_i \bigcap AL_j = \varnothing \ where \ i \neq j \ and \ i, j \leq p$. So $\sum_{i \neq j}^{p} Repeat(AL_i, AL_j) = 0$ making the range adjacency partitions different from the adjacency partitions in arbitrary partitioning.

In this approach, adjacency partitions are contiguous range of vertex ids. The number (and even the size) of partitions can tailored to match memory availability. Substructure partitions are created to match adjacency partitions (can also be created arbitrarily.) Range partitioning can be done over a single pass of the graph data input. In this approach, the adjacency partitions are disjoint (unlike the previous approach) and hence each vertex id and its adjacency list belongs to only one adjacency partition. Figure 2.5 shows the range-based adjacency partitions created here. Any substructure partitions can now be used for this approach.

If the substructure partitions are arbitrary, 1-edge substructures are grouped in the first iteration based on the adjacency partition needed to expand them. This results in the same substructure being served by multiple adjacency partitions. Since adjacency partitions are disjoint on vertex ids, each substructure, though catering to multiple adjacency partitions, is still expanded on each vertex id exactly once.

In this approach, the adjacency partitions do not change across iterations. The *same* adjacency partition is loaded for expansion, in each iteration, thereby avoiding the costly update phase. The overhead is this approach is incurred in the form



Vid	Adj List
1	5
2	5,6
3	5
4	5,8
5	1,2,3,4

Adjacency Partition 1

Vid	Adj List
6	2,10
7	10
8	4,10
9	10
10	6,7,8,9

Adjacency Partition 2

Figure 2.5: Range-based Partitioning

of routing the same substructure to multiple partitions. Routing substructures to multiple partitions will affect the shuffle cost.

One interesting feature of the ranged adjacency partitions is that, the vertices inside an adjacency partition and also across adjacency partitions are within an upper and a lower bound of vertex ids (the vertices need not be ordered inside a range.) If $highest(AL_i)$ denotes the vertex with highest vertex id in AL_i while $lowest(AL_i)$ represents the vertex with lowest vertex id in AL_i then $lowest(AL_i) > highest(AL_{i-1})$ and $highest(AL_i) < lowest(AL_{i+1})$. Hence the highest or the lowest values of vertex ids in each of the p adjacency partitions across p processors (a total of p values) is the range information needed to hash a substructure to its appropriate adjacency partition(s). The range information is also small enough to be kept in memory of each processor. See that a substructure may be sent for expansion to more than

1 adjacency partitions as the vertex ids in *that* substructure may be in multiple adjacency partitions.

Both the approaches, group expanded substructures based on isomorphism involving the same cost. However the range-based partitioning involves an additional shuffle cost for routing substructures to adjacency partitions. Since, mappers shuffle data to all reducers anyway, an increase in shuffle is likely to be significantly less than the update cost of each adjacency partition. Hence we believe that the range partitioning system, though non-intuitive, should perform better than the arbitrary partitioning scheme.

2.4 Partition-Based Substructure Discovery in Graphs

Finding the best substructures that compress the graph entails: generating substructures of increasing sizes (starting from an edge or 1-edge substructure) in each iteration and counting isomorphic (identical) substructures. Figure 2.6 shows the overall flow of how substructure discovery is done using the Map/Reduce paradigm. In a Map/Reduce based substructure discovery, a mapper uses a substructure partition and an adjacency partition to expand the substructures. The adjacency partition is loaded in the memory once every mapper (using the setup method in Map/Reduce). After expansion, the combiners (which are in mapper reducers) remove intra partition duplicates. All these expanded substructures barring the intra partition duplicates are sent across mappers to reducers where inter partition duplicates are removed and counting of exact similar substructures (or isomorphs) is done. A pruning metric like beam (or top-k) is used to determine the best substructures in an iteration and are used as candidates for the subsequent iteration.

However we need to first deal with duplicates that are generated as a byproduct of expansion. Duplicates must be eliminated to avoid wrong count. Below we

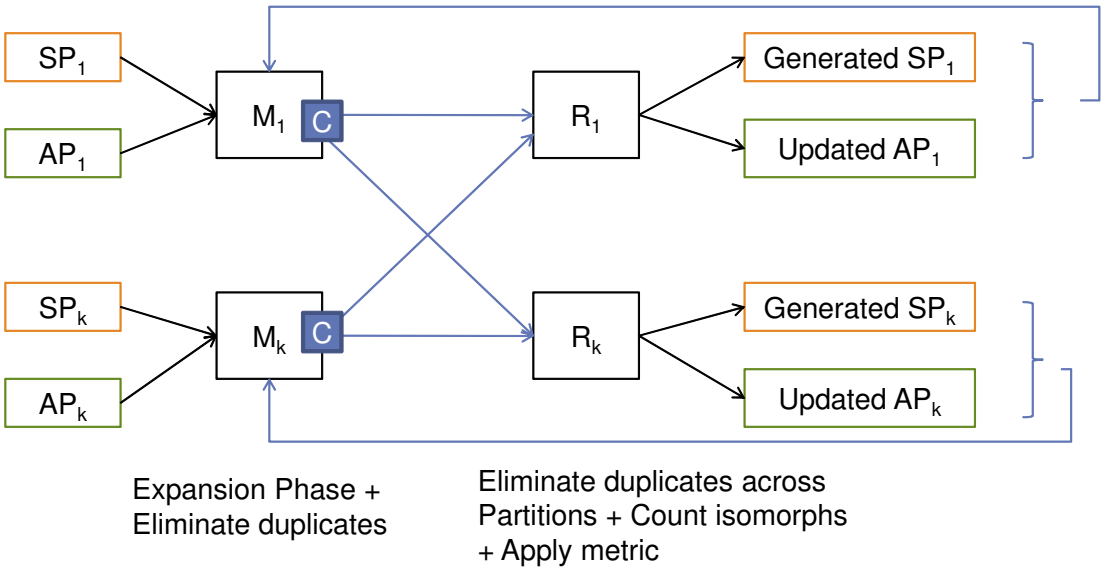


Figure 2.6: Substructure discovery using arbitrary partitioning

define the notions of a canonical instance and a canonical substructure to distinguish duplicates and isomorphic substructures.

2.4.1 Handling Duplicates

Systematic expansion in graph mining leads to generation of duplicates. Figure 2.7 shows an example of how duplicates are formed during substructure discovery. Some duplicates are formed inside a partition while some duplicates are generated across partitions. Therefore we need a technique to identify duplicates. Note that duplicates have the same vertex ids and the same connectivity among those vertex ids. For example in Figure 2.7 a and c are duplicates formed across partitions (a in P1 and c in P2) while b and d are generated in the same partition (here P1.) Duplicates generated in the same partition need not be shuffled across network to reducer to incur extra network cost and is removed at each mapper (where a partition is handled by using a combiner).

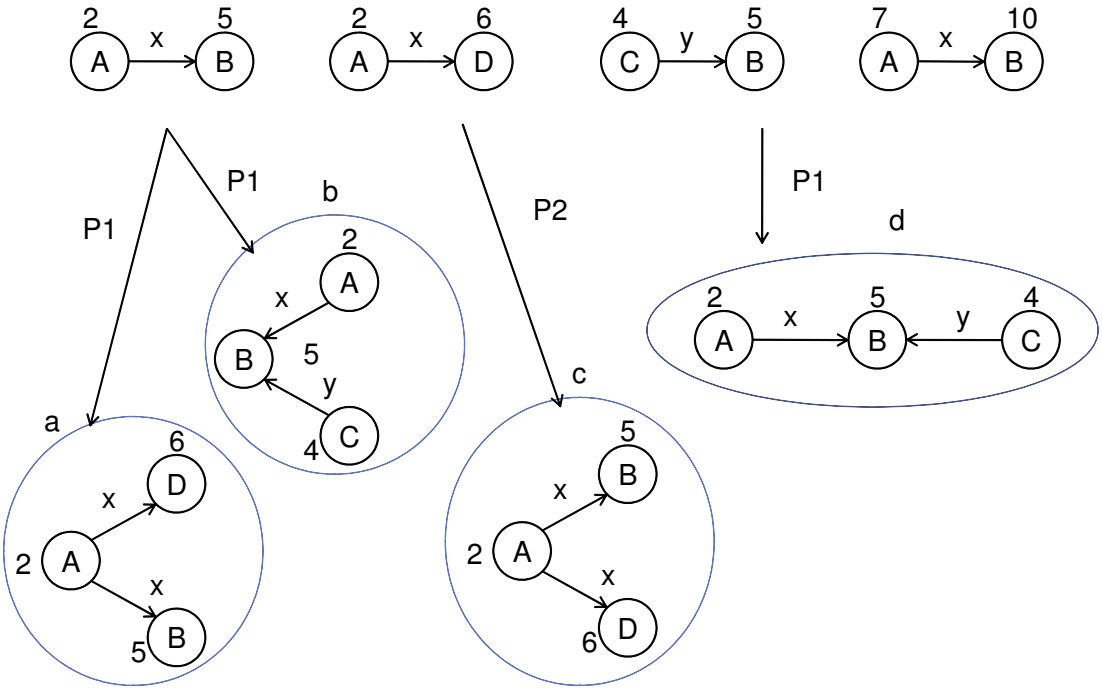


Figure 2.7: Duplicates in graph expansion

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 2.8 shows an example of duplicates and a canonical instance and shows how duplicates have the exactly same canonical instance.

Expanding a substructure therefore should maintain this lexicographic ordering. Expansion in substructure discovery is therefore similar to adding an element in a sorted list and is linear in the size of the substructure. The canonical representation

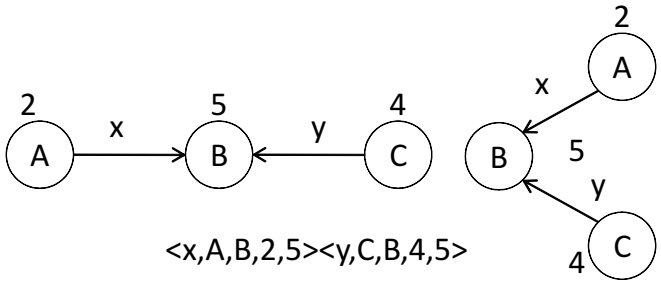


Figure 2.8: Example of canonical instance

of instances are useful for identifying and eliminating duplicate instances. However, we need to count the frequency of isomorphic (exact) substructures for which vertex ids cannot be used. Hence, we derive a canonical form of the substructure (without vertex ids) from the canonical instances.

2.4.2 Handling Isomorphs

Another result of expansion is the generation of isomorphic substructures. Figure 2.9 shows how multiple substructures grow into isomorphs. These isomorphs have the have vertex and edge labels but differ in vertex ids. Hence we first investigate if dropping of the vertex ids from the previous canonical instance representation is correct for identifying isomorphs.

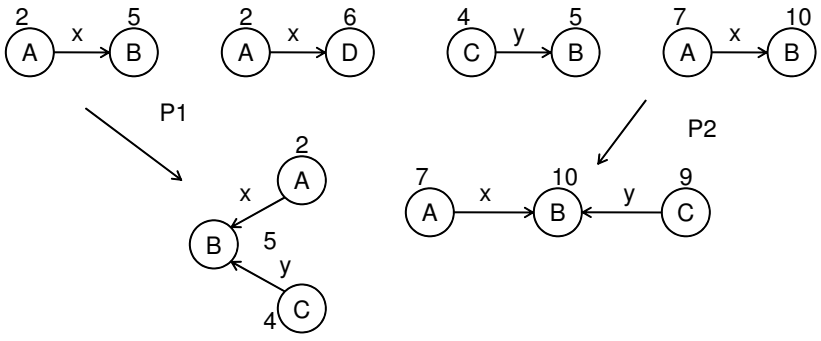


Figure 2.9: Isomorphs in substructure discovery

Theorem 2.4.1 *Dropping vertex id from canonical instance representation identifies isomorphs correctly*

Proof By Contradiction: Assume that dropping the vertex ids from canonical instance representation identifies isomorphs correctly. We just need to show one case where it does not work. See in Figure 2.10 there are two different substructures. When the vertex id was dropped from their canonical representation both bear the same representation. However these substructures still differ in connectivity. Vertex id 10 in one has an out degree of 2 while the similar vertex id 50 in the other substructure has 1 in degree and one outgoing edge. Hence we need to incorporate the notion of connectivity into the representation of isomorphs to demarcate them clearly.

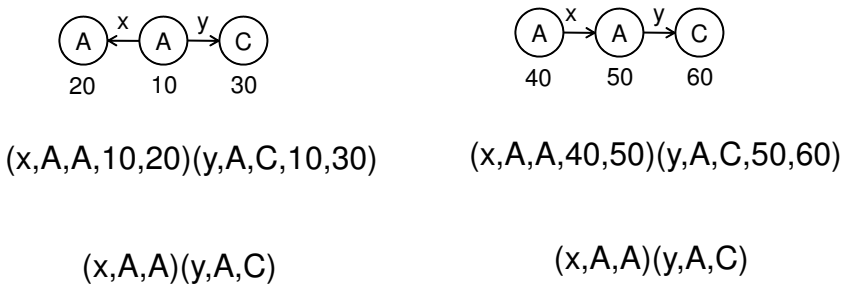


Figure 2.10: Need for newer representation for Isomorphs

This motivated us to capture the relative positioning of the vertex ids in a substructure for identifying isomorphs. A ***canonical k-edge substructure***, derived from canonical k-edge instance honors the relative positions of vertex ids in the canonical instance. Intuitively, two isomorphic substructures shall have the same relative ordering of vertex ids. Note that, the canonical instance already follows the lexicographic ordering. Therefore, we can easily construct a canonical k-edge substructure, following relative positions of the unique vertex ids in the order of their appearance

in the canonical instance. [2]. Figure 2.11 shows an example of how canonical substructure is created from the canonical substructure. See that the isomorphs have different canonical instances. However the relative positions following the canonical instance $(2, 5, 4)$ for the canonical instance 1 and $(7, 10, 9)$ for the canonical instance 2 boils down to $(1, 2, 3)$. Hence isomorphs can be identified using the canonical substructure.

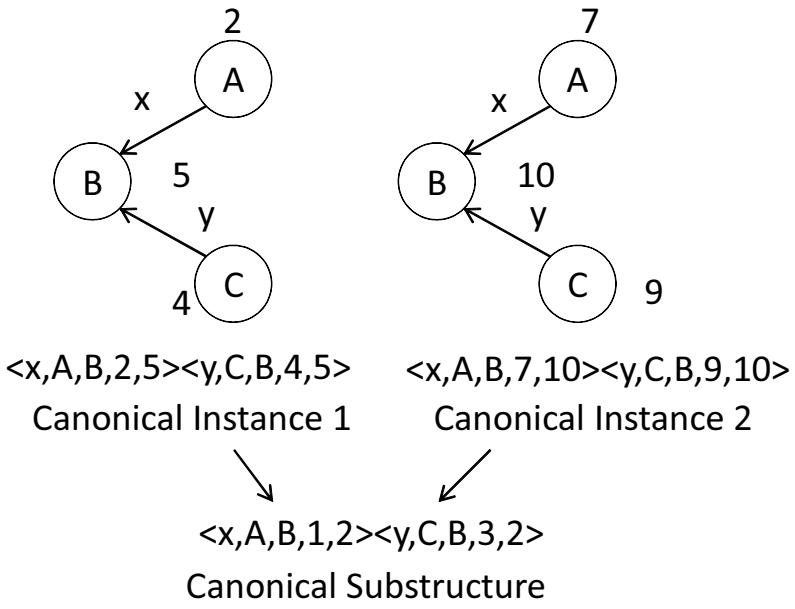


Figure 2.11: Example of canonical substructure

Below we introduce our two algorithms, one with arbitrary partitioning schemes and adjacency partitions, updates dynamicAL-SD(dynamic adjacency list substructure discovery) and the other with range-based static adjacency partitions, staticAL-SD(static adjacency list substructure discovery.)

[2]The canonical k-edge substructure can not distinguish pathological substructures with $k > 2$ where *all* node and edge labels in a single substructure are identical and hence cannot distinguish bigger substructures which have a partial substructure with identical labels.

2.4.3 Using Arbitrary Partitions (dynamicAL-SD)

For this algorithm, a mapper expands a substructure by one edge in all possible ways in each iteration while a reducer counts isomorphic substructures and updates adjacency partitions for use in the next iteration.

Substructure Expansion by Mapper: Each processor (mapper), uses a substructure partition and the corresponding adjacency partition for expanding a canonical instance by adding an edge. The substructure partition is read as mapper input (one canonical instance at a time) while the corresponding adjacency partition is loaded and kept in memory. Each expanded instance maintains the lexicographic order.

A *combiner* is then used to remove duplicates within a mapper. Removing duplicates using a combiner has the same cost as removal in a reducer, but improves the shuffle cost by not emitting duplicates in the same partition to the reducers. Reducers still remove duplicates generated independently in different partitions.

Algorithm 1 details our subgraph expansion routine in the mapper. For the first iteration, the input key is the line number and the input value is the 1-edge instance. Subsequently, for the k^{th} iterations, the value is the k-edge canonical instance. Line 3 loads the adjacency partition in memory.

Lines 8 to 17 expands the canonical instance and derives the canonical substructure from it. Creating canonical substructure from instance requires a hash table to identify relative positions of unique vertices (line 8.) Typically a mapper materializes the output to local disk. The combiner loads the mapper outputs, and eliminates intra partition duplicates at the mapper.

Duplicate Elimination and Frequency Counting by Reducer: The reducer receives instances across mappers grouped on the canonical substructure representation. After duplicate removal, the number of elements with the reducer does not reflect the proper count of discriminative isomorphic canonical instances due to over-

Algorithm 1 *Mapper of dynamicAL-SD Algorithm*

```
1: class Mapper
2: function SETUP
3:     Map = load corresponding adjacency partition
4: end function
5: function MAP(key, value)
6:     vMap = hashtable to hold unique vertex info
7:     for each vertex id v in value do
8:         vMap.put(v,null)
9:     end for
10:    for each vertex id v in vMap do
11:        aL = get adjacency list of v from Map
12:        for each 1-edge substructure e in aL do
13:            newValue = if e is not in value expand value by adding e
14:            if (u = e.getNewVertex) not in vMap
15:            vMap.put(u,null)
16:            update vMap with positions from newValue
17:            newKey = generate canonical substructure
18:            emit(newKey,newValue) to combiner
19:            delete u from vMap
20:        end for
21:    end for
22:    delete vMap
23: end function
```

lap (instances may overlap on vertices/edges.) We employ the most restrictive node (MRN) metric as defined in Section 4 to estimate the count of non overlapping isomorphic substructures.

To limit future expansions to high quality substructures, each reducer uses the metric (frequency or MDL) to store the top-k canonical substructures and all of their instances. Intuitively, this prunes the unimportant candidates and uses the best candidates for future expansion. The reducer uses a notion of beam (which can be specified by user) to store these best instances. We honor ties in the top-k ranked substructures by storing all the canonical instances (even with different canonical substructures).

We depict our reducer in Algorithm 2. Line 5 allocates a beam of size k as a hashmap to hold MDL values and all instances with that MDL value. Line 8 to line 14 remove duplicates and find instances with top k MDL values. The reducer also keeps an additional data structure for keeping the new vertices needed by adjacency partition for future expansion.

Expansion and grouping may introduce vertices not present in partition. This necessitates updates to the adjacency partition for the next iteration. The update continues from line 18 to line 24. Since we do not know the location of adjacency lists for a given vertex, we load all other adjacency partitions, one at a time randomly, until the current adjacency list is updated. After the update, line 27 emits all the instances in the beam for next iteration.

Algorithm 2 *Reducer of dynamicAL-SD Algorithm*

```
1: class Reducer
2: function SETUP()
3:     cId = corresponding adj partition id
4:     load adjMap //load adj partition cId
5:     beamMap = null //map to store best substructures
6: end function
7: function REDUCER(key, values)
8:     create Set isoSet to store isomorphs
9:     for each canonical k-edge instance ks in values do
10:        add ks to isoSet //remove duplicates
11:    end for
12:    c = count(substructures in isoSet)
13:    mdl = MDL(c,#vertices and #edges in key)
14:    update beamMap with mdl
15: end function
16: function CLEANUP()
17:    newV = vertices in beamMap not in adjMap
18:    for each partition p in hdfs do
19:        if p != cId load partition p
20:        for each vertex id vId in newV do
21:            if p has vId
22:            add adjacency list of vId in adjMap
23:            remove vId from newV
24:            if newV is empty write adjMap back in HDFS
25:        end for
26:    end for
27:    emit (null, each instance in beamMap)
28: end function
```

2.4.4 Using Range-based Partitions (staticAL-SD)

We now introduce our static adjacency list substructure discovery algorithm (staticAL-SD) which follows the range-based partition management strategy as discussed in Section 2.3.2. A closer look at this partitioning management strategy reveals two grouping criteria among substructures: instances across different processors must be routed to use the adjacency partition for expansion and the expanded instances then need to be grouped (based on their canonical substructure) for ranking purposes.

These two processes are sequential: we can only count after expansion. Hence each iteration of staticAL-SD requires two chained Map/Reduce jobs. The first Map/Reduce job aids in expansion of a substructure while the second Map/Reduce job removes duplicates and finds the top-k substructures with the best rank (using frequency or MDL.)

Expansion in first Map/Reduce job: Given a single canonical instance, a mapper uses the range information (from range-based partition management) to route a canonical instance to use appropriate adjacency partitions. With fixed adjacency partitions, the range information remains the same across iterations. All canonical instances needing the *same* adjacency partition are therefore hashed to the *same* reducer for expansion. The reducer with *that* adjacency partition in memory expands all canonical instances only on vertex ids present in *that* partition.

Algorithm 3 depicts the first Map/Reduce job for expansion. Each mapper loads the range information only once (line 3.) Note that an instance can have multiple vertices in the same adjacency partition(s.) The function in line 8 stores unique partition ids for vertices in an instance ensuring that an instance is routed only once to appropriate partitions for expansion. The mapper emits each unique partition id along with the canonical instance. Expansion in reducer logic is same as the mapper in dynamicAL-SD.

Duplicate Removal and Frequency Counting: The expanded canonical sub-

Algorithm 3 First Map/Reduce job for substructure expansion in staticAL-SD

```
1: class Mapper
2: function SETUP()
3:     R = load range information of adj partitions
4: end function
5: function MAP(key, value)
6:     PIds = Set for unique partition ids for an instance
7:     for vertex–id v in value do
8:         PIds.addPartitionId(v,R)
9:     end for
10:    for partition–id p in PIds do
11:        emit(p, value)
12:    end for
13: end function
14: class Reducer
15: function REDUCE(key, values)
16:    Load partition key from HDFS
17:    for each k-edge canonical instance ks in values do
18:        expand ks to (k+1)-edge canonical instance
19:        emit new key and new value similar to mapper in dynamicAL-SD
20:    end for
21: end function
```

structure from the first Map/Reduce job should be grouped for isomorphism checking.

In Map/Reduce environment, the reducer output cannot be an input to another reducer forcing us to use an intermediate mapper. Moreover we still need to generate the canonical substructure from the expanded canonical instance.

Instead of generating canonical substructures in the reducer for expansion, we do it in this intermediate mapper. Irrespective of where the canonical substructures are generated they require the same computation time. If canonical substructures are generated in the reducer of counting phase, they need to be sent from reducer to intermediate mapper and then again to the reducers of counting phase, incurring more I/O. Again, we use a combiner with the mapper to remove duplicates in a partition. The reducer now groups instances by the canonical substructure, removes duplicates across partitions and counts isomorphic substructures for evaluating MDL. It is worth mentioning that newer paradigms like Spark [24] with caching technique will save considerable disk I/O for loading partitions in both the approaches.

Analysis of Approaches: Typically the worst case update in dynamicAL-SD requires loading every adjacency partitions making it equivalent to loading the entire graph and hence proportional to the graph size. A worst case update, even at one reducer in dynamicAL-SD, incurs a worst case update time for the algorithm. Hence we believe that with graphs of increasing sizes, staticAL-SD, with no update stage, will be an improvement over dynamicAL-SD.

Connectivity of graphs also has an influence on the two approaches. The heavy interconnection between partitions in a densely connected graph will also lead to an increase in shuffle cost. Hence both dynamicAL-SD and staticAL-SD should perform better on sparse graphs involving lesser shuffle cost.

One prime factor affecting the two approaches is the number of duplicates generated. Depending on graph connectivity, a line graph (with k edges and k+1 nodes) generates 2 duplicates, while a completely connected graph with k-edges generate

k duplicates. With increasing iterations (or increasing k) the number of duplicates typically increase. Duplicates if generated across partitions, are shuffled to reducers, increasing shuffle cost.

Another discriminating factor between our algorithms is the I/O cost. For both the approaches, combiners have to read data from disks of mappers involving disk I/O. Moreover mappers have to send intermediate results to reducers introducing network I/O for both approaches. StaticAL-SD because of 2 map/reduce jobs in each iteration involves additional network I/O over its counterpart. Note that the update phase in dynamicAL-SD involves both disk I/O and network I/O as the adjacency partitions are loaded from a distributed file system and not a local file system.

One more factor affecting the I/O is the substructure size. Typically a k-edge substructure is represented as a sequence of k 1-edges and hence with increasing iteration, the size of key/value pairs which increase the shuffle cost. The increase in substructure size, has an effect on the shuffle cost with increasing iterations.

For both the algorithms, beam act as a heuristic to limit best substructures. Increase in beam will allow storing more substructures thereby increasing the set of intermediate results across each iteration resulting in increase of both I/O and processing time.

With increasing number of processors, dynamicAL-SD ends up with a constant update phase, prompting staticAL-SD to outperform it in speedup as well. As is typical of partitioned approach, graphs of smaller sizes will not benefit from our approach, where the overhead of managing partitions may dominate the performance cost.

2.5 Experimental Analysis

In this section we experimentally evaluate staticAL-SD and dynamicAL-SD. All experiments are conducted using Java on a Hadoop cluster with 1 front-end server and 17 worker nodes each having a 3.2 GHz Intel Xeon CPU, 4 GB of RAM and 1.5 TB of local disk. The server has same specification but with 3 TB of local disk. Each node was running Hadoop version 1.0.3 on ROCKS Cluster 6.3 operating system and connected by gigabit Ethernet to a commodity switch.

2.5.1 Experimental Setup

We experiment on several real world and synthetic datasets to establish the effectiveness and scalability of our approach. Table 2.2 shows the datasets that we use to evaluate our approach. For both the Orkut and LiveJournal datasets, we have randomly assigned a category to the user nodes from a set of distinct 100 categories following a Gaussian distribution. Subgen, a synthetic graph generator is also used which allows embedding small graphs with user defined frequency in a single graph giving better control in generating graphs of different sizes and characteristics.

Table 2.2: Datasets for experiments

Name	Nodes	Edges	Use
LiveJournal (snap.stanford.edu/data/com-LiveJournal.html)	3.99M	34.68M	scalability
Orkut (snap.stanford.edu/data/com-Orkut.html)	3.07M	117.18M	scalability
Subgen (ailab.wsu.edu/subdue)	800K	1600K	correctness

2.5.2 Correctness and Efficiency

The correctness of both our algorithms are verified by running them on small synthetic graphs generated by Subgen. Subgen was used to generate small graphs with predefined embedded substructures from size 20V/40E to 800KV/1600KE and Subdue [1], staticAL-SD and dynamicAL-SD when run on them discovered the same number of substructures. For comparison, both the Map/Reduce based algorithms are run on a single mapper and a reducer. Figure 2.12 shows the performance of the algorithms on varying graph sizes. For small sized graphs, the performance of Subdue was better than our Map/Reduce based techniques as Map/Reduce required an initial setup cost. However from graphs of size 5KV/10KE our algorithms started performing better than Subdue. Subdue being a main memory algorithm required dedicated data structures and pointer operations for duplicate elimination. Even with increasing graph sizes above 20KV/40KE, Subdue failed to complete while our techniques still ended up completing the process. Note that at the crossover point of 15KV/30KE staticAL-SD started performing better than dynamicAL-SD hinting at the benefits of using staticAL-SD over its counterpart..

2.5.3 Comparison of Approaches

To compare individual performance of dynamicAL-SD and staticAL-SD algorithms we run 5 iterations of both algorithms with a beam of size 4 on the LiveJournal and Orkut graphs. The average of 5 runs is taken to avoid cold start issues. A comparison between the two approaches in terms of iterations on the liveJournal graph is shown in Figure 2.13. The update dominates the execution time at smaller iterations. The update shows a linear growth with increasing number of iterations. The effect of iterations on staticAL-SD is portrayed in Figure 2.14. See that with increasing iterations, the differentiating factor between the two algorithms lie in its update cost.

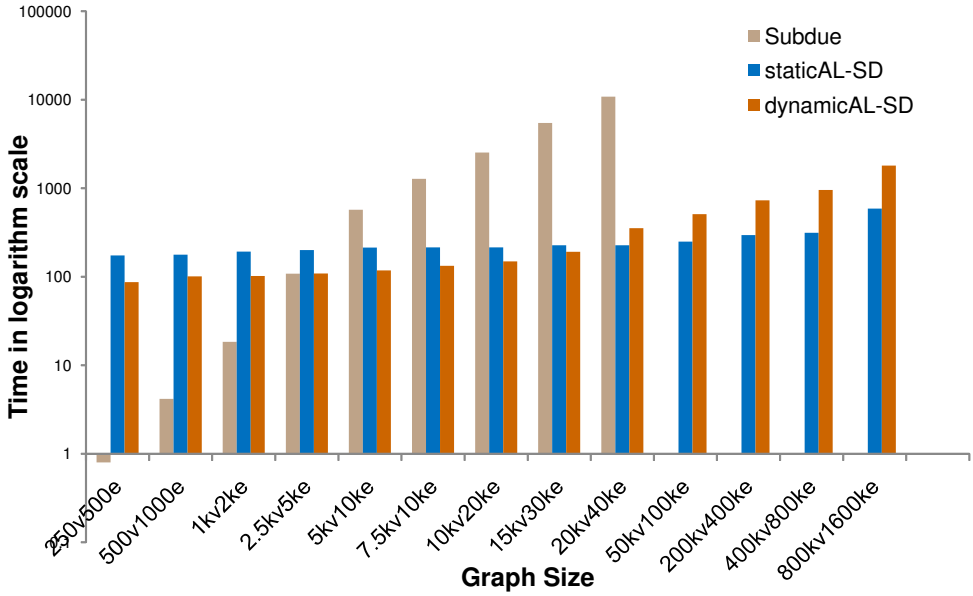


Figure 2.12: Comparison of all techniques with varying graph sizes

Removal of the update phase makes staticAL-SD perform better than dynamicAL-SD. The performance improvement depends on the ratio of the in-memory computation and the HDFS I/O cost. StaticAL-SD is better in 5,10 and 15 iterations than dynamicAL-SD by 38.8%, 27.5% and 26.2% respectively.

The comparison of dynamicAL-SD and static AL-SD with respect to speedup is shown in Figure 2.15. With increasing iterations, the number of partition loaded for update tends to reduce by a little margin making the speedup vary. If the update nullifies at any iteration, both staticAL-SD and dynamicAL-SD shall have the same speedup in that iteration as the amount of computation done by both the approaches is identical. For any iteration, we observe that staticAL-SD gives a better speedup than dynamicAL-SD.

Our results in Figure 2.16 indicate that in LiveJournal, staticAL-SD outperforms dynamicAL-SD by 38.8%, 46.1% and 52.1% with 2, 4 and 8 reducers respectively while the trend remained same in Orkut with 42.1%, 49.8% and 56.8% im-

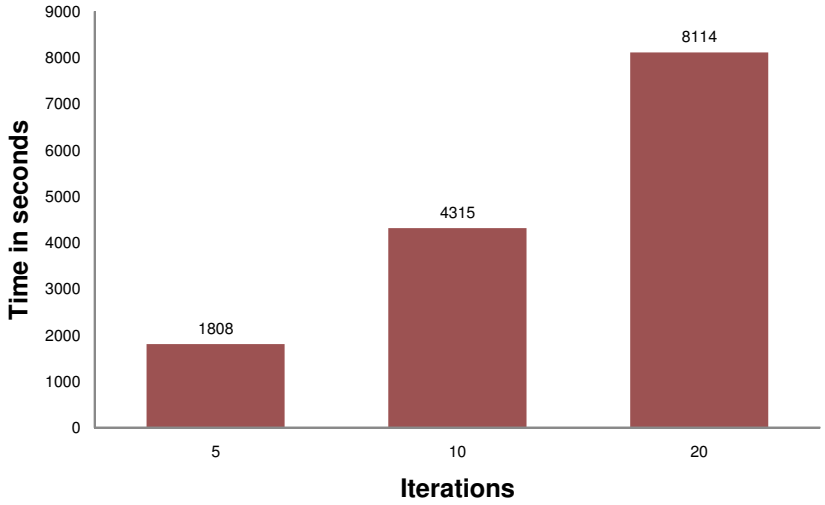


Figure 2.13: Running time of dynamicAL-SD over iterations

provement. The increase in improvement on the same graph on increasing reducers is attributed to decrease in computations on increasing number of processors, and a constant partition update time in each reducer. The increase in improvement across graph sizes is due to more time spent in the update phase. Hence with graphs of bigger sizes, static AL-SD will show a steady improvement over its counterpart.

2.5.4 Scalability

Our results in Figure 2.17 and Figure 2.18 indicate that staticAL-SD has a speedup of 39.4% from 2 to 4 reducers in LiveJournal and 34.4% from 4 to 8 reducers as compared to dynamicAL-SD's 22.9% and 19.3% respectively. StaticAL-SD on Orkut also shows the same trend with 42.6% from 2 to 4 reducers and 39.7% speedup from 4 to 8 reducers as compared to DynamicAL-SD's 23.7% and 20.2% respectively. DynamicAL-SD has a lower speedup due to its constant update phase at each reducer. Note that with doubling number of reducers, the time for mining is not exactly halved. This is due to two reasons. First, with doubling the number of reducers,

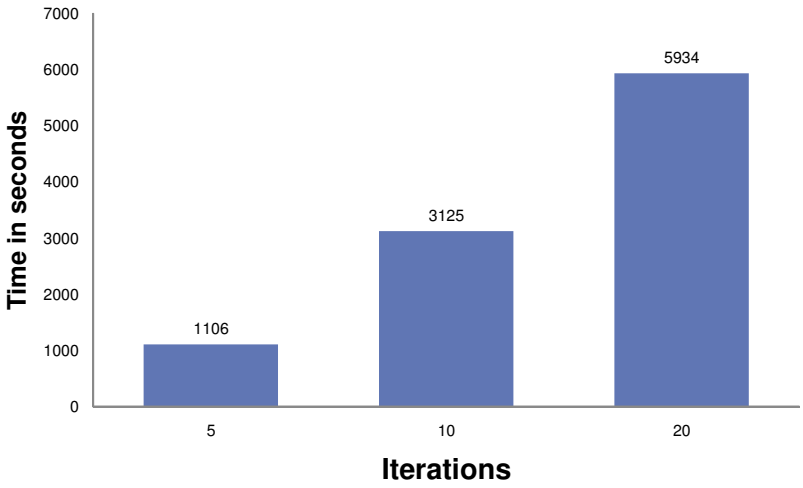


Figure 2.14: Running time of staticAL-SD over iterations

each reducer emits the instances with local top-k MDL values to find the instances with global top-k best MDL values. So the amount of data handled by each reducer is not necessarily halved. Second, as the reducer output records increase, the I/O cost increases. Typically in a Map/Reduce system the I/O is defined as the sum of map input records, map output records, reduce input records and reduce output records. Increasing number of reducers coupled with the notion of beam in each reducer increase the reducer output in each iteration thereby not generating a linear speedup.

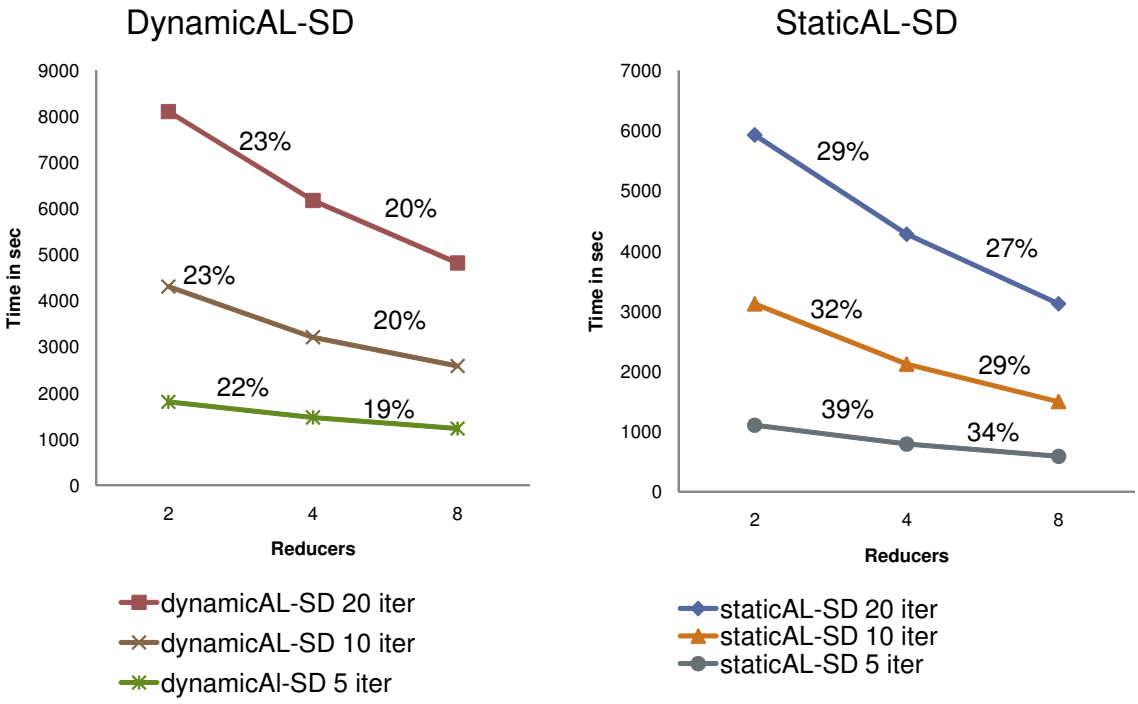


Figure 2.15: Speedup with varying iterations

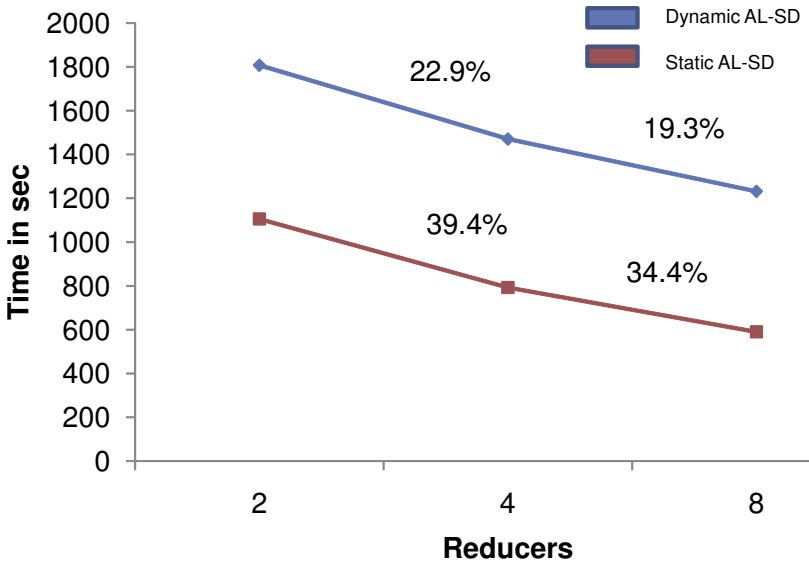


Figure 2.17: Speedup staticAL-SD vs. dynamicAL-SD over LiveJournal graph

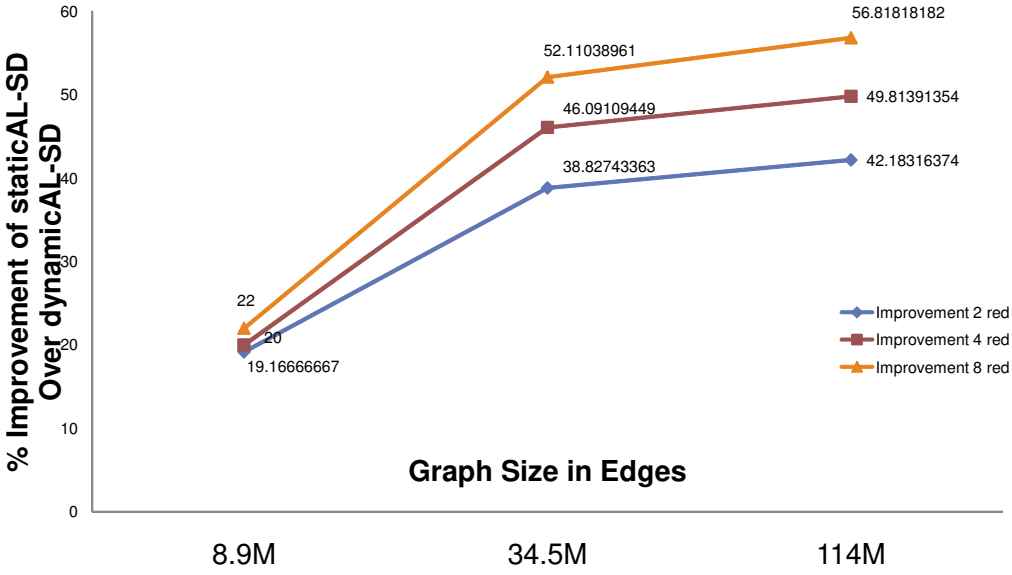


Figure 2.16: Improvement of StaticAL-SD over DynamicAL-SD varying graph sizes and processors

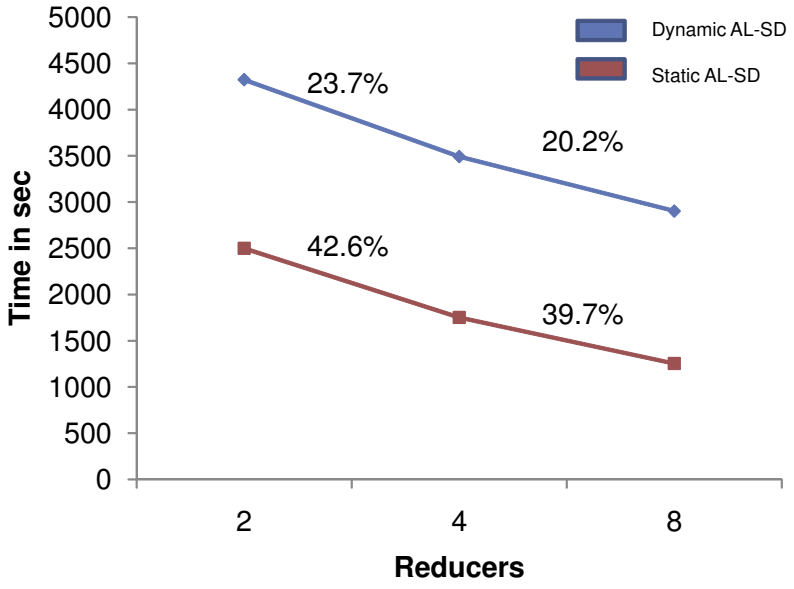


Figure 2.18: Speedup of staticAL-SD vs. dynamicAL-SD over Orkut graph

2.6 Conclusion

In this dissertation, we have shown how substructure discovery algorithms can be made to work efficiently using the Map/Reduce paradigm. Correctness and effectiveness has been the main concern as we develop Map/Reduce based algorithms. Since partitioning is the key to scalability of graph mining, we proposed an approach where the shuffle can be effectively used for keeping the adjacency partition fixed. It still needs to be argues if the choice of initial partitioning plays a role in performance of the distributed graph mining algorithms. The next chapter of this dissertation extends this work by introducing a cost model to further analyze the effect of costs in a distributed graph mining environment.

CHAPTER 3

COMPONENT COST ANALYSIS IN DISTRIBUTED FRAMEWORK

This chapter focuses on deeper understanding of performance of substructure discovery in a distributed framework by investigating component costs across different stages of the earlier algorithms. The division of these costs help evaluate different partitioning schemes and paves way for further optimizations by choosing one partitioning strategy over another. In a nutshell this chapter evaluates partitioning strategies from a component cost perspective. First, existing widely-accepted partitioning schemes are evaluated for their advantages and limitations for the chosen framework. Second, new partitioning schemes are proposed to improve the performance of the algorithm in several ways. Component cost analysis is performed analytically, and experimentally validated for different partitioning schemes using diverse synthetic and real-world graphs. Third, three algorithm classes of interest are analyzed for the traditional Map/Reduce architecture and compared with respect to component costs to provide insights into algorithm design. Finally, the portability of our algorithms to other paradigms such as Spark is discussed to ascertain the

3.1 Introduction

One of the most widely used techniques for scaling is divide and conquer which translates to partitioning of graphs. This approach is commonly used in the chosen Map/Reduce distributed framework as well. Although there have been some effort on partitioned approach to substructure discovery [18,19], they do not discuss expansion of substructures across partitions leading to approximate results. If one wants to

get the same (and correct) results with and without partitioning, the algorithms become complex due to bookkeeping associated with substructures that cross partition boundaries and exchanging that information across processors.

In order to achieve scalability for graphs of any size, the paradigm chosen should be capable of scaling the number of partitions as well as scaling the number of processors to match the graph size. One of the few paradigms that satisfies the above criteria is Map/Reduce in any of its variants (traditional, spark, etc,) However, none of the existing algorithms are designed for a parallel paradigm. Hence casting them into a parallel paradigm (such as Map/Reduce) correctly requires a re-examination of the algorithms (along with an appropriate representation) to parallelize them effectively. Moreover, some of the unique features of the Map/Reduce paradigm, such as the use of key/value pairs, what can be done during the map phase and what can be done during the reduce phase, need to be taken into account for developing an algorithm. In this paper, we propose two Map/Reduce algorithms to efficiently discover substructures thus facilitating substructure discovery on graphs of any size.

In addition to ensuring scalability, performance of the approach is equally important. In a partitioned, distributed processing approach, the distribution of effort incurred for processing a partition is important and most likely depends on the partition size as well as other graph characteristics. This makes the choice of partitioning schemes a key ingredient of partitioned substructure discovery. The criteria used for partitioning graphs need to match with the processing needs to address performance aspects. Existing widely-used and popular graph partitioning approach METIS [23] focuses on making connected partitions while minimizing the number of edges between partitions (termed cut set.) We discuss the suitability of partitions generated by METIS for substructure discovery leading to new partitioning schemes to overcome some of its disadvantages. METIS uses a multi-level approach

and is intended for creating partitions once. As a result, it tries to optimize the partitioning criterion and hence has a higher partitioning cost as well.

Performance of an algorithm using a framework, such as Map/Reduce, is dependent on a number of component costs. We analyze our substructure discovery algorithm in terms of component costs including loading and update of persistent data across iterations. We believe that an analysis of these component costs will provide insights into potential improvements in addition to evaluating the performance. Moreover, a linear speedup is desirable in a distributed system to ensure effective use of parallelism. For a chosen number of machines, the distribution of these costs contributes to the speedup achieved.

Finally, based on the insights gained during the development and analysis of this algorithm and its evaluation, we try to characterize algorithmic classes for the Map/Reduce framework. The goal is to understand the performance issues for different classes of algorithm with different requirements. Specifically, we analyze the following classes of algorithms: (i) non-iterative algorithms (such as joins), (ii) iterative algorithms (such as PageRank [22]) with no need for persistent data across iterations, and (iii) iterative algorithms (such as Substructure Discovery) with need for persistent data across iterations.

The purpose of this comparison is to make it easier to develop and tune algorithms pertaining to different classes based on their performance bottlenecks. The contributions of this chapter are:

- Development of correctness-preserving Map/Reduce algorithms to substructure discovery using graph partitions and their analysis
- Analysis of existing partitioning strategies for graph mining and two new partitioning schemes to overcome their limitations

- A detailed analysis of the costs associated with each component of graph mining with Map/Reduce with varying user-defined parameters using large scale real-world graph databases and synthetic datasets
- Back of the envelop comparison of several classes of Map/reduce algorithms from a performance perspective.

This paper extends and improves upon our earlier work [25, 26] by incorporating partition comparison and analysis and a detailed discussion of component costs along with their analysis and experimental validation. In addition, different algorithm classes are compared from a performance perspective.

RoadMap: The rest of the chapter is organized as follows. Section 3.2 introduces the component costs for graph mining. Section 3.3 compares a state-of-the art partitioning strategy (METIS) with our techniques. Section 3.4 continues onto the variation of the computation costs along with associated experiments while Section 3.5 and Section 3.6 discusses various classes of graph mining problems from component cost perspective and portability of those techniques to other paradigms such as Spark. Finally Section 3.7 concludes this chapter.

3.2 Cost Analysis

In addition to ensuring scalability, performance of the approach is equally important. In a partitioned, distributed processing approach, the distribution of effort incurred for processing a partition is important and most likely depends on the partition size as well as other graph characteristics. This makes cost analysis and the choice of partitioning schemes two key ingredients of partitioned substructure discovery. The criteria used for partitioning graphs need to match with the processing needs to address performance aspects.

Performance of an algorithm using a framework, such as Map/Reduce, is dependent on a number of component costs. Costs specific to substructure discovery for this framework during each iteration are: (i) computation cost for expanding substructures in a partition (including duplicate generation, identification, and removal) (ii) communication cost of sending substructures from mapper machines to reducer machines (iii) cost of read/write on local file system (iv) cost of read/write on distributed file system The read/write cost includes the cost of loading or updating information needed for substructure expansion. For a traditional Map/Reduce framework where persistence of data is not possible or allowed across iterations, they need to be loaded during each iteration. We analyze our substructure discovery algorithm in terms of these costs including loading and update of persistent data across iterations. We believe that an analysis of these component costs will provide insights into potential improvements in addition to evaluating the performance. Moreover, a linear speedup is desirable in a distributed system to ensure effective use of parallelism. For a chosen number of machines, the distribution of these costs contributes to the speedup achieved. Below we discuss each cost individually and highlight the factors affecting these costs

3.2.1 Computation Cost

Computation cost includes (i) expansion of substructures in each processor (ii) counting isomorphs in each reducer. Expansion of a substructure requires addition of an edge in a lexicographically ordered canonical instance. Adding an edge requires linear effort with respect to size of substructures. The number of substructures however depends on choice of beam, grows up to a certain iteration and then starts decreasing. The second component of computation cost is duplicate removal. Duplicate removal requires pairwise computation among same sized substructures.

This incurs quadratic complexity on the number of instances to be checked for duplicate elimination. The pairwise computation is done first in each combiner for removing intra partition duplicates and then again at the reducer for removing inter partition duplicates.

A part of update cost at the mapper includes an initial loading time for the partitions (substructure and adjacency.) In Map/Reduce terminology, the loading of files occur after the workers have been initiated in a separate process called *setup*. The first line of the substructure partition contains the connected adjacency partition description. Hence the adjacency partition is loaded when the computation has begun thereby including it in the computation cost.

3.2.2 Shuffle Cost

All the expanded substructures need to be sent from mapper to reducer for comparison incurring I/O cost. If combiners do not remove any duplicates, in the worst case, the amount of data shuffled across the network is equal to the number of substructures generated in the mapper and hence dependent on the computation cost. Typically the mappers write their output to a circular buffer which spills them on the disk when 80% of the buffer is full. Thereby duplicate removal in a combiner requires fetching these files from local disk and comparing them for duplicates. These instances after duplicate removal are sent to the reducers. The shuffle cost is dependent on the quality of the network between the mappers and the reducers. Our experiments use a 1GHz Ethernet switch for data transfer across mappers and reducers.

Note that the mapper and reducer processing are not sequential. A mapper, depending on its workload can finish early thereby allowing a reducer to start before all the other mappers have finished executing. Therefore it is difficult to accurately

measure the shuffle time. We estimate shuffle cost by noting the first mapper start time, the first reducer start time and the final reducer end time.

3.2.3 Update Cost

An adjacency partition needs to be updated in an iteration for correctness and for use in subsequent iteration. Update of an adjacency partition requires loading other adjacency partitions from HDFS. Considering **p** partitions of adjacency lists, update of a partition requires a worst case loading of all other *(p-1)* partitions. In the best case, update is satisfied by loading a single partition. In the average case, half of the partitions need to be loaded for the update. It is possible to load a partition completely and then experience zero contribution during update. However we do not have any apriori information about the partitions and the order of loading the partitions. Hence we load partitions, based on ascending order of partition ids. In a small set of experiments on small sized graphs, the order of loading did not matter as each partition contributed in the update.

Note that this update process occurs in each reducer. The number of reduce tasks is equal to the number of partitions. Different reducers can have different update costs. However the mapper of the next stage can not start unless all reducers have finished processing. Therefore the reducer requiring the worst update cost dictates the total update cost for an iteration. Note that, at every iteration, newer vertices are added to a partition but no vertices are deleted to preserve correctness. This makes the number of partitions loaded to decrease across iterations. However the total amount of vertices loaded in the worst case for update is equivalent to the entire adjacency list of the graph. With increasing number of processors or user parameters the worst case update cost remains same for the same graph size.

3.3 Partitioning Strategies

Expansion of a 1-edge substructure to a 2-edge substructure (and beyond), adds new edges (and hence new vertices) to the substructure partition. Moreover isomorphic expanded substructures across partitions are grouped together to create newer partitions which contains vertices across different adjacency partitions. These new vertices, if not present in the current adjacency partition, needs to be added along with their adjacency list in the current partition after each iteration. This is necessary to ensure correctness in the subsequent iteration. Here we look at the existing partitioning strategy (METIS) for a graph mining usability perspective.

3.3.1 Existing Partitioning Strategies

METIS [23] is one of the earliest graph partitioning schemes that uses multilevel algorithm. It supports different heuristics in each phase for compressing the graph to hundreds of vertices, run a main memory partitioning algorithm and un-coarsen the graph to the bigger size preserving equal sized partitions. Such heuristics include random matching (for unweighted graphs), heavy edge matching and light weight matching (for weighted graphs) and heavy clique matching (also for any graph.)

Metis can be also perceived as a version of the arbitrary partitioning scheme that minimizes edge cuts. However METIS suffers from the following drawbacks as outlined in Table!3.1.

First, we want a partitioning scheme where we can specify any number of partitions. This makes the scheme applicable to any group of machines. METIS with its k-way heuristic allows such a setting. Second, in a distributed environment we deal with heterogeneous machines. These machines have different main memory and hard disk capabilities. This is similar to Amazon EC2 instances where a cluster can be a combination of machines with different computing power (such a micro, large, small

Table 3.1: Our Requirement for Partitioning and METIS

Requirement	METIS	Reasoning
Control over number of partitions	AND	Amenable to any number of partitions
Control over size of partitions	NO	Usable for any heterogeneous group of machines
Edge based partitions	NO	Vertex based partitioning better
Adjacency List for each partition	NO	Adjacency Lists are needed for expansion
Focus on connectivity and labels	NO	Similarity depends on both connectivity and label similarity

etc.) METIS focuses on creating equal sized connected partitions and hence is not suitable for a heterogeneous group of machines. METIS if used in such a scenario will create different workloads across the machines hampering speedup. Arbitrary partitioning can easily create heterogeneous partitions and is a better fit for such applications.

METIS creates vertex based partitions. Expansion with vertex based partitions occur by adding a vertex in every iteration. Such an expansion scheme is harmful as addition of a vertex may add multiple edges. This might lose important intermediate instances. As a result post processing on the METIS output is required to convert vertex-based partitions to edge based partitions. Definitely METIS will outweigh arbitrary partitioning only if gains more savings than the post processing costs. Additionally, METIS does not create adjacency partitions. This requires an extra pass over the entire graph to create the adjacency list. With these graphs being big in size, each pass on a graph takes considerable amount of time.

Finally, METIS focuses only on the connectivity and not on the labels. Substructure discovery deals with isomorphs and duplicates which have the same labels

as well as connectivity. This makes partitioning strategies considering only one connectivity significantly less powerful.

3.3.2 Cost Analysis (METIS vs. Arbitrary Partitioning)

The main difference between arbitrary partitioning and METIS lies in the quality of initial partitioning. The computation cost is almost the same as the same number of expansions are always attempted. However with the initial different configuration of the partitions the work done by each processor will vary slightly. The shuffle cost depends on the number of substructures to be routed across mappers to reducers and remains almost the same. The only difference is due to variation in inter partition duplicates. Therefore the main difference lies in the update stage. If Metis allows significantly lesser number of updates in every iteration then Metis shall be considered as a better scheme for substructure discovery.

Figure 3.1 shows the performance of METIS and Arbitrary partitioning on 5, 10 and 20 iterations of the dynamicAL-SD algorithm. We do not staticAL-SD here as Metis is a variant of arbitrary partitioning and hence update is required. We observe that using Metis only gives a 2-3% overall improvement over arbitrary partitioning. This translates to a maximum of 153 sec runtime improvement even in 20 iterations. Our conjecture for an improvement this small is that the amount of update required by both approaches is almost the same.

To justify our conjecture experimentally we ran 5 iterations of LiveJournal graph on 8 partitions and recorded the number of updates required in each iteration. Table 3.2 shows the number of updates in each iteration. Note that the number of updates by METIS and arbitrary partitioning are almost the same. Moreover, there is no guarantee that METIS will always load lesser number of partitions than arbitrary partitioning. See in iteration 4, arbitrary partitioning ended up loading one

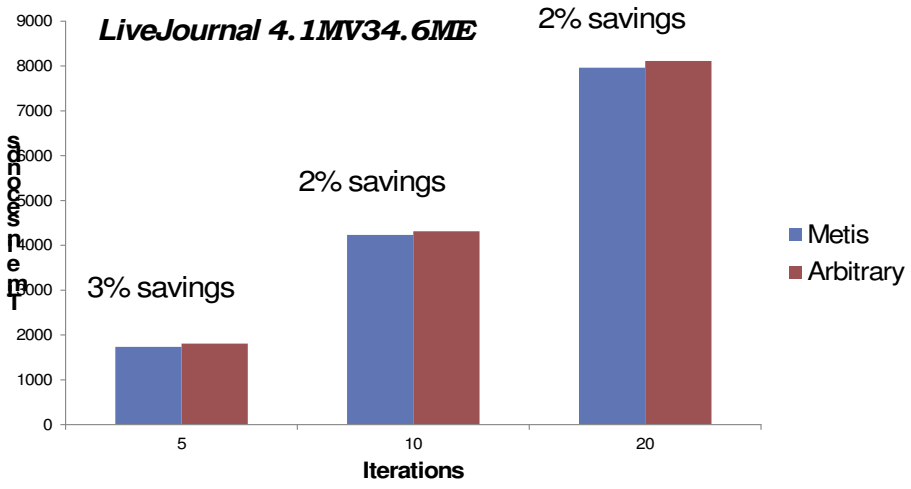


Figure 3.1: Comparison of partitioning strategies (METIS vs. Arbitrary) using DynamicAL-SD

less partition during update than METIS. As the amount of time required to run dynamicAL-SD on both arbitrary partitioning and METIS are almost the same, the need of the hour is to evaluate the partitioning schemes from a creation standpoint.

Table 3.2: Number of Updates METIS vs. Arbitrary

Iteration	METIS	Arbitrary
1	6	7
2	6	6
3	5	6
4	5	4
5	3	5

We now investigate the difference between METIS and arbitrary partitioning from the initial partitioning time perspective. Noe that a multilevel scheme like METIS makes multiple passes over the graph and thereby take more time in partitioning. Moreover, METIS generates vertex based partitions and added post processing

is needed to convert the vertex based partitions to edge based partitions. This post processing is equivalent to expanding a vertex to an 1-edge substructure and then updating the adjacency partitions for correctness. Table 3.3 shows the time takes to partition the LiveJournal graph into 64 partitions.

Table 3.3: Initial Partitioning Cost (METIS vs. Arbitrary)

Component	METIS (gpmetis using random match heuristic)	Arbitrary Partitioning
Type of Partitioning	vertex based	edge based
Passes on graph	1	1
Passes on adjacency list	10	1
Time for partitioning	235 sec	35 sec
Post processing cost	450 sec	0 sec

Note that METIS requires about 8 times more time in partitioning as it needs more passes over the graph. Moreover post processing using METIS takes a huge time. A combination of initial partitioning time along with dynamicAL-SD runtime shows Arbitrary partitioning taking lesser time. The initial cost of partitioning in METIS outweighs the savings obtained by using better quality partitions thereby making arbitrary partitioning a better candidate for substructure discovery than METIS.

3.3.3 Cost Analysis (Arbitrary vs. Ranged Partitioning)

We have already established arbitrary partitioning as a better candidate over METIS. One way to improve dynamicAL-SD is to eliminate the overhead of updating adjacency partitions. An alternative is to keep adjacency partitions fixed and expand a substructure in multiple substructure partitions using a fixed adjacency partition. This will avoid the I/O intensive adjacency partition update after each iteration

but will require grouping substructures differently and sending them to appropriate partitions for expansion. This increases the shuffle cost as the same instance needs to be routed to multiple partitions for expansion. However this also introduces a trade off between the update and shuffle cost. Substructure discovery using range-based partitioning completely eliminates the update cost and will be beneficial if the amount of shuffle increased is lesser than the update cost. Figure 3.2 highlights the specific component costs across 20 iteratiuons in the liveJournal graph. As expected the computation cost grows upto a certain iteration before starting to decrease. However, the update cost is constant across iterations and consumes around 30% of the total cost.

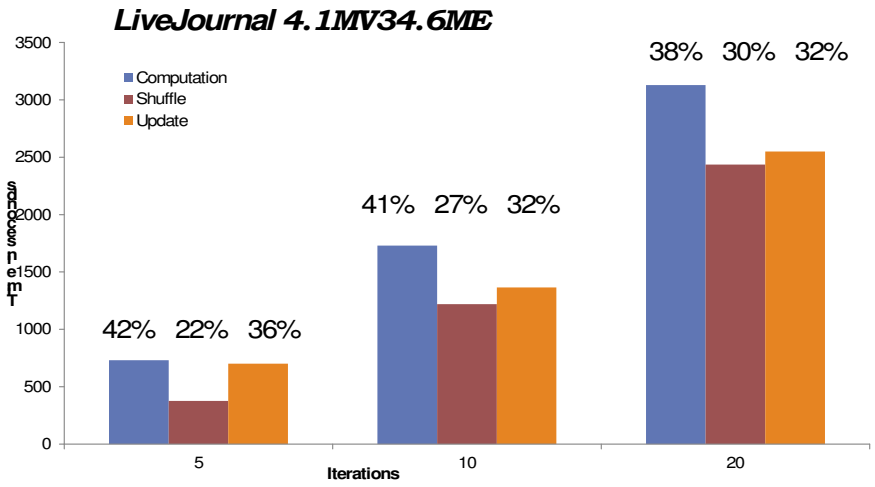


Figure 3.2: Cost breakdown of DynamicAL-SD on 5 iterations using arbitrary partitioning

While using a ranged partitioning scheme, Figure 3.3 shows the total distribution of costs. Note that the update phase has been completely eliminated with some additional increase in shuffle time. The computation cost remained almost the same. Considering shuffle cost only, there has been a 10% increase in shuffle for 5 iterations

and a respective increase of 12% and 18% across 10 and 20 iterations. However, sacrificing the entire update phase for an increase in shuffle led to an overall improvement of 35%, 28% and 27% respectively across 5, 10 and 20 iterations. Therefore further optimizations and improvements were obtained just by switching to a different partitioning scheme.

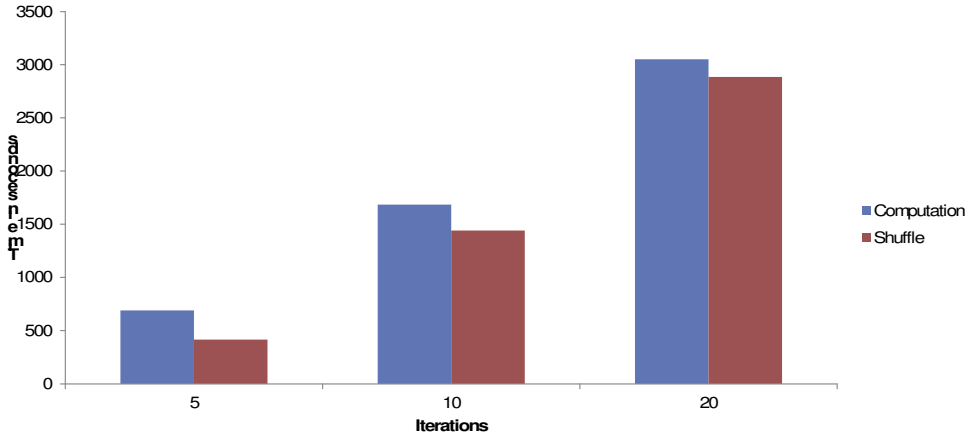


Figure 3.3: Cost breakdown of StaticAL-SD (range-based partitioning) for 5 iterations

Note that the initial partitioning cost for range-based partitioning is similar to arbitrary as it needs a single pass over the graph to partition the vertices based on the ranges. It is also suited for a heterogeneous environment by correlating the ranges to the capacity of each machine in the cluster and can be still done in one pass of the graph. To study the effects of the improvements on varying number of reducers we ran 5 itrations of dynamicAl-SD (which uses arbitrary partitioning) and staticAL-SD (which uses range-based partitioning) on 5 iterations of the LiveJournal graph. Figure 3.4 highlights the improvement in speedup and overall run time of the two algorithms.

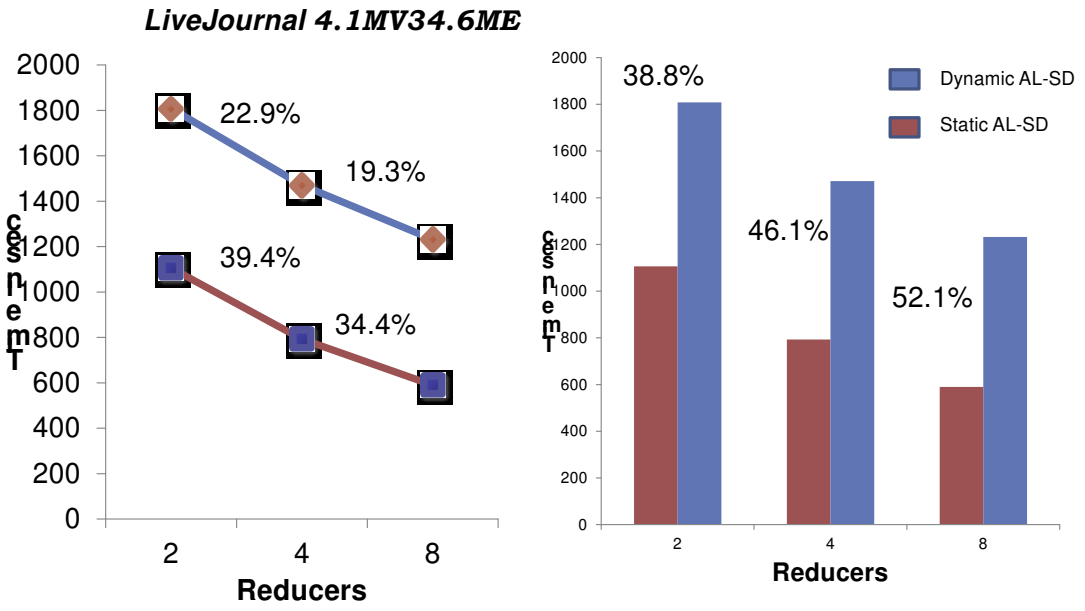


Figure 3.4: Improvements of staticAL-SD over dynamicAL-SD with varying number of processors

The lack of the constant update cost in each iteration gave staticAL-SD much better speedup over its counterpart. With increasing number of processors, the computation cost decreased in each machine but the worst case partitioning cost remained the same. Thereby while increasing number of processors the extent of improvement of staticAL-SD over dynamicAL-SD kept on increasing.

3.4 Variation of Costs

Here we individually discuss the variation of these three costs based on graph characteristics and user-defined parameters.

3.4.1 Varying Graph Sizes

Assume p partitions with r processors in the distributed system. Without adding more machines, as graph sizes increase so does the number of partitions. In

an ideal scenario, the number of partitions are kept equal to the number of processors to extract maximum scalability. As p increases, it is often common to have $p \geq r$. In such a scenario, r partitions are processed at a time resulting in $\lceil p/r \rceil$ sequential batches of processing. In $dynamicAL - SD$, the total update cost is dependent on the number of batches and hence the cumulative update cost increases as number of partitions increase unlike $staticAL - SD$ which does not involve updates. As the update cost is proportional to graph size, improvement between staticAL-SD and dynamicAL-SD should improve with graph size. In both these methods, with increase in graph sizes, the computation cost (and hence the shuffle cost) increases. However, even with increasing graph sizes, the computation and shuffle costs vary based on choice of user parameters.

3.4.2 Varying User Parameters

As explained in Section 4.3, the expansion is constrained in each iteration by storing the top-k substructures and associated instances in beam. The choice of this k or the ***beam size*** affects the costs of mining. With a small sized beam, only very few substructures and associated instances are stored and expanded in subsequent iterations. As the beam size increases, more substructures are stored, thereby increasing the computation and shuffle cost. Note that, the update cost has an upper bound (equal to the number of partitions) and hence is independent of the size of the beam. Increasing beam size however attracts some frivolous substructures and their instances which might not be of use to the end user. Hence a higher beam increases shuffle and computation cost but not the update cost.

Another factor affecting the costs are the ***number of iterations*** (optionally controlled by user.) Keeping beam constant, the number of intermediate substructures in graph mining keeps growing till an iteration before starting to decrease. But

with each iteration, the size of the substructures increase monotonically. Therefore, the total computation cost (and thereby the shuffle cost) is dependent on the size of the substructure and the number of instances in a particular iteration.

3.4.3 Varying Number of Processors

Without altering a graph size, an increase in number of processors will be beneficial for mining. To achieve the best parallelism, the number of partitions (p) are kept equal to the number of processors (r.) Therefore increase in r, leads to smaller sized partitions and the computation cost in each processor is reduced. This reduction in computation contributes to the amount of speedup achieved. Note that the update cost still involves loading other partitions and in the worst case, the entire graph thereby incurring a fixed worst case update cost in each iteration. With increasing processors, $dynamicAL - SD$ ends up with a constant update phase, prompting $staticAL - SD$ to outperform it in speedup as well. As is typical of partitioned approaches, the overhead of managing partitions may dominate if the partition sizes are small.

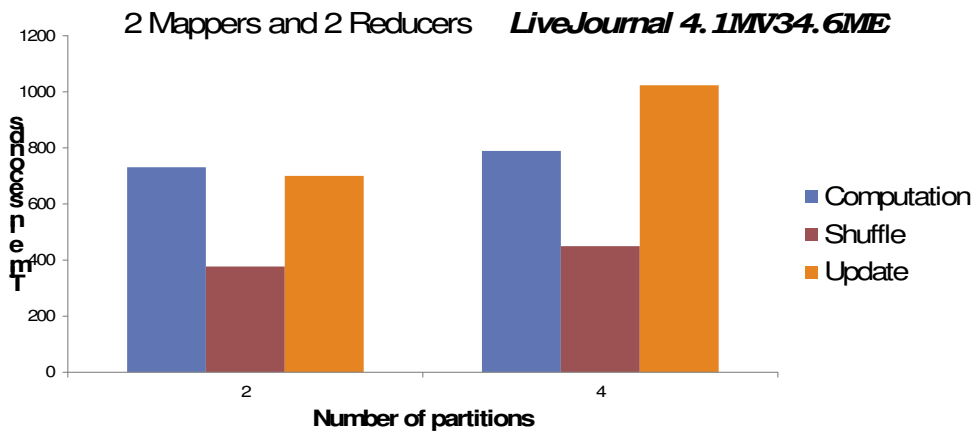


Figure 3.5: Cost analysis on DYnamicAL-SD with fixed processors and varying number of partitions

Note that our algorithms are guaranteed to work even on smaller number of machines. With limited capacity of each processor, each processor can work on a smaller partition. Hence there might be a case when the number of partitions are more than the number of partitions. Figure 3.5 shows how the same number of processors (here 2) can handle multiple partitions of the same graph. Keeping the number of processors fixed, as the number of partitions increase the cumulative computation and shuffle cost remained the same. However the sequential update cost on each of the partitions increased the total update cost. See here that there was a minor increase of 10% in computation and shuffle cost by varying the number of partitions as opposed to a 46% increase (almost doubling) of the update cost. Ranged partitioning with no updates will be a winner here too. Both our algorithms perform best when the number of partitions are equal to or lesser than the number of processors. However, both are correct if we have to deal with bigger graphs with limited number of processors making our algorithms correct and scalable to clusters of any size.

3.4.4 Varying Graph Connectivity

Connectivity of graphs also influences performance of our 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 completely connected graph to a very sparsely connected line graph. With dense graphs, where each vertex is connected to more vertices on the average, the computation cost increases as there are more scopes of expansion. With update cost dependent on the vertices in the graph, typically a dense graph shall offer a better speedup than a sparsely connected graph. Figures 3.6 and 3.7 shows the effect of varying graph connectivity on graphs with different sparsity. These graphs were generated using the subgen graph generator where the dense graph with n nodes has n(n-1) edges while the sparsest graph has only

(n-1) edges. We generated synthetic graphs in this spectrum to test our algorithms on varying levels of connectivity.

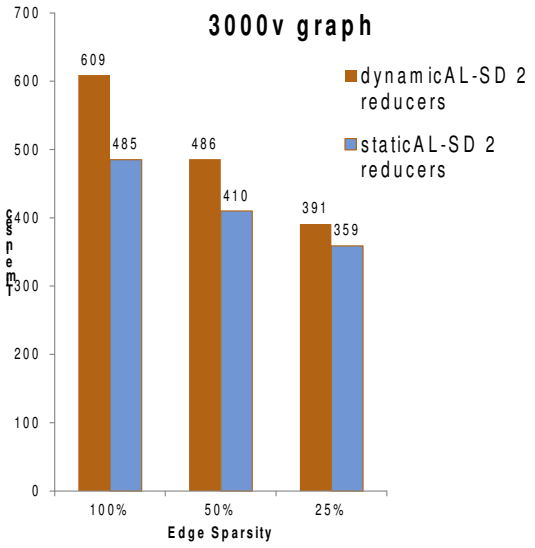


Figure 3.6: Comparison of staticAL-SD and DynamicAL-SD using 2 processors on synthetic graphs of varying connectivity spectrum

With graphs of increasing connectivity, staticAL-SD due to lack of update remained a better performer. Obviously on small graphs the extent of improvement both inn terms of overall time and speedup are less. However moving to larger graphs, better speedup and higher improvement is registered which indicates the effectiveness of our algorithms to scale to graphs of bigger sizes.

3.4.4.1 Effect of partitioning on duplicates

Duplicates are generated when multiple substructures expand to the same substructure. Figure 3.8a shows how duplicates are generated by expanding on the same vertex id while Figure 3.8b shows how the same substructure is generated by expanding on two different vertices (here 5 and 2.) Following range-based partitioning

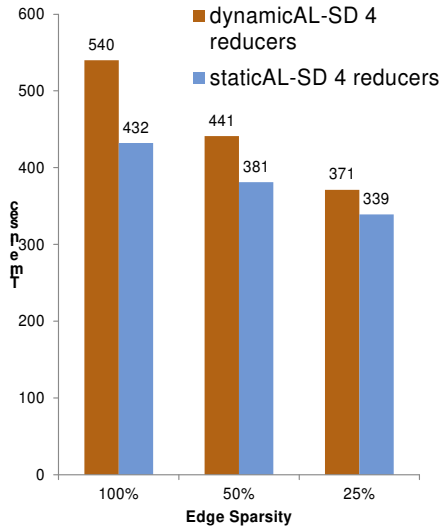
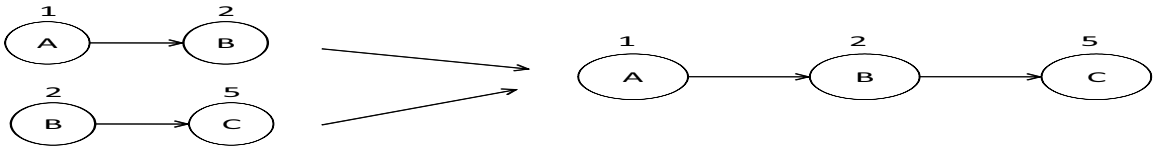


Figure 3.7: Comparison of staticAL-SD and DynamicAL-SD using 4 processors on synthetic graphs of varying connectivity spectrum

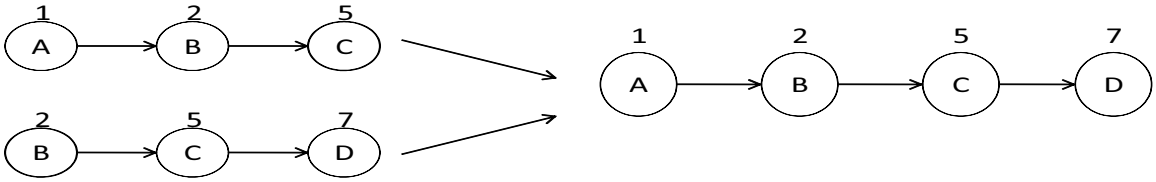
scheme, all expansions on the same vertex use the same adjacency partition hence all these duplicates are guaranteed to be caught in the same partition in $staticAL-SD$. However in $dynamicAL-SD$, there is no guarantee on number of intra partition duplicates to be eliminated. The frequent substructures and instances irrespective of our partitioning scheme needs to be shuffled from mappers to reducers. Therefore the shuffle cost for $dynamicAL-SD$ and $staticAL-SD$ depends on the number of duplicates. There has been effort to eliminate duplicates using heuristics which shall be discussed in the next chapter.

3.5 Classes of Graph Mining and their Costs

Based on our related work on Substructure discovery, mining in graphs can be broadly divided into two categories/classes: non iterative and iterative. Here we discuss the applicability of our costs to these classes of graph mining problems and thereby inspect choice of partitioning for different classes of mining problems.



(a) Example of duplicates expanding on same vertex id(here 2)



(b) Example of duplicates expanding on different vertex id

Figure 3.8: Example of duplicates resulting from expansion

3.5.1 Non-Iterative Graph Mining

on iterative can be conceptualized as joins where the graph is joined with itself or with a smaller subset to find worthwhile information. If a graph is joined with itself, based on the joining attribute, parts of graphs need to be shuffled from mappers to reducers. This is often known as reduce side join. If the joining criteria is known apriori, the graph can be partitioned on the joining criteria. Each mapper can join each partition with itself thereby not needing the reducer and saving the shuffle cost. Being a non iterative approach, the cost of partitioning using the non iterative approach requires at least one pass of the graph. The shuffle cost in a non iterative scenario is equal to the graph size and is same as the lower bound of partitioning. Therefore partitioning should not make much of a difference for non iterative approaches. The computation cost both with and without partitioning remains same as the total joining cost.

3.5.2 Iterative Graph Mining

Iterative graph mining requires the same graph information or updated graph information in each iteration for processing. The iterative graph mining can be subdivided again into two categories:

1. ***Fixed computation cost:*** These are application where in each iteration, the same computation cost is incurred. One example is PageRank where in each iteration, each vertex exchanges information only with its neighbors. Hence the computation cost and the shuffle cost remains fixed across iterations. Necessarily to get the neighborhood information, the adjacency list can be shuffled either across the network increasing the shuffling cost. Another solution is to partition the graph once and reuse the partitions in every iteration for computations. Lin [22] used range-based partitioning following this approach beneficially to save computations on graphs. However as the computations remain same across iterations, there was no need to update partitions. It is worth noting that the savings come from the trade-off between shuffle and loading costs. As the partitions do not chance across iterations, they can be cached and Spark can be used to improve the cost even further.
2. ***Variable computation cost:*** Our substructure discovery falls in this category where the amount of computation in each iteration is dependent on choice of user parameters. Both arbitrary partitioning and range-based partitioning can be used correctly for this purpose. The trade-off between shuffle and update cost once again makes one partitioning scheme better than the other. $staticAL-SD$ with fixed partitions eliminates the update cost completely by trading a slight increase in shuffle cost making it better then $dynamicAL-SD$ using arbitrary partitioning scheme or METIS. Moreover $staticAL-SD$ with fixed adjacency

partitions becomes a better candidate of porting to newer paradigms like Spark making it better than its counterpart.

3.6 Portability to Other Distributed Paradigm

We have used Map/Reduce as the choice of our distributed paradigms but our algorithms can be ported effectively on other paradigms such as Spark [27]. These distributed systems use a distributed file system to store the graph partitions and hence the three costs discussed here with respect to Map/Reduce remains the same in Spark. However Spark brings caching where the same partition can be cached and reused in every iteration. Using $dynamicAL-SD$ in Spark, the adjacency partitions keep changing in every iteration making caching useless. However in $staticAL-SD$ the adjacency partitions once created do not change across iterations making it an ideal candidate for caching. This helps in reducing one load operation in every iteration thereby improving the overall run time. Thus among our two approaches $staticAL-SD$ is a better fit for the Spark paradigm.

3.7 Conclusion

The cost analysis introduced here indicated the level of improvements that can be achieved by tradeoff between various costs. As range-based partitioning eliminates the update cost, further optimizations can be done by investigating techniques of reducing the computation cost. The next chapter optimized substructure discovery even further by introducing heuristics to reduce the computation cost and the accompanying shuffle costs.

Table 3.4: Choice of partitioning for classes of graph mining problems

Problem	Category	Our Analysis
Joins	Non iterative	Partitioning strategy based on join criteria should work best while joining two large tables. While joining a smaller graph with a larger graph (broadcast join), arbitrary partitioning on the larger graph is useful.
Page Rank	Iterative Fixed cost	Arbitrary partitioning can be used as no updates are required. However better response is noted by removing shuffle cost of graph with a single loading of range-based graph partitions at the reducer. Since these partitions do not change, the algorithm can be extended easily to Spark
DynamicAL-SD	Iterative variable cost	Arbitrary partitioning is used. Updates make it unsuitable for spark as the adjacency partitions cannot be cached in every iteration.
StaticAL-SD	Iterative variable cost	Range-based partitioning is better suited as it avoids update. As partitions are not updated in every iteration, they can be cached easily and ported to Spark for even further improvements

CHAPTER 4

OPTIMIZATIONS OVER GRAPH MINING

At the core of graph mining lies *independent* expansion where a substructure (also referred to as a subgraph) independently grows into a number of larger substructures in each iteration. Such an independent expansion, invariably, leads to the generation of duplicates. In the presence of graph partitions, duplicates are generated both within and across partitions. Eliminating these duplicates (for correctness) not only incurs generation and storage cost but also additional computation for its elimination. Our primary aim is to design techniques to reduce generating duplicate substructures as we show that they cannot be eliminated. This paper introduces three constraint-based optimization techniques, each significantly improving the overall mining cost by reducing the number of duplicates generated. These alternatives provide flexibility to choose the right technique based on graph properties. We establish theoretical correctness of each technique as well as its analysis with respect to graph characteristics such as degree, number of unique labels, and label distribution. We also investigate the applicability of their combination for improvements in duplicate reduction. Finally, we discuss the effects of the constraints with respect to the partitioning schemes used in graph mining. Our experiments demonstrate benefits of these constraints in terms of storage, computation, and communication cost (specific to partitioned approaches) across graphs with varied characteristics.

4.1 Introduction

Substructure discovery is the process of discovering substructure(s) (a connected subgraph) in a graph (or a forest) that best characterizes a concept embedded in that graph based on some criterion (frequency, compressibility etc.). Many approaches for substructure discovery have been proposed in the literature. Main memory based [1, 9–12], disk-based [13–15] and database-oriented approaches [16, 17]) address substructure discovery on a single machine. With graphs that overwhelm main memory, partition-based approaches to substructure discovery [25, 26] have also been proposed.

All of the above-mentioned approaches use an iterative algorithm that: generates all substructures of increasing sizes (starting from substructure of size one that has one edge), counts the number of distinct identical (or similar) substructures and applies a metric (e.g. frequency, Minimum Description Length [1], minimum support [10] etc.) to rank them. In each iteration, either all expanded substructures or a subset (using the rank) are carried forward to limit the search space. This process is repeated until a given substructure size is reached or there are no more substructures to generate. This way of expansion grows each node in a substructure in *all possible ways* in each iteration by adding an edge generating many substructures of the next size. This technique is referred to as *independent expansion* where a substructure is expanded unaware of other expansions in the same iteration. Non-independent expansion makes the process sequential, affecting performance and making it not suitable for large graphs and partition-based approaches.

The unconstrained independent expansion is complete as it guarantees generation of all possible substructures in every iteration. However, as a byproduct of independent unconstrained expansion, duplicates are generated when different substructures, in the same iteration, expand into multiple copies of the same (exact)

bigger substructure. The number of duplicates, thus generated, depends on graph characteristics. Following expansion, these duplicates need to be removed to ensure correctness, incurring additional computation cost.

The challenge, therefore, is to augment the unconstrained independent expansion strategy using heuristics which limit the generation of duplicates. Needless to say, these heuristics should be correct (i.e., sound and complete), and generate the same results as an unconstrained expansion. Moreover, heuristics cannot use the knowledge of any other substructure and their expansion details and should retain their independent nature[1]. Hence, reduction of duplicates with low overhead seems more pragmatic than their elimination. If we are able to identify information that characterizes a substructure locally, use that to define a constraint that is sound and complete, and is computationally inexpensive to apply, that would satisfy our requirements.

Correct expansion generates a huge set of intermediate substructures. The number of intermediate results grow exponentially to a certain substructure size before starting to decrease. Users, on the other hand, are interested in mining patterns following some user-defined parameters. Hence some pruning properties (based on user-defined parameters) are typically applied that limit the search space and use the best substructures for further expansion. Therefore, completeness needs to be guaranteed even in presence of these pruning properties.

In this chapter, we augment the independent expansion strategy by introducing three heuristics, each reducing the number of duplicates generated during graph expansion. Our heuristics can be seamlessly integrated into most of the graph representations used for the iterative algorithm. In addition to the theoretical correctness of

[1]Keeping additional information for checking whether an exact match has already been generated is what we are trying to avoid in the first place.

each constraint, we show that they cannot be combined for additional improvement. For scalability, we validate the benefits of using our heuristics on partition-based graphs. Finally, we present an extensive comparative analysis of the constraints with respect to the graph characteristics to help choose the best heuristic for an arbitrary graph.

Contributions: The contributions of this chapter are:

- Identified several constraint-based heuristics to reduce the number of duplicates generated along with their theoretical correctness
- Analysis of two pruning strategies with respect to soundness and completeness using the proposed heuristics
- Comprehensive analysis of the algorithms on single and partitioned graphs with diverse characteristics using the heuristics and pruning properties
- Back-of-the-envelope analysis of the choice of heuristics to be used for an arbitrary graph

RoadMap: The rest of the chapter is organized as follows. Section 4.2 presents some definitions and concepts needed for the work along with the problem statement. Section 4.3 introduces our heuristic based approaches along with theoretical correctness in Section 4.4. Section 4.5 shows our experiments while Section 4.6 concludes the chapter.

4.2 Preliminaries

Here we briefly describe the basics of graph representation, duplicates, exact substructures, ranking metrics and pruning heuristics.

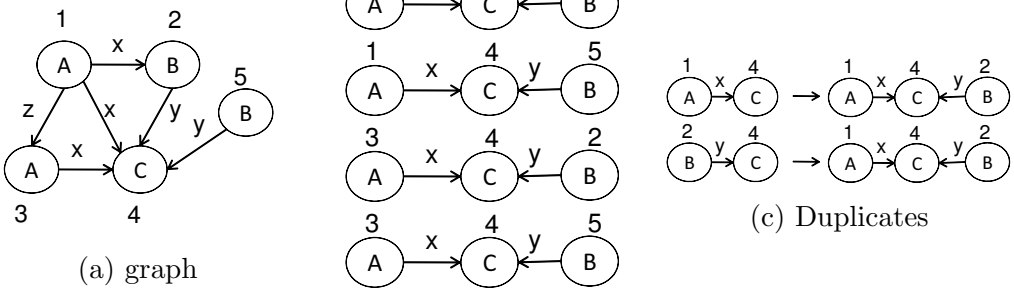
4.2.1 Graph Representation

Definition *An input graph is represented as* $G = (V, E)$ *where* V *is a set of vertex ids and* E *is a set of edges. For each vertex id* $v_i \epsilon V$*, the vertex label is* vl_i*. A directed edge connects a source vertex* v_i *with destination vertex* v_j *where* $1 \leq i, j \leq |V|$ *and has an edge label* e_{ij}*. Formally, every edge in E connecting source* v_i *and a destination vertex* v_j *is represented as 5-element tuple* $< e_{ij}, v_i, vl_i, v_j, vl_j >$*.*

For undirected graphs the smaller vertex id can be used as the source. Support for multiple edges, between a pair of vertices, can be provided by adding a unique edge identifier. A k-edge subgraph is represented as a set of k edges.

Definition *Graphs* $G_1 = (V_1, E_1)$ *and* $G_2 = (V_2, E_2)$ *are* ***isomorphic*** *if (i)* $|V_1| = |V_2|$ *and* $|E_1| = |E_2|$*, (ii) there is a bijection (one to one correspondence) f from* V_1 *to* V_2*, (iii) there is a bijection g from* E_1 *to* E_2 *that maps each edge* $(e_{uv}, u, vl_u, v, vl_v)$ *in* E_1 *to* $(f(e_{uv}), f(u), f(vl_u), f(v), f(vl_v))$ *in* E_2*.*

We use isomorphism to detect identical (or exact) substructure patterns. Figure 4.1b shows 4 isomorphic subgraphs present in the example graph in Figure 4.1a.



(a) graph

(b) Isomorphs

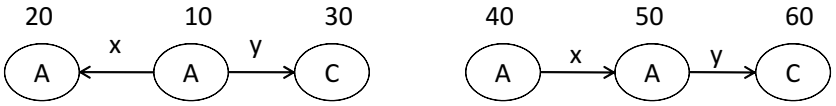
(c) Duplicates

Figure 4.1: An example graph

Definition *Isomorphic graphs* $G_1 = (V_1, E_1)$ *and* $G_2 = (V_2, E_2)$ *are* ***duplicates*** *if* $V_1 = V_2$ *and* $E_1 = E_2$ *and* f *and* g *are identity functions.*

Duplicates are formed when multiple subgraphs expand to the same subgraph. Figure 4.1c shows how duplicates are formed during expansion.

Canonical Instance (CI): A k-edge substructure is represented as an ordered sequence of k edges arranged in the smallest lexicographic order. Such a representation is called a *canonical instance*. While establishing the order, edge labels are given the first preference. In case of ties, the source vertex label, destination vertex label, source vertex id and destination vertex ids are considered, in that sequence. For example the CI of the top most isomorphic substructure in Figure 4.1b is captured as <x,1,A,4,C; y,2,B,4,C> as x comes before y in dictionary order. Note in Figure 4.1b, the isomorphic subgraphs have the same labels but different vertex ids while duplicates as in Figure 4.1c are exactly same in both.



(a) Instance 1 (b) Instance 2

Figure 4.2: Canonical instance and connectivity

Canonical instance representations of two substructures which are duplicates will match exactly. However, one cannot merely remove vertex ids from this representation to check for isomorphism. Consider the example in Figure 4.2a with its CI $< x, 10, A, 20, A;\ y, 10, A, 30, C >$ while Figure 4.2b has a CI $< x, 40, A, 50, A;\ y, 50, A, 60, C >$. If we drop the vertex ids, both these instances have the same ordering of labels $< x, A, A; y, A, C >$. However the instances differ in connectivity. Hence, a CI needs

to be converted into a canonical substructure (CS) that preserves connectivity correctly while checking for exact matches.

Canonical Substructure (CS): A canonical substructure is derived from the canonical instance by replacing each vertex id with their relative positions in the instance starting from one. For our previous example in Figure 4.2, CS for $< x, 10, A, 20, A; y, 10, A, 30, C >$ is derived as $< x, 1, A, 2, A; y, 1, A, 3, C >$ where vertex id 10 is the first vertex id followed by vertex ids 20 and 30. Similarly CS for $< x, 40, A, 50, A; y, 50, A, 60, C >$ is derived as $< x, 1, A, 2, A; y, 2, A, 3, C >$. The inclusion of these relative positions is critical for differentiating the connectivity of the instances.

4.2.2 Ranking and Pruning Alternatives

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 substructures. Oner of the commonly used ranking metrics for graph mining are: *Minimum Description Length (MDL)* and *frequency* to rank the substructures. Minimum description length (MDL) is an information theoretic metric that has been shown to be domain independent and 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. However, instances of a substructure may overlap on vertices (and/or edges) influencing the count of substructures.

Definition *Graphs* $G_1 = (V_1, E_1)$ *and* $G_2 = (V_2, E_2)$ *are* ***overlapping*** *if* $V_1 \bigcap V_2 \neq \emptyset$.

Edge overlap requires vertex overlap. There are two distinct ways of frequency computation: with and without overlap.

Overlap-independent Frequency: Here overlapped instances of substructures are considered as independent occurrences and counted as such. For example in Figure 4.1b, the frequency is computed as four although vertex ids 1 and 4 overlap in the top two substructures and vertex ids 3 and 4 overlap in the bottom two substructures.

In the above, the frequency count is not a true representation of frequency as compared to disjoint substructures with the same vertex and edge labels. Hence, this is not a frequency that is typically used.

Overlap-cognizant Frequency: Here overlapped substructures are not counted independently. It has been shown [28] that MRN (Most Restrictive Node) based counting can be used for counting frequency of overlapped substructures. Also, this supports some interesting properties. Hence, overlap-cognizant frequency is computed using the maximal number of non-overlapping instances for a canonical substructure. MRN [8] is defined below.

Definition *Let $F = \{f_1,...,f_k\}$ be the set of k isomorphisms of a subgraph S(V_s, E_s) in a graph G(V,E). Also let $F(v) = f_1(v), ..., f_k(v)$ be the set that contains the distinct nodes in G whose functions $f_1, ..., f_k$ map a node $v \in V_s$. The overlap-cognizant frequency is calculated as $MRN = min\{t|t = |F(v)|\ for\ all\ v \in V_s\}$.*

For the example in Figure 4.1b, we have F(A) = 1,3, F(C) = 4 and F(B) = 2,5. Therefore, the MRN is min(2,1,2)=1.

For frequency, the above-mentioned ranking metrics are used to order all substructures in an iteration. All substructures generated in iteration i can be carried on to iteration $i+1$. However, this is typically not the case as the search space grows exponentially. Some properties (e.g., anti-monotone) will allow us to limit the search space without sacrificing correctness.

Definition *A function f defined on a subset is called anti-monotone if for all sets X and Y such that* $X \subseteq Y$*, one has* $f(Y) \leq f(X)$

The frequency of substructures captured by Overlap-cognizant frequency (or MRN) is the function (f) here that follows the anti-monotone property [8].

4.2.3 Limiting the Search Space

Following ranking, limiting the search space can be done by carrying forward the substructures using two criterion widely used in the literature on mining: (i) In an iteration, keep the substructures with the *top-k* ranks for expansion or (ii) keep all substructures that has a rank greater than or equal to an user specified threshold(τ).

The major difference between the two pruning properties lies in the substructures preserved in an iteration. To ensure correctness, a pruning criteria should always abide by the *apriori property* [28], that is, if a subgraph is preserved in an iteration, all of its subgraphs must be frequent and preserved in the previous iteration. For a graph in an iteration, $top - k$ does not retain *all subgraphs* in the previous iteration unlike the threshold-based pruning property.

Table 4.1: Effect of a combination of Ranking and Pruning technique on apriori property

Ranking Metric	Pruning Technique	Anti-monotone Property	Completeness
MDL	top-k	no	no
MDL	Threshold	no	no
MRN	top-k	no	no
MRN	Threshold	yes	yes

A combination of ranking and pruning must preserve completeness during exhaustive expansion of substructures in an iteration. Literature [28, 29] shows the

following for a combination of ranking and pruning along with exhaustive expansion as expressed in Table 4.1. Therefore, we focus on a combination of MRN ranking metric and threshold-based pruning technique for our approach.

4.3 Heuristics for Constrained Expansion

Independent substructure expansion in *each iteration* of graph mining generates duplicates. As an instance expands only from connected edges, a line graph can be generated in 2 ways (one from each vertex missing an edge at the end) while a completely connected graph of k-edges can be generated in k ways (from k (k-1)-edge instances each missing an edge out of k edges). Only one copy of each instance needs to be preserved.*For each* substructure of size k being expanded, the number of duplicates vary from 1 to k-1. Duplicate identification using the canonical representation requires either sorting or pairwise comparisons - both of which are expensive for large numbers. This, when done without any additional information generates duplicates even on a single machine or without partitioning the data. When a distributed approach (by partitioning the data set) is used, where different substructures expand in different machines (or at different times) generation of duplicates cannot be avoided. Following this expansion a connected substructure grows in each iteration by adding an edge. To maintain completeness, unconstrained expansion needs to be used which generates a large number of duplicates. The number of duplicates vary from 1 (a line graph) to k-1 (a completely connected k-edge substructure). Identification of duplicates and removal of all except one makes sure the same substructure is not expanded more than once in the next iteration.

Following independent expansion, duplicates impact the computation and storage costs. First, generation of all duplicates incurs extra computing cost. Second, the expanded substructures (including duplicates) may need to be persisted before elimi-

nating all but one instance of the duplicate. In the presence of partitions, the number of duplicates generated in individual partitions affect the network cost as these duplicates need to be brought together for elimination. If there is a way of reducing the number of duplicates, performance can be significantly improved across all components of the algorithm (storage, I/O, network and/or shuffling, and computation cost.)

Hence, identification of heuristics which can be applied locally to each instance to reduce duplicates and exhibit completeness globally is needed. For any substructure representation, the local information available are: vertex ids, vertex labels, and edge labels.

We explore each available information as candidates for designing heuristics for constrained expansion strategies to aid in duplicate reduction. The goal is to avoid generating as many substructures/instances as possible that need to be pruned later (we shall use substructure and instance interchangeably where the interpretation is clear from the context.) However, the mechanisms used to prevent unnecessary expansion should satisfy the following properties:

- Soundness: It should not eliminate any substructure that is not a duplicate.
- Completeness: It should generate all substructures that should be generated.
- Low overhead: The cost of applying the heuristic should be significantly less than the cost of generating and removing duplicates.

Below, we present three heuristics, along with associated conditions to reduce the number of duplicates generated. We shall establish the soundness and completeness of each of our heuristics. We postulate that with *independent expansion approaches*, it is not possible to completely eliminate the generation of all duplicates using heuristic(s) without incurring cost that is proportional to duplicate detection. We shall also show that our heuristics reduce but not eliminate all duplicates.

4.3.1 Identification of Heuristics

As defined earlier, an edge consists of five elements: *edge label*, *vertex labels* and *vertex ids*. Each vertex has a unique id while labels (both vertex and edge) are not necessarily unique. Note that for the same substructures to occur multiple times in a graph, it is mandatory for edge and vertex labels to repeat. We introduce a heuristic based on each of these edge components. We prove correctness (soundness and completeness) for each heuristic and analyze their effectiveness. Further, we analyze if a combination of them can be combined to reduce duplicates further. We shall use the illustrative example graph in Figure 4.3 to provide an intuitive understanding of the heuristics. Before analyzing the correctness for heuristics we first discuss the correctness of the unconstrained expansion.

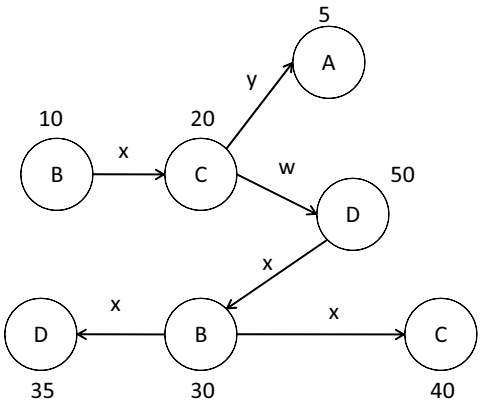


Figure 4.3: Example graph for explaining optimization using heuristics

Correctness of Unconstrained Expansion: The unconstrained expansion is both complete and sound. It is complete as every node in a substructure is expanded in every possible way in each iteration. That is all possible expansions are generated without fail. It is sound as each expansion is correct with respect to the given graph. No incorrect expansion takes place. The downside of the above unconstrained expansion is that the same (or duplicate) substructure is generated by

multiple substructures due to independent unconstrained expansion on each node in every substructure.

As our heuristic-based expansion prevents some substructures from being generated, completeness needs to be established to show that a substructure eliminated during one substructure expansion is indeed generated by another substructure in the *same iteration* using the same heuristic. Soundness is not an issue as all possible expansions are still attempted. Hence, we need to establish only the completeness for each of the expansion technique using a heuristic.

4.3.1.1 Heuristic Based On Vertex Id

We have information on the vertex ids in the substructure to be expanded, the vertex id on which it is expanded (by adding an edge) and the vertex id in the added edge. We formulate a heuristic based on these vertex ids.

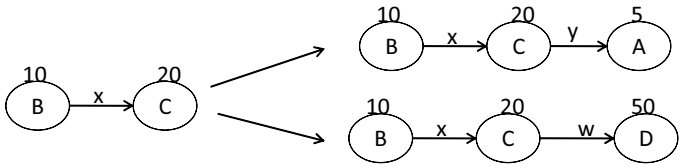
Definition *The minVid of a substructure containing the set of k vertex ids as* $V = \{v_1, v_2, ..., v_k\}$ *is defined as* $min(V)$.

Basically, *minVid* of a substructure is the smallest vertex id present in *that* substructure. For example in Figure 4.4, the $minVid$ of <x,10,B,20,C> is 10 while the $minVid$ of the subgraph in Figure 4.3 is 5.

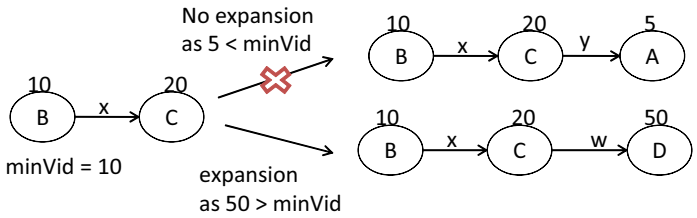
minVid heuristic: *Expand a substructure by adding an edge on any of its vertices if the added vertex id is greater than or equal to its minVid.* The intuition here is to restrict expansions using $minVid$[2].

Figure 4.4 shows a subset of 1-edge substructures from Figure 4.3. In Figure 4.4a, <x,10,B,20,C> is expanded on vertex 20 without using any constraint in all possible ways. In Figure 4.4b the same substructure is expanded using the $minVid$

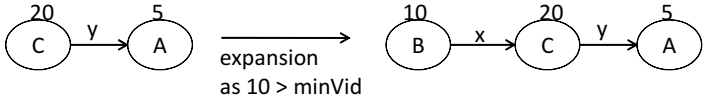
[2]A similar constraint can be formulated with the maximum vertex id (maxVid). Expansion will occur if the added vertex is $\leq$ the $maxVid$ of a substructure



(a) Without constraint



(b) Using minVid constraint



(c) Completeness example

Figure 4.4: Expansion with vertex id based constraint

heuristic when <x,10,B,20,C> (with a $minVid$ of 10) is not expanded by adding vertex 5 but expanded by adding the vertex id 50. The substructure eliminated by the constraint is a valid substructure and must be generated by some other substructure *in the same iteration*. In the same iteration, Figure 4.4c shows another 1-edge substructure <y,20,C,5,A> generating the substructure which was eliminated by constrained expansion in Figure 4.4b by applying the same condition. This substructure has a $minVid$ of 5 and is expanded by adding vertex id 10 which is greater than the current $minVid$. We theoretically establish the completeness of $minVid$ constraint in Section 4.4. Detailed analysis of costs of overhead using $minVid$ is in Section 4.3.2.

4.3.1.2 Heuristics using Vertex Label

Analogous to the vertex ids, we have vertex label information present in the substructure to be expanded and the vertex label of the added vertex. We formulate a heuristic based on the vertex label information.

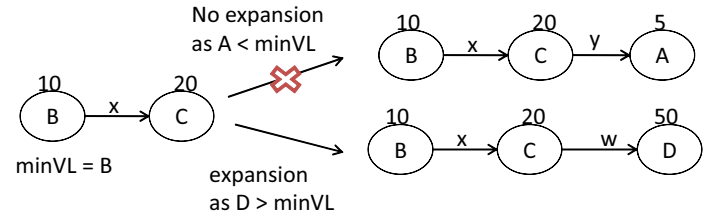
Definition *The $minVL$ of a substructure containing the set of k vertex labels as $VL = \{vl_1, vl_2, ..., vl_k\}$ is defined as $min(VL)$. Note that labels, unlike vertex ids, are not unique. Hence $vl_i = vl_j$ for some $i \neq j$*

Basically, *$minVL$* of a substructure is the smallest vertex label present in *that* substructure. For example in Figure 4.4, the $minVL$ of <x,10,B,20,C> is B while the $minVL$ of the subgraph in Figure 4.3 is A.

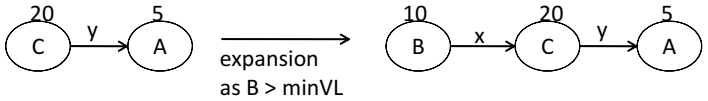
minVL Heuristic: *Expand a substructure with minVL by adding an edge only if the added vertex label is greater than or equal to the minVL*

The intuition here (similar to the one with minVid) is to restrict expansions using a constraint: expand a substructure on any of its vertices only if the added vertex label associated with the added edge is greater than or equal to the minVL of the substructure. A similar heuristic can be established with the maximum vertex label (maxVL.) Expansion will occur if the added vertex has a label smaller than the $maxVL$ of a substructure.

Figure 4.5 shows a subset of 1-edge substructure from Figure 4.3. In Figure 4.5a $< x, 10, B, 20, C >$ is expanded by adding vertex with vertex label D but not by adding the vertex label A. Figure 4.5b shows how the substructure which missed generation in Figure 4.5a is now generated in the same iteration from a different substructure in the same iteration. In section 4.4 we theoretically establish the completeness of the $minVL$ constraint during substructure expansion. Section 4.3.2 contains detailed discussion on the overhead while using $minVL$ heuristic. We can



(a) Using minVL constraint



(b) Completeness example with minVL

Figure 4.5: Expansion with vertex label based constraint

formulate similar heuristics ($minEL$ and $maxEL$) based on the edge label information.

4.3.1.3 Heuristics using Edge Label

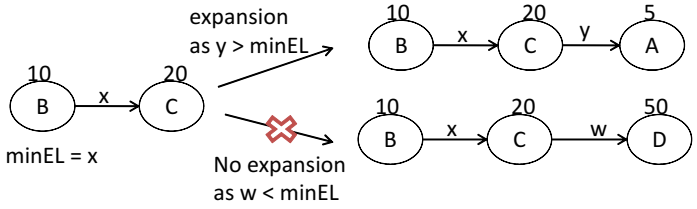
We have the information on the edge label present in the substructure to be expanded and the edge label of the added edge. We can formulate a similar heuristic based on the edge label information. We exclude the details here for lack of space.

Definition *The $minEL$ of a substructure containing the set of k edge labels as $EL = \{el_1, el_2, ..., el_k\}$ is defined as $minEL = min(EL)$. Note that labels, unlike vertex ids are not unique hence $el_i = el_j$ for some $i \neq j$*

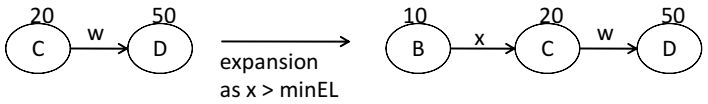
Basically, *minEL* of a substructure is the smallest edge label present in *that* substructure. For example in Figure 4.4, the $minEL$ of <x,10,B,20,C> is x while the $minVL$ of the subgraph in Figure 4.3 is w.

minEL Heuristic: *Expand a substructure with minEL by adding an edge only if the added edge label added is greater than or equal to the minEL*

The intuition here (similar to the one with minVL) is to restrict expansions using a constraint: expand a substructure on any of its vertices only if the added edge label is greater than or equal to the minEL of the substructure [3]. Proof of completeness of $minEL$ heuristic will be similar to the other two heuristics.



(a) Using minEL constraint



(b) Completeness example with minEL

Figure 4.6: Expansion with edge label based constraint

In Figure 4.6a $< x, 10, B, 20, C >$ is expanded by adding edge with vertex label y but not by adding the edge label w. Figure 4.6b shows how the substructure which missed generation in Figure 4.6a is now generated in the same iteration from a different base substructure. Below we theoretically establish the completeness of the minEL constraint in absence of any pruning properties.

4.3.2 Analysis of Proposed Heuristics

We have 3 heuristics each with max and min variants (making a total of 6.) All of them have been shown to be correct and hence any one of them can be chosen for a given graph. This entails an analysis of these heuristics in terms of their cost and

[3] A similar constraint can be designed with the maximum edge label (maxEL)

their ability to reduce the number of duplicates based on graph characteristics. Also, whether we use the *min* or *max* version, the additional information stored and used for each substructure need not be recomputed after each expansion. It is determined once in the first iteration.

4.3.2.1 Comparison Cost and Storage Overhead

Typically, a k-edge substructure has k edge labels and k+1 vertex ids and vertex labels. Therefore, the edge label heuristics require one less comparison per substructure in each iteration than their counterparts. With the substructure being represented in their canonical form, the first edge label of the first edge in the canonical representation is *always* the $minEL$. This makes the $minEL$ heuristic the one with the least overhead.

However, the other heuristics need an overhead for storing the min (or max) values. For $minVid$ (and $maxVid$) the storage overhead for a substructure is an integer, while $minVL$ (and $maxVL$) and $maxEL$ it is a label (which is a string) whose size is application dependent. String comparison is in general more expensive than integer comparison and also depends on the length of the string which may vary across graphs. Hence, $minVid$ is the least cost heuristic where as $minEL$ has the least storage overhead. Since each heuristic is complete in an iteration, different heuristics can be used in different iterations depending on graph characteristics with appropriate costs.

4.3.2.2 Effect of Degree Distribution

In each iteration, each substructure is expanded on all nodes in all possible ways. Hence, the higher the degree of vertices in the subgraph, the more expansions it takes part in. The numbering of vertices also plays a role in the application of the

proposed heuristics. Vertex degree distribution can be analyzed in two ways: (i) using the average degree of the graph and (ii) the edge density distribution

Average degree holistically captures the ratio between edges and vertices in a graph. However it fails to capture the variation of edge connectivity with vertex ids. Edge density distribution, on the other hand, depicts the connectivity for each vertex in the graph. For graphs with same edge density distribution, irrespective of average degree, the heuristics should behave similarly.

For an edge density distribution where smaller vertex ids have much higher connectivity than larger vertex ids, most substructures shall have small vertex ids as $minVid$. Hence, using $minVid$ as a choice of heuristics ends up with more expansions. For this case, use of $maxVid$ is a better choice over $minVid$. For an edge distribution opposite to the above, $minVid$ is preferred over $maxVid$. The degree distribution does not affect labels, hence the performance of label based heuristics should not vary for graphs with different degree distributions.

4.3.2.3 Effect of Label Distribution

For a given degree distribution, label distribution (uniform or skewed) does not effect the performance of id-based heuristics. Furthermore, if the frequency of labels are uniformly distributed over the graph, even the label-based heuristics (either max or min variant) are likely to give similar (if not identical) performances.

However, in case of a non-uniform label distribution in a graph, if lexicographically smaller vertex labels are disproportionately more in number than the lexicographically larger labels, most substructures shall have smaller $minVL$ thereby reducing the applicability of the $minVL$ heuristic and will not eliminate as many duplicates. In this situation, $maxVL$ will perform better. The opposite is true if the label distributions are reversed.

This analysis (as well as the experimental results shown in Section 4.5.3) clearly indicates that the label distribution can play a significant role on the choice of heuristics for maximizing the performance of the heuristics for a given graph. It is also possible that the label distribution can vary over multiple ranges of values. Hence, some form of histogram analysis of labels will help in determining the choice of heuristic to match the graph characteristic. This observation and label analysis can also be beneficially used for choosing the appropriate heuristic for each (or a consecutive number) of iterations to maximize the performance of the algorithm.

4.3.2.4 Effect of Cycles, Loops, and Multiple Edges

When a substructure expands to a cycle or a loop, addition of an edge connects to an existing vertex of that substructure. Therefore, the added vertex and the added vertex label is always greater than or equal to the $minVid$ or $minVL$ of the substructure (or lesser than or equal to the $maxVid$ or $maxVL$ of the substructure.) Therefore these 4 heuristics will not reduce duplicates when the expansion results in a new cycle or loop in the expanded substructure. However $minEL$ or $maxEL$ does not have this limitation and will eliminate duplicates as the added edge label is *not* a part of the substructure being expanded.

Graphs with multiple edges between two nodes need an additional edge id to discriminate edges between a pair of nodes. Our model can be extended to handle multiple edges by using edge id-based heuristics. This is similar to the use of vertex ids. Since edge ids are also unique, the heuristics designed with edge ids will perform similar to the vertex id heuristic.

We would like to point out that the heuristics will perform as expected even in the presence of cycles, loops, and multiple edges as only a small percentage of substructures are affected by this. The correctness is not an issue and the heuristics

can be used with slightly reduced performance if the graph characteristics are not known.

4.3.2.5 Effect of Partitions

A commonly used approach to achieve scalability and speedup of graph mining is to develop mining algorithms on partitioned graphs. There are a number of graph partitioning schemes proposed in the literature [23, 30]. In addition to correctness, we would like to see that individual partition performances are not skewed resulting in reduced overall speedup.

Some of the earlier discussion on graph characteristics are individually applicable not only to each partition, but across partitions as well. For example, if a graph is, say, ordered monotonically on vertex ids (e.g., using a breadth-first order) and then partitioned using ranges of vertex id, there will be partitions with contiguous small value for vertex ids and also partitions that have contiguous larger vertex ids. With the use of $minVid$ heuristic, the partitions with smaller vertex ids will allow more expansions (i.e., will not eliminate as many duplicates) than the ones with larger vertex ids. This can introduce a skew (assuming all other things being equal) in the computation cost for an iteration across partitions. Use of $maxVid$ introduces a similar skew that is complementary to the above. Non-uniform computation costs across partitions (irrespective of how it happens) is not congenial for parallelism and scalability. For these scenarios, label-based heuristics which are independent of the vertex ids across partitions will be a better choice.

Similarly, if the labels are ordered monotonically and the graph is partitioned based on ranges of labels, skew will be introduced while using the label based heuristics. Vertex id-based heuristics will then be preferred over label-based heuristics.

In presence of a skew, some partitions generate more duplicates than others. However, these duplicates inside a partition are removed by combiner (in a map/reduce framework.) The shuffling cost only depends on the unique number of inter partition duplicates which is insignificant compared to the processing cost in each partition.

Skewness in computation can be overcome in different ways based on information collected during partitioning. For example, the size of the partition can be determined to mitigate skewness in computation. As each substructure is expanded only once in a partition, number of total duplicates generated is independent of the size of the partitions for a chosen heuristic. Also for an arbitrary partitioning scheme, using the random assumption of labels, the performance of a heuristic will follow a similar trend in each partition as the performance of the same heuristic on an the graph.As a substructure can be grown into from different partitions, use of different heuristics across partitions in the same iteration will not guarantee completeness.

4.3.2.6 Summary

Based on the above analysis, it would be useful to identify, if possible, a heuristic that is good in the absence of graph characteristics and also match a heuristic when some of the graph characteristics are known. Although $minEL$ achieves the lowest storage cost and handles cycles and loops effectively, it is dependent on the label distribution of the graph and also the partitioning scheme (specific to partitioned graphs.) However, while creating the initial adjacency list for the graph[4], the degree distribution and label distribution can be obtained and the proper heuristic can be chosen appropriately. For an arbitrary graph Table 4.2 summarizes the choice of heuristics (ranked in increasing order of preference) for graph characteristic discussed

[4]whether partitioned or not, adjacency list is used for expansion of a subgraph in each iteration.

earlier. Same preference of two heuristics indicate a tie for that graph characteristic.

Table 4.2: Ranked choice of heuristics based on graph characteristics

Graph Characteristic	Ranking of Heuristics					
	min Vid	max Vid	min VL	max VL	min EL	max EL
Uniform (edge/vertex) label distribution	3	3	2	2	1	1
Unknown degree Distribution	3	3	2	2	1	1
All vertex and edge labels same	1	1	-	-	-	-
Higher percentage of smaller vertex labels	3	3	4	1	2	2
Higher percentage of smaller edge labels	3	3	2	2	4	1
Higher percentage of larger vertex labels	3	3	1	4	2	2
Higher percentage of larger edge labels	3	3	2	2	1	4
Overall	3	3	2	2	1	1

An absence of rank indicates no improvement using a heuristic. Note that, there is no column in Table 4.2 which always gives the best rank. From this analysis, it is evident that there is not a single heuristic among the 6 heuristics proposed which achieves the best performance improvement for any arbitrary graph. The last row captures the ranking of a heuristic for an arbitrary graph. Edge label based heuristics (*min* or *max* depending on edge label distribution) are mostly the winner in all cases. Only when smaller vertex labels are disproportionately present in high frequency, vertex label based heuristics are chosen. Id based heuristics are only chosen when label-based heuristics fail (as in the case of all labels being same in the graph.) Since label based heuristics are independent of degree distributions, a combination of graph characteristics will be dependent on the label distributions in the graph.

4.3.3 Combining Heuristics

An interesting issue would be to see whether a combination of more than one heuristics (e.g., $minVid$ and/or $minVL$) improves the number of duplicates eliminated and still preserves completeness. This would be a trade-off between overhead and savings in terms of duplicates reduced. The proposed heuristics, when applied individually, decide whether to expand a substructure or not (yes or no.) Considering two heuristics $h1$ and $h2$, the decisions made by them while expanding the same substructure using the same edge are shown in Table 4.3.

Table 4.3: Possible Outcomes of two heuristics on a subgraph for the *same* edge expansion

Case	Expand using h1	Expand using h2
1	yes	yes
2	no	no
3	yes	no
4	no	yes

When both heuristics give the same decision, the choice is clear (as in cases 1 and 2.) As the heuristics use different, independent components of the graph (vertex id, vertex label, and edge label), there is no guarantee that their decisions will be the same. For example in Figure 4.7 $minVL$ and $minVid$ used on the same substructure for expansion gives two different decisions for expansion. Hence, one has to combine these decisions using an operator such as Boolean "and" or "or". Using the "or" operator (i.e., allowing the expansion if one of the heuristic indicates yes) is worse as more duplicates are likely to be generated. Use of "and" operator does not satisfy the completeness property making it useless.

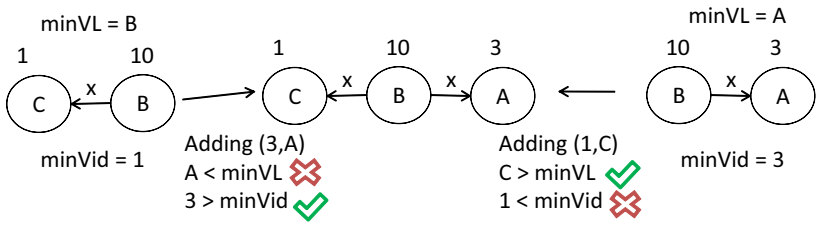


Figure 4.7: Different decisions by multiple constraints ($minVid$ and $minVL$) on same substructure

Based on the above, we can conclude that combining more than once heuristic for duplicate reduction is not beneficial. Hence one needs to wisely choose only one heuristic to apply. However, this does not imply that the *same* heuristic need to be used in each iteration. AS each heuristic is complete with respect to an iteration, heuristic can be changes across iterations. This will be beneficial if the graph characteristics of the subgraphs generated can be inferred during the expansion in an iteration.

4.4 Theoretical Proofs

In this section, we formally establish the completeness of the proposed heuristics.

4.4.1 Completeness using vertex id based constraints

Theorem 4.4.1 $minVid$ heuristic satisfies completeness in each iteration

Proof. We shall prove this theorem using induction on the size of the substructure.

Base Case: Consider the base case with the smallest graphs (or 1-edge substructures.) Consider 2 different substructures with vertex ids $< v_i, v_j >$ and $< v_j, v_k >$ which can be independently expanded. Following unconstrained expansion, for any combination of v_i,v_j and v_k, both the substructures expand to $< (v_i, v_j)(v_j, v_k) >$ in

the first iteration. This unconstrained expansion results in generating one duplicate as the same 2-edge substructure is generated by both 1-edge substructures. Without loss of generality, given three unique values (verted ids), Table 4.4 shows all possible relationships for the values of vertex ids. There are only 6 cases as shown in the combination column. We need to establish that for each of these 6 possible cases, at least one correct 2-edge substructure is generated when the $minVid$ constraint is applied.

Table 4.4 shows how expansion happens for any combination of v_i,v_j and v_k in the last two columns of the table. For example in case 1, v_i is the $minVid$ of $< v_i, v_j >$ and expansion goes through as $v_k > v_i$ in column 3. But the $minVid$ of $< v_j, v_k >$ is v_j and is not expanded by adding v_i as $v_i < v_j$ as shown in the column 4 of the table. Any expansion in case of 1-edge substructures, has to follow one of these 6 cases, and at least a 2-edge substructure is generated in each case guaranteeing correctness. Note that for cases 3 and 4, duplicates are still generated. Note also that the generation of a duplicate is not done using this constraint in 4 out of 6 possible cases. This also supports our earlier statement that total elimination of duplicates is not possible using independent expansion and heuristics based on local substructure information. Although we have discussed this for two 1-edge substructures, the 1-edge substructures of any arbitrary connected graph can be broken down into a number of pairs of 1-edge substructures and the same logic can be applied for each pair.

Induction Case: Now consider a n-edge substructure S to be expanded to (n+1)-edge substructures. Let the set of vertex ids for S is be $V = \{v_1, v_2, ..., v_{n+1}\}$. Let the $minVid$ of this substructure be m=min(V). When this substructure is expanded, a vertex id v is added to this substructure to form S'. The vertex id set of S' is V' and can be defined as $V' = \{v_1, v_2, ..., v_{n+1}, v\}$. Addition of v can result in 2 cases.

Case 1: $v \geq m$. The expansion takes place as it is allowed by the constraint. As this

Table 4.4: Completeness of base case using minVid

Case	Combination	$< v_i, v_j >$ expands to $<(v_i, v_j)(v_j, v_k)>$	$< v_j, v_k >$ expands to $<(v_i, v_j)(v_j, v_k)>$
1	$v_i < v_j < v_k$	yes	no
2	$v_i < v_k < v_j$	yes	no
3	$v_j < v_i < v_k$	yes	yes
4	$v_j < v_k < v_i$	yes	yes
5	$v_k < v_i < v_j$	no	yes
6	$v_k < v_j < v_i$	no	yes

expansion takes place we do not have to show anything else.

Case 2: $v < m$. Here S is not expanded to S'. Note that, since m is the $minVid$ of the substructure, $v_j \geq v$ for any $v_j \in$ V. Here we have to show that this substructure will be generated in this iteration by another substructure in the presence of the constraint. For this, there has to be a n-edge substructure $S1$ that contains the edge $< v_j, v >$ where $1 \leq j \leq n$ and is missing an edge $< v_{j1}, v_{j2} >$ that is in S where $1 \leq j1, j2 \leq n + 1$. Such a substructure has to exist due to completeness during the previous iteration. Also, the $minVid$ of that $S1$ is v as v is smaller than all other vertex ids. Note that $S1$ is expanded to S' by adding the edge $< v_{j1}, v_{j2} >$, as either $v_{j1} > v$ or $v_{j2} > v$. So S' is not generated from S but from $S1$ proving the completeness of $minVid$ constraint for the general case.

Figure 4.8 illustrates representation of case 2 diagramatically. When vertex id v was added to S to create S' where $v < minVid$ (here m), expansion did not happen. However another connected substructure $S1$ (this substructure has to exist as completeness is true for the previous iteration) with a $minVid$ of v generated this missing substructure by adding vertex v_k where $v_k \geq v$ thus asserting correctness.

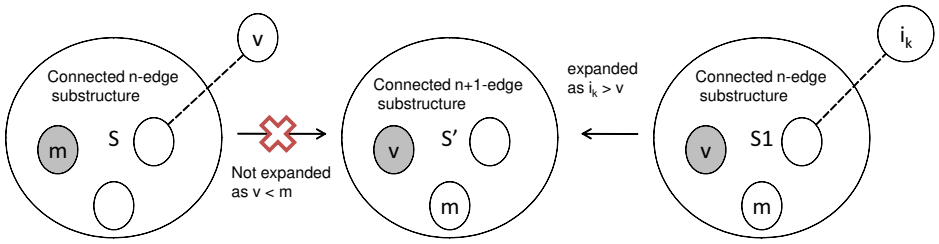


Figure 4.8: Illustration of completeness using minVid for the induction step

4.4.2 Completeness using vertex label based constraints

Theorem 4.4.2 minVL heuristic is complete in each iteration with respect to substructure generation

Proof. We shall prove this using induction which is very similar to the previous one. We denote the vertex label of a vertex id v_i as vl_i. Consider the base case, when two substructures $< v_i, v_j >$ and $< v_j, v_k >$ expand to $< (v_i, v_j)(v_j, v_k) >$. Without loss of generality, given three label values, Table 4.5 shows all possible relationships for them. There are 13 cases (6 cases numbered 1 to 6 where all label are different, 6 cases from 7 to 12 where two of them are equal and 1 last case where all three labels are the same.) We need to establish that in each of these 13 possible cases, at least one correct 2-edge substructure is generated on application of the $minVL$ constraint.

Table 4.5 shows how expansion happens for any combination of vl_i, vl_j and vl_k in the last two columns of the table. Note that in some cases duplicates are still generated. This also supports our earlier statement that total elimination of duplicates is not possible using independent expansion with heuristics.

Now consider a n-edge substructure S to be expanded to a (n+1)-edge substructure S'. The set of vertex ids for S is defined as V = $\{v_1, v_2, ..., v_{n+1}\}$. The set of vertex labels is defined as VL = $\{vl_1, vl_2, ..., vl_{n+1}\}$. Let the $minVL$ of this substructure be m=min(VL). When this substructure is expanded, a new vertex v with vertex label

Table 4.5: Completeness of base case using minVL

Case	Combination	$< v_i, v_j >$ expands to $<(v_i, v_j)(v_j, v_k)>$	$< v_j, v_k >$ expands to $<(v_i, v_j)(v_j, v_k)>$
1	$vl_i < vl_j < vl_k$	yes	no
2	$vl_i < vl_k < vl_j$	yes	no
3	$vl_j < vl_k < vl_i$	yes	yes
4	$vl_j < vl_i < vl_k$	yes	yes
5	$vl_k < vl_i < vl_j$	no	yes
6	$vl_k < vl_j < vl_i$	no	yes
7	$vl_i = vl_j$ and $vl_i < vl_k$	yes	yes
8	$vl_i = vl_j$ and $vl_k < vl_i$	no	yes
9	$vl_i = vl_k$ and $vl_k < vl_j$	yes	yes
10	$vl_i = vl_k$ and $vl_j < vl_k$	yes	yes
11	$vl_j = vl_k$ and $vl_j < vl_i$	yes	yes
12	$vl_j = vl_k$ and $vl_i < vl_j$	yes	no
13	$vl_i = vl_j = vl_k$	yes	yes

vl_v is added to the substructure to form S'. Addition of v and hence vl_v can result in 2 cases.

Case 1: $vl_v \geq m$. The equality condition can happen as vertex labels are not unique. The expansion takes place following the constraint. We do not have to show anything as there is no elimination.

Case 2: $vl_v < m$. Here S is not expanded to S'. Note that $vl_i \geq vl_v$ for any vl_i in VL. Here we have to show that the substructure will be generated in the same iteration from another substructure in the presence of the $minVL$ constraint. Such a substructure S1 has to exist due to completeness during the previous iteration. This substructure S1 has the following properties: (i) It contains the edge $< v_j, v >$ and (ii) It is still connected but missing an edge $< v_{j1}, v_{j2} >$ from S where $1 \leq j, j1, j2 \leq n+1$. The $minVL$ of $S1$ is vl_v. Hence $S1$ is expanded to S in the same iteration as either $vl_{j1} > vl_v$ or $vl_{j2} > vl_v$. So S' is not generated from S but from $S1$ proving the completeness of the $minVL$ constraint for the general case.

4.4.3 Completeness using edge label based constraints

Theorem 4.4.3 *minEL* heuristic is complete in each iteration with respect to substructure generation

Proof. This can be proved using induction on the size of substructures. Consider the base case, with the smallest graphs (1-edge substructure) where two edges e_i and e_j expand to form 2-edge substructure $< e_i, e_j >$. Let the edge label of an edge e_i be denoted as el_i. Table 4.6 shows all the possible 3 combinations of edge labels and how expansion happens for any of those combinations.

Table 4.6: Completeness of base case using minEL heuristic

Case	Combination	e_i expands to $< e_i, e_j >$	e_j expands to $< e_i, e_j >$
1	$el_i < el_j$	yes	no
2	$el_j < el_i$	no	yes
3	$el_i = el_j$	yes	yes

Now consider a n-edge substructure S to be expanded to a (n+1)-edge substructure S'. The set of edges for S is defined as E = $\{e_1, e_2, ..., e_n\}$. The set of edge labels is defined as EL = $\{el_1, el_2, ..., el_n\}$. Let the *minEL* of this substructure be m=min(EL). When this substructure is expanded, a new edge e with edge label el_e is added to the substructure to form S'. Addition of e and hence el_e can result in 2 cases.

Case 1: $el_e \geq m$. The equality condition can happen as vertex labels are not unique. The expansion takes place following the constraint. We do not have to show anything as there is no elimination.

Case 2: $el_e < m$. Here S is not expanded to S'. Note that $el_i \geq el_e$ for any el_i in VL. Here we have to show that the substructure will be generated in the same

iteration from another substructure in the presence of the $minEL$ constraint. Such a substructure S1 has to exist due to completeness during the previous iteration. This substructure S1 has the following properties: (i) It contains the edge $< e_j, e >$ and (ii) It is still connected but missing an edge $< e_{j1}, e_{j2} >$ from S where $1 \leq j, j1, j2 \leq n$. The $minEL$ of $S1$ is el_e. Hence $S1$ is expanded to S in the same iteration as either $el_{j1} > el_e$ or $el_{j2} > el_v$. So S' is not generated from S but from $S1$ proving the completeness of the $minVL$ constraint for the general case.

Following a pruning technique, we need to show that the substructure kept in an iteration i and expanding using our heuristics are bound to generate all substructures abiding by the same pruning technique in iteration $i+1$. With MRN metric following the anti-monotone property and *user defined threshold* (τ) abiding by the apriori property (as in Section 4.2.3), and each heuristic being complete, a combination of them is correct and complete.

In Section 4.2.3, we observe that MRN ranking metric with $minSup$ pruning guarantees correctness for exhaustive expansions. We hypothesize that our proposed heuristics exhibit completeness under those same conditions.

We have two observations in this regard:

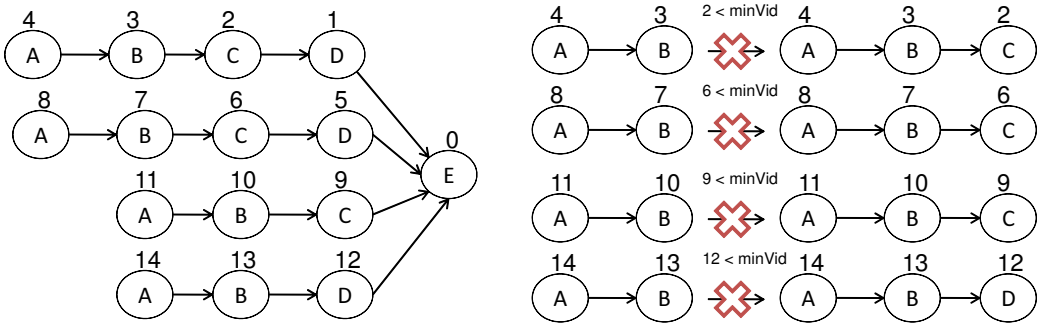
- Heuristics proposed in this paper are not complete in each iteration with respect to substructure generation if *top-k* is used for pruning (*top-k* based on MRN)
- Proposed heuristics are complete in each iteration with respect to substructure generation if $minSup$ property is used for pruning

We theoretically establish these two conjectures using one of our heuristics ($minVid$.)

Theorem 4.4.4 $minVid$ heuristic is not complete in each iteration with respect to substructure generation using $top - k$ pruning property

Proof by Contraction: Assume that top-k heuristic is complete for an iteration n. We just need to show a counter example that does not guarantee completeness

for a value of n and k. Without loss of generality, let us assume n=1 and k=1. Consider the graph in Figure 4.9. Table 4.7 shows the top-1 1-edge substructure with all their instances ranked using MRN. Since k=1, the substructure A-B and all its instances are considered as the only candidates for expansion. When these instances are expanded using $minVid$ constraint, as seen in Figure 4.9b none of them are expanded as in all the cases, the added vertex ids are less than the $minVid$ of the instance. Since none of the candidates were expanded, the top-1 2-edge substructures (which should have been discovered using unconstrained expansion) are now never discovered. Therefore, keeping only the top-1 substructures in an iteration failed to generate the top-1 substructures in the next iteration. Hence, top-k heuristic is incorrect.



(a) Example graph (b) Unsuccessful expansion with top-1 instances

Figure 4.9: Correctness not preserved in presence of top-k property

Theorem 4.4.5 $minVid$ heuristic satisfies completeness in each iteration using MRN ranking and $minSup$ pruning property

Proof: Assume we have a $minSup$ of τ. We need to show that, in an iteration, say k, if a k-edge substructure has a MRN frequency of at least τ, all of its subsets with

Table 4.7: Substructures and instances ranked based on overlap-cognizant frequency

Substructure	Instances	MRN Freq.	Rank
A-B	$[4-3, 8-7, 11-10, 14-13]$	4	1
B-C	$[3-2, 7-6, 10-9]$	3	2
C-D	$[2-1, 6-5]$	2	3
B-D	$[13-12]$	1	4
D-E	$[1-0, 5-0, 12-0]$	1	4
C-E	$[9-0]$	1	4

a MRN frequency of at least τ have been persisted in the previous iteration using the *minSup* pruning property. We prove this using induction.

Base Case : Assume there is a 2 edge substructure $a-b-c$ with a MRN frequency $\geq \tau$. Let the instances of this substructure be represented by F $= \{f_1, ..., f_j\}$ where $j \geq \tau$. Each f_i for $i\epsilon(1, j)$ has two subgraphs $(a-b)_i$ and $(b-c)_i$. Since MRN is overlap cognizant, if $MRN(a-b-c) \geq \tau$, $MRN(a-b) \geq \tau$ and $MRN(b-c) \geq \tau$ and they must have been discovered in the first iteration with $minSup = \tau$

Induction case: Similarly, if a n-edge substructure has τ non-overlapping occurrences, then all of its (n-1)-edge subsets are non overlapping too and have a frequency greater than or equal to τ. With the completeness assumption, all these subsets have been discovered in the previous iteration using $minSup = \tau$ thus proving correctness. To explain with an example, see in Figure 4.9, if $\tau = 3$, both the substructures in the top two rows of Table 4.7 are kept in the first iteration. Even though following $minVid$ constraint, the $A-B$ had unsuccessful expansions, $B-C$ expanded correctly thereby generating 3 independent copies of $A-B-C$.

4.5 Experimental Analysis

In this section we experimentally establish the significance of our heuristic-based expansion techniques. Following our discussion in Section 4.3.2, our experiments portray the performance of using our heuristics on graphs with varying degree distributions, label distributions and also on partitioned graphs.

Following our discussion in Section 4.3.2, our experiments shall validate the following:

- Range of duplicate reduction percentages using our heuristics on graphs with varying degree and edge density distribution
- Comparison of the three heuristics on graphs with varying label distributions leading to the choice of heuristic for an arbitrary graph or a graph with known characteristics
- Validating scalability of using the heuristics on large graphs using an existing partitioning strategy and its effect on speedup.

4.5.1 Experimental Setup

As the number of duplicates depend on graph characteristics, we validate our approach using graphs where we have control over the graph characteristics: degree and label distribution. For this, we have used two graph generators: Subgen [4] and RMAT [5] to generate graphs to meet our size and other requirements. We also use real-life LiveJournal graph for the effect of heuristics. All our experiments are conducted on a single machine with 6GB of RAM and using a Core i5 1.70GhZ processor. To show the effects of our heuristics on partitioned approaches we use a Hadoop cluster with 4 nodes. Each node has a 3.2 GHz Intel Xeon CPU, 4 GB of RAM and 1.5 TB of local disk. Each node was running Hadoop version 1.0.3 on

ROCKS Cluster operating system and connected by gigabit Ethernet to a commodity switch.

Subgen Graph Generator: Subgen allows generation of directed graphs with user-defined number of vertices and edges. The vertices and edges are labeled uniformly at random. For analyzing the performance of heuristics on graphs of varying average degrees, we generated two graphs both with $100,000$ vertices but $200,000$ and $400,000$edges.

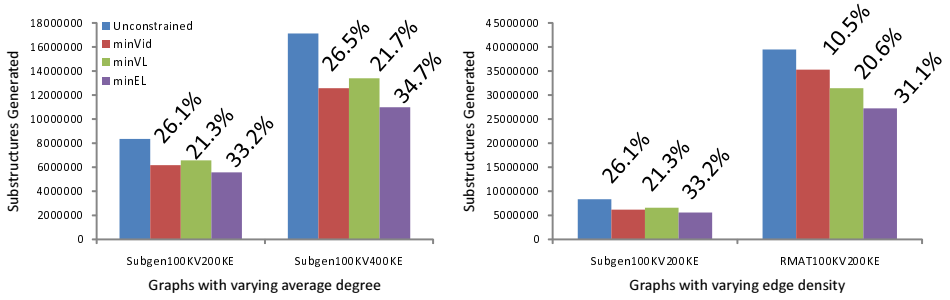
RMAT Graph Generator: RMAT mimics social network graphs using recursive matrix model by specifying vertices and edges. A square matrix of vertices is divided into 4 quadrants recursively where we specify the probability of an edge appearing in each quadrant. We specified the probabilities in the 4 quadrants as 0.57, 0.19, 0.19 and 0.05 to generate a graph with $100,000$ vertices and $200,000$ edges. It has the same average degree as the same-sized Subgen graph but the average edge density of this graph is 150.93 as compared to 8.03 for Subgen.

Real World Graph: We use the LiveJournal graph [2] to examine the performance of the heuristics on a real world graph. This graph has $4.03M$ nodes and $34.68M$ edges. We insert 100 unique vertex labels and 200 unique edge labels into this graph following an uniform distribution. We use a range partition-based approach [26] on this graph along with heuristics to observe the performance of heuristics on partitioned graphs.

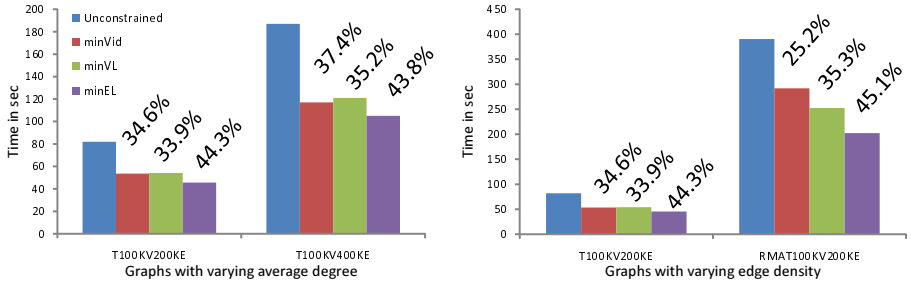
4.5.2 Effect of Degree Distribution

For each of our experiments we use the notion of ***substructures generated*** which is the cumulative of the actual expanded substructures along with the duplicates. As our expansion is complete, the improvement is attributed to reduction of duplicates. Because of the different degree distributions of the graphs, Subgen graph

was run with a τ of 5 while the RMAT graph was executed with a τ of 15. For the Subgen and RMAT graphs, Figure 4.10a shows the percentage improvement of duplicates generated using our heuristics over the unconstrained approach.



(a) Reduction in Duplicates generated using heuristics



(b) Improvement in time using heuristics

Figure 4.10: Improvement of performance using proposed heuristics

The number of substructures generated shows an improvement for all heuristics for two different degree distributions. With increasing average degree across $Subgen100KV200KE$ and $Subgen100KV400KE$, the improvement for all three heuristics remain similar. However between same-sized Subgen and RMAT graphs, $minVid$ attributed to only 10.5% improvement in RMAT as opposed to 26.1% improvement in Subgen graphs.

Subgen has uniform edge density distribution unlike RMAT which has a non-uniform distribution based on the probabilities used during graph generation (as

explained in Section 4.5.1). The highest probability in the first quadrant allowed smaller vertex ids to have much higher connectivity than larger vertex ids. The dramatic increase in the number of substructures generated is also due to the increase in higher connectivity. These contrasting degree distributions (as discussed in Section 4.3.2), resulted in the variation in performance of $minVid$ constraint across Subgen and RMAT graphs. As expected, the performance of label-based heuristics remain unaffected.

The trend of improvement in substructures generated for both graphs is equally reflected in the savings in time as in Figure 4.10b. Improvements of time are from three major causes: 1. Preventing generation of some duplicates 2. Lesser computation cost in generating canonical substructures for the total substructures generated 3. Less duplicates to be pruned.

4.5.3 Effect of Label Distribution

To compare performance of heuristics with varying label distributions we can pick *any* graph and control the label distributions in it. we took the $Subgen100KV200KE$ graph and embedded 10 vertex labels and 20 edge labels randomly following uniform, Gaussian and Poisson distribution. Figure 4.11 shows the average performance of the heuristics over 10 runs of random label generation for each distribution for 5 iterations. For each of these distributions, our heuristics always show an improvement over the unconstrained expansion technique. This helps us experimentally establish our earlier conjecture that our heuristics always provide an improvement on graphs with any degree distribution and label distribution.

In an effort to critically analyze the effect of the heuristics with respect to the graph label distribution, we established more discipline on the Subgen graph by controlling the edge and vertex label distribution. After ordering the edge labels in

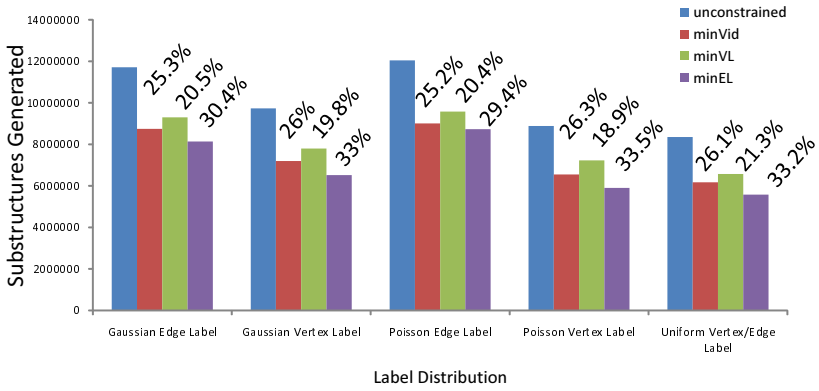


Figure 4.11: Comparison of heuristics with different label distributions

canonical sequence, we divide the edge labels into 10 groups denoted as EG_i where $1 \leq i \leq 10$. we made a group to cover 50% of the graph edges while the rest of the edges are evenly distributed among the other groups thereby creating an edge label distribution. An example edge label distribution, ED_5, with EG_5 group being the most frequent is shown in Figure 4.12. We create 10 such distributions denoted ED_i where $1 \leq i \leq 10$. We create similar distributions for vertex labels denoted VD_i.

Figure 4.13 shows the relative comparisons of $minEL$ and $maxEL$ for 10 different edge label distributions for 5 iterations of our algorithm. While moving

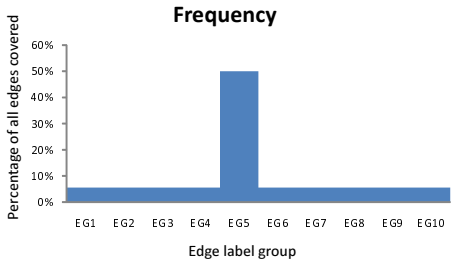


Figure 4.12: Edge label distribution ED5 for EG5 label group

from $ED1$ to the other end of the spectrum we see an improvement in using $minEL$ constraint. $ED1$ has smaller edge labels disproportionately more in number than

lexicographically larger edge labels. This allows $minEL$ more expansions giving worse performance than $maxEL$. As discussed in Section 4.3.2 as we move to the other end, the distributions are reversed with $minEL$ being the better performer. Vertex labels show a similar performance with min and max versions of vertex label based heuristics. As the max versions mirror the min versions of the heuristics, we show performance of heuristics on the min version of the heuristic.

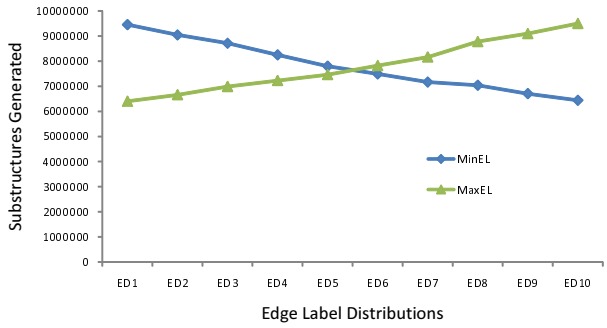


Figure 4.13: Comparison of min and max versions of heuristics on Subgen100KV200KE with varying edge label distributions

To compare relative performance of heuristics, we run 5 iterations of our algorithm with each heuristic on each of the 10 different edge label distributions. Although $minEL$ has more advantages in handling cycles and loops and the least overhead across all heuristics, see in Figure 4.14 that $minEL$ performs not as good as the other heuristics for $ED1$ and $ED2$. As the distribution varies, the larger edge labels become more frequent and $minEL$ starts having a better performance. For example, in $ED10$ the largest decrease in the number of duplicates is noted. As $minVid$ and $minVL$ do not consider edge labels, the number of substructures generated for them do not vary with varying distributions.

Figure 4.15 shows the performance of the heuristics on vertex label distributions. The performance of heuristics with varying vertex label distributions show a

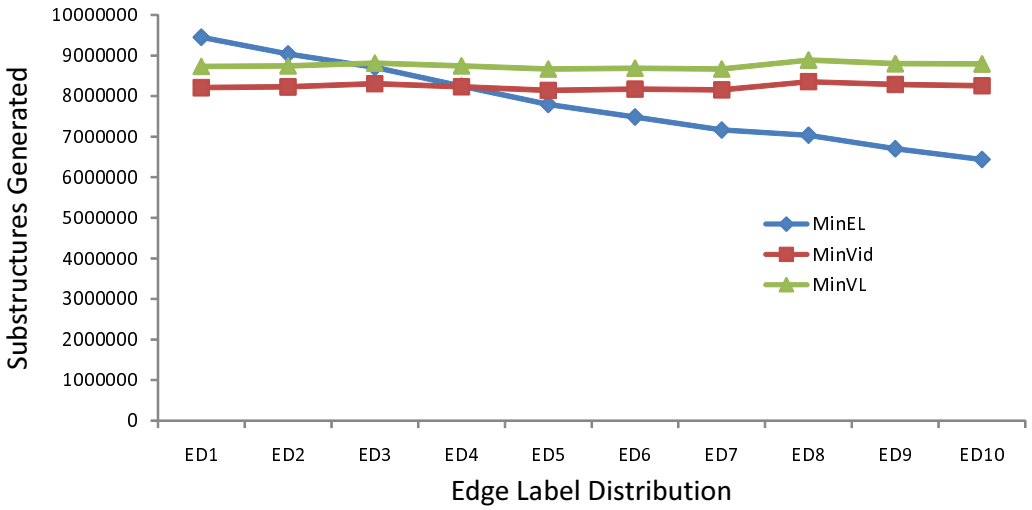


Figure 4.14: Comparison of heuristics with different edge label distributions

similar trend to the performance with edge label distributions. With lexicographically smaller vertex labels being more in number, the number of comparisons using $minVL$ decreases from $VD1$ to $VD10$. The $minEl$ and $minVid$ heuristics remain unaffected by the change in vertex label distributions.

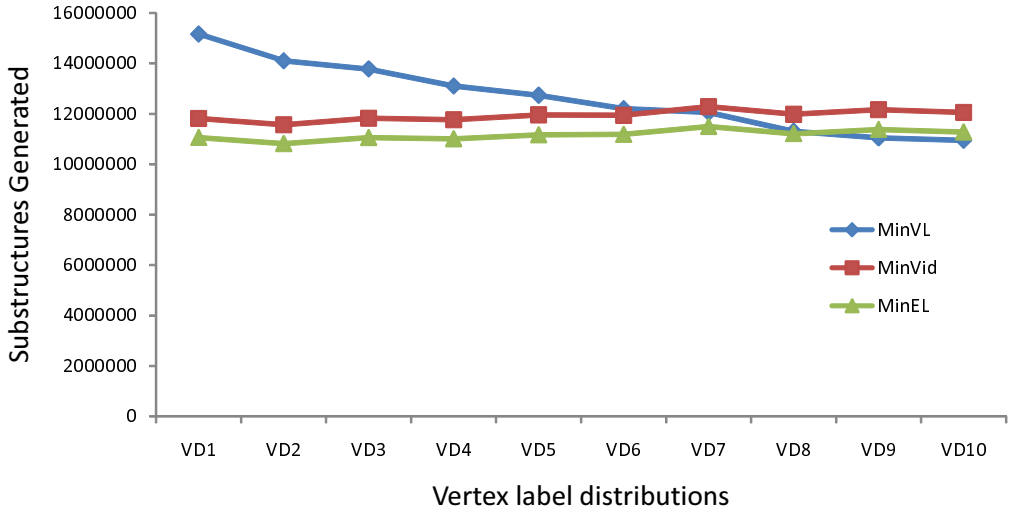
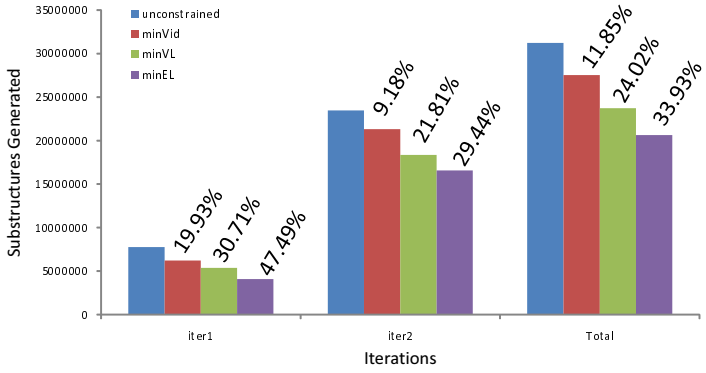


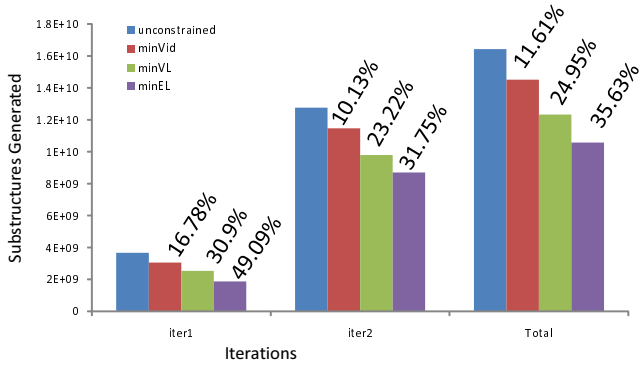
Figure 4.15: Comparison of heuristics with respect to vertex label distribution

4.5.4 Effect of Partitions

Even for a graph with 100,000 vertices and 800,000 edges, the first iteration of the mining algorithm fails on our existing main memory machine. This calls for the usage of partition-based approaches for larger graphs. To show the benefits in a larger graph, we modified a partition-based existing graph mining algorithm in [26] by incorporating the heuristics.



(a) Duplicate Reduction in RMAT graph



(b) duplicate reduction in LiveJournal Graph

Figure 4.16: Comparison of heuristics for two iterations in social network graphs

To verify the effect of our heuristics on large graphs, we use the *liveJournal* graph and divide it into 64 partitions and the τ was set to 350. This graph has 170 ***times more edges*** than the main memory 100KV200KE graph. We compare the

performance on this graph only with the RMAT graph as they both mimic social networks and have non-uniform degree distribution. The goal is to verify the performance of our heuristics over the entire spectrum of graph sizes – small to very large.

The improvements for 2 iterations on $RMAT$ graph are shown in Figure 4.16a followed by the cumulative improvement on the iterations. The improvements across all partitions for first two iterations of *liveJournal* graph, shown in Figure 4.16b, exhibits a similar trend as the smaller $RMAT$ graph. This establishes the fact that our heuristics can effectively be used in a partitioned graph yielding similar performance benefits.

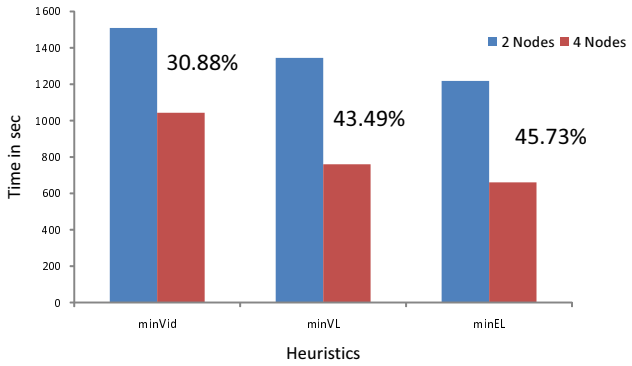


Figure 4.17: Speedup in liveJournal graph

With partitions of graphs, in Figure 4.17, close to linear speedup is obtained while using label based heuristic (45% for $minEL$ and 44% for $minVL$) as opposed to a much lesser speedup for vertex id based heuristic (30%). The reason can be attributed to the partitioning technique used in the partitioned graph mining approach [26]. The speedup for the unconstrained case remains at 43% with all partitions having approximately similar workload.

The partitioning technique in [26] orders graph adjacency list monotonically on vertex ids and then partitions them using ranges of vertex ids. As discussed in Section 4.3.2, this introduces a skew in computation across partitions thereby not producing a close to linear speedup. Other heuristics, unaffected by such a vertex id based partitioning, achieve a close to linear speedup. .tex

4.6 Conclusion

Scalability has been the major concern in graph mining and has been, typically, addressed and overcome by using different approaches and architecture to the same algorithm. Use of partitioning schemes and Map/Reduce have already helped us in achieving scalability [25, 26]. However, it is equally important and useful to identify intrinsic opportunities in each algorithm/approach for significant performance improvements. This enhances the performance of the algorithm universally for all approaches – with or without partitions.

This chapter in this dissertation takes such an approach in identifying the effect of duplicates on the performance of graph mining algorithms. Based on that observation, it proposes a number of heuristics to reduce the number of duplicates generated to significantly improve the performance of these algorithms. Further, we establish its correctness as well as its performance analysis for a number of graph characteristics. Based on these analysis, we show that it is possible to choose the best heuristic whether we have additional information about the graphs or not. Of course, with more information further fine-tuning is possible.

As future work, we plan on investigating the possibility to further improve the performance by changing heuristics at run time based on small amounts of information gathered during each iteration. Also, these algorithms and heuristics need to be analyzed for other paradigms such as Spark [24] and Pregel [31].

.tex

CHAPTER 5

RELATED WORK

This Chapter presents a survey of the work related to the dissertation topics.

5.1 Graph Mining

Systematic substructure generation is central to graph mining. Given a graph (also called a substructure if part of a larger graph), one problem is to find all occurrences of that exact (or even similar) substructure in a larger graph and/or count them. This can be used to find identical or even similar patterns in a large graph to a *known pattern*. Another problem is to find the best substructure that transforms a given graph (or a forest) to satisfy a metric. For example, finding a substructure that minimizes the minimum description length (MDL) or occurs above a certain frequency is important as that substructure demonstrates some interesting property of that graph (e.g., interesting concept in a graph). For both of these problems, it is important to generate substructures of increasing sizes and analyze them in various ways. Hence we focus first on the problem of substructure generation and its analysis using the Map/Reduce paradigm. Research on graph mining in the context of substructure discovery and analysis over the years can be broadly classified as follows.

5.1.1 Main Memory Approaches

Earlier approaches and algorithms for graph mining were developed by AI researchers whose goal was identifying important concepts in a structured data set. These algorithms loaded a complete representation of the graph (either in the form

of an adjacency list or a matrix) into memory for efficiency reasons. Subdue [1] was the first main memory algorithm where substructures were generated iteratively and evaluated using the MDL metric. The substructure discovery algorithm used by SUBDUE is a computationally-constrained beam search. The algorithm begins with the substructure of size one (corresponding to an 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. The evaluation of each substructure is guided by the minimum description length principle and background knowledge rules provided by the user.

The other notable contemporary work on graph mining include either apriori-based approach like AGM [32] and FSG [10] or pattern-growth as in gSpan [11], FFSM [33] and GASTON [34]. These algorithms, however, have some inherent limitations: handling large patterns, huge candidate set generation and multiple scans of the database.

In an *Apriori-based* approach search for repetitive substructures starts with graphs of small sizes and proceeds in a bottom-up approach. At each iteration existing substructures are expanded by joining two similar but slightly different substructures (For e.g., a 3-edge substructure is discovered by joining two 2-edge substructures which differ by 1 edge)). Apriori Graph Mining (AGM) took advantage of apriori property introduced in [32]. Frequent substructure Discovery (FSG) [10], also an apriori-based approach, aimed at discovering substructure(s) that occurred frequently over the set of given graphs (graph database). FSG used the concept of canonical labeling for graphs and worked with the fact that two exactly similar graphs should have the same canonical labeling. A straightforward way of determining a canonical

label is to use a flattened representation of the adjacency matrix of a graph. By concatenating rows or columns of an adjacency matrix one after another we can construct a list of integers. By regarding this list of integers as a string, we can obtain total order of graphs by lexicographic ordering. To compute a canonical label of a graph, we have to try all the permutations of its vertices to see which order of vertices gives the minimum adjacency matrix. To narrow down the search space, they first partition the vertices by their degrees and labels, which is a well-known technique called vertex invariant. Then, they try all the possible permutations of vertices inside each partition. These algorithms however have some inherent limitations: handling large patterns, huge candidate set generation and multiple scans of the database.

In the *Pattern-growth* approach (as in gSpan [11]) a frequent pattern is expanded directly by adding a node or edge in all possible combinations. A potential problem with such an approach is that the same structure will be discovered multiple times. A canonical DFS lexicographic order was used which assigned a DFS code to each graph. Based on the codes, a hierarchical tree was constructed and a pre-order traversal of the tree gave all the frequent substructures with the required support. DFS solved the problem of duplicate generation by expanding only on the vertices in rightmost path (the path from root to the rightmost node of the hierarchical tree). Though DFS is better than Apriori, the major problem of DFS was generating many DFS trees for the same graph as different start nodes will result in different trees for the same graph. Several other methods like FFSM [33] and GASTON [34] followed similar expansion strategies like gSpan and had similar disadvantages.

5.1.2 Disk-Based Approaches

As larger applications with structural information became common place, disk-based graph mining techniques [13–15] were developed to overcome the problem of

storing the entire data set in main memory. A section of data was kept in memory and the rest was on the disk. Since random accesses to disk-based graphs were difficult, a natural choice was to index the graph databases. Designing effective and efficient index structures therefore is one of the most invaluable exercises in database research. As a big graph can be indexed by the frequent substructures present in it, as in gIndex [35] and GBLENDER [36], mining frequent patterns for large graphs became possible.

5.1.3 Database-oriented Approaches

Disk-based algorithms solve the problem of keeping portions of the graph in memory for processing. However, these algorithms need to marshal data between external storage and main memory buffer and this 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 is to make use of efficient buffer management and query optimization – already mature in the context of a Database Management System (DBMS) – by mapping these graph mining algorithms to SQL [16,17] and storing data in a DBMS. Many a times, the data is already in a DBMS as enterprises use DBMS for storing their corporate data.

Although scalability was achieved to graphs with over a million each of nodes and edges, use of joins for substructure expansion generated duplicate substructures, the removal of which required sorting columns (in row based DBMSs) making it expensive. Therefore duplicate elimination is an important task for these algorithms. As this required sorting columns (whereas DBMSs are efficient in sorting rows), several joins had to be used for sorting columns making it expensive. As operations in DBMSs were limited by the number of columns in a relation, there was an up-

per bound on largest substructure that can be represented using database-oriented approaches.

5.1.4 Recent Trends in Graph Mining

Some of the graph mining [37–40] algorithms have been shown to be effective in a cloud architecture. Frequent substructure enumeration [38] improves upon the best known algorithm for substructure enumeration but focuses mostly on improving the communication cost by exploiting the techniques of Ullman and Arafati [41]. There also have been efforts on pattern finding in large graphs using Map/Reduce [39]. However this pattern finding algorithm [39] requires a pattern to be given to search for all instances of *that* pattern in the graph. In contrast we are looking for a pattern that possesses certain properties (e.g. best compressibility) and the pattern is not known apriori. Approaches to partitioning big graphs for distributed processing is also an area of active research [42]. Our perspective on the proposed problem is novel and is quite different from the work in the literature. Most of the work in the literature is looking at specific subproblems (e.g. analyzing structural property, page rank, diameter, connected components etc. of graphs) whereas our intent is to develop scalable mining of substructures to count or apply any metric that is suitable for evaluating the substructures [25, 26].

5.2 Optimizations For Mining

Recently, there have been several work in the area of scalable mining. Frequent subgraph mining has been done using map/reduce or MPI [43–46]. All of these algorithms work on a database of graphs and are interested in finding a single occurrence of a pattern in each graph. Subgraph mining has also been tackled as join-based approach in a distributed framework along with some theoretical analysis in [37–40].

Substructure discovery in a large single graph using map/reduce has also received attention in [25, 26]. Recently, Grami [8] proposes an approach that finds a subset of interesting subgraphs by modeling frequency evaluation as a constraint satisfaction problem.

In all these existing mining techniques, subgraphs are expanded in an unconstrained manner by adding all possible nodes/edges. Though expansion is constrained in querying for known nodes/edges, presence of an unknown node/edge requires all possible expansions. The unconstrained expansion technique used above generates large number of duplicates during expansion requiring additional storage and computation to eliminate them.

Our approach is different from these existing techniques in terms of the expansion technique. Even in the absence of any apriori information or graph characteristics, our heuristics significantly reduce duplicates during substructure expansion. Our approach offers benefits at several stages of the mining algorithm over existing techniques. First, we reduce the number of duplicates generated at expansion time thereby reducing the amount of intermediate results. Second, we save duplicate removal cost which can be of quadratic complexity in the worst case. Third, storage cost of intermediate results is reduced due to non-generation of duplicates. Finally, our approach works for both partitioned and non-partitioned mining without any modification. We have also established correctness and soundness of these heuristics along with experimental validation of performance improvements. We also believe that the proposed duplicate reduction strategy can be easily adapted in graph querying in addition to mining.

5.3 Graph Partitioning for Mining

Graph partitioning has been predominantly used in the process of query answering on graphs. The goal is to divide the graph vertices (and their associated adjacency list) into roughly equal sized partitions. The edges connecting partitions form the edge cut and the aim is to minimize the total number of edge cuts for the best partitioning scheme. METIS [23] was one of the earliest and still the best in market graph partitioning scheme. It coarsens the graph in stages to hundreds of vertices and edges, runs a main memory recursive bisection or k-way partitioning technique and projects it back to the original size. Although METIS provides high quality partitions, it consumes a fair amount of time in the coarsening and un-coarsening phases of the graph. METIS however is not suited for temporal graphs which allows addition and deletion of vertices and edges with time.

As an improvement over METIS, Sedge [42] analyzed graph partitioning from a query workload perspective. The initial partitions provided by METIS may not be a good fit if most of the queries in the workload of queries cross partitions or hit the edge cuts. Sedge solves this problem by repartitioning the graph to introduce complementary partitions. The goal is to keep frequently answered results inside a partition to facilitate faster query answering for the next queries. The underlying assumption is that multiple users typically end up asking popular or similar queries and keeping their results inside one partition achieves least number of partition loading. However from a graph mining perspective, the user/system has no apriori knowledge of the patterns in the graph. This calls for an investigation of applicability of these partitioning techniques for a graph mining perspective.

Recently, graph mining using partitions has also received some attention. Frequent subgraph mining has been done using Map/Reduce or MPI in [43–46]. But all of these approaches focus on mining patterns on graph databases (a large number

of small graphs) where multiple appearances of a pattern in a single graph are not evaluated. Partitioninng a graph database is very different and much easier than partitioning a very large graph. Our technique focuses on a large single graph (although applicable to a forest), considers multiple occurrences and can also be used for a database of graphs making this approach a general purpose one.

Partitioned approaches have also been used to find all occurrences of a given substructure (query) in a large graph (or in a database of graphs.) Map/Reduce has also been rigorously used to query [47–49] graphs. All of these techniques start with a given query while our substructure discovery starts with no *a priori* information making substructure discovery on partitions different from other Map/Reduce based efforts on graphs. Additionally we analyze component costs and choice for partitioning making our work different from others.

Although other paradigms (like Giraph [50]) have been used extensively for graph mining, the Pregel architecture is not suited for applications where substructure structure is passed from one iteration to the next. Our goal in this paper is to explore alternative approaches to substructure generation, exact match of substructures, and counting using the Map/Reduce paradigm. As this forms the basis for many other problems (e.g., frequency counting, graph query processing), we believe that this problem needs to be explored first to understand its nuances for developing a scalable algorithm using the Map/Reduceparadigm.

To summarize, although there is a body of work on graph mining and substructure discovery using different approaches and architectures, the problems addressed in this thesis focus on all aspects of a specifica problem which has not been addressed in the literature. Our approach is holistic in that we have developed new algorithms for a chosen framework, analyzed their scalability using paertitioned schemes, proposed

new partitioned schemes, we have also analyzed component cost for the framework and proposed optimizations to improve overall performance.

CHAPTER 6

CONCLUSION

6.1 Summary of Contributions

This dissertation takes a holistic approach to scalable substructure discovery by proposing divide and conquer strategy over graph partitions. Specifically, a set of related problems were identified and addressed as a part of this research.

Partitioned Substructure Discovery: First, this dissertation introduced generic Map/Reduce based algorithms for horizontal scalability of substructure discovery that can work with any partitioning strategy. The basic components of graph mining - subgraph expansion, duplicate removal and counting of isomorphic substructures were incorporated into the algorithms for the Map/Reduce paradigm by carefully orchestrating new representations. Experiments validated the advantage of using Map/Reduce based substructure discovery to scale to large graphs over traditional methods.

Analysis of Partitioning Schemes: The dissertation then continued on to evaluating the suitability of existing partitioning strategy for substructure discovery. A state-of-the-art graph partitioning technique (METIS) was analyzed from a graph mining perspective and its limitations were addressed by introducing two new partitioning schemes (arbitrary and range-based.)

Component Cost Analysis in Distributed Framework: In an effort to analyze the partitioning strategies and associated algorithms from a performance standpoint, the dissertation analyzed the component cost analysis of substructure discovery in a distributed framework. The cost analysis identified places for improvements in using

the range-based partitioning strategy over its counterpart. Theoretical justification along with experimental evaluation of the improvements were verified by varying a number of user parameters. The cost analysis also pointed out the portability of our algorithms to a different paradigm such as Spark to reap similar benefits.

Optimizations for Mining: Use of partitioning schemes and Map/Reduce have already helped us in achieving scalability. Further, as part of the holistic approach, we have looked at intrinsic opportunities in each algorithm/approach to improve performance significantly. This resulted in enhancing the performance of the algorithms universally for all approaches - with or without partitions. Effect of duplicates on the performance of graph mining algorithms was one such opportunity. Based on that observation, we proposed a number of heuristics to reduce the number of duplicates generated to significantly improve the performance of these algorithms. Theoretical correctness was again established along with performance analysis. Effects of graph characteristics have been studied extensively to infer the use of heuristics based on graph characteristics. Based on these analyses, this dissertation infers ways to choose the best heuristic when we have additional information about the graphs.

In summary, this dissertation takes a holistic approach to scaling substructure discovery using a distributed paradigm. It specifically developed a generic partition-based technique for substructure discovery, identified better partitioning strategies, provided detailed component cost analysis along with optimizations for improving substructure discovery on partitions. Detailed evaluation over synthetic data sets and real world graphs were performed to validate the effectiveness of the techniques presented and to corroborate theoretical analysis.

6.2 Future Work

Partitioned graph mining is a promising research field and offers several exciting future directions. We present some of the future work that builds on the work presented in this dissertation.

(i) ***Partitioned Graph Querying:*** When a graph (also referred to as knowledge base) is accessed by multiple users, at any instance the system has to deal with a batch or workload of queries. Although query processing for a single query on a partitioned graph is well known, solving a batch of queries on a partitioned graph faces several challenges. First, answering a query will require either loading multiple partitions or route intermediate search results across partitions incurring shuffle and loading cost. This calls for further investigation on partitioning techniques for query answering. Our initial analysis [51] used heuristics gleaned from the graph to reduce number of partition loads. This enables us to focus more attention to using a distributed paradigm (Map/Reduce or Spark) for query answering. Moreover, answering one query at a time may not be appropriate as users expect a quick query answering time. One solution is to use query characteristics to group queries for simultaneous answering on partitions. We believe that an ideal distributed query answering system should use a partitioning scheme that uses heuristics garnered from both the graph as well as the queries.

(ii) ***Mining based on Similar Substructures:*** This dissertation has focused on techniques to identify interesting patterns in a large graph. In the process of comparing patterns the notion of isomorphism (or exact match) was used. As of today most of the graphs are created by crowd sourcing (or user adding data to increase the knowledge base), there are chances that some information might be lost in process. As a result approximate mining is necessary to

group patterns which are similar to each other bounded by a threshold. This is similar to the edit distance principle in graphs. This calls for investigation of techniques for graph similarity - both structural and semantic. There are a few measures for approximate query answering in graphs like percentage of missing neighborhood [52] and AGraMi [53] where users can specify similarity such as missing labels, missing edges etc. Definitely approximate mining shall increase the computation costs as more alternatives need to be compared and graph comparison in itself is a computationally expensive task. The challenge is to explore the limitations of the current partitioning strategies in the light of approximate mining. A strategy for that shall be to use graph characteristics which we already extract for our catalog generator [54] and see if a combination of these characteristics can be further used for partitioning.

The ultimate goal in the long run shall be to develop a suite of graph algorithms that are theoretically correct (work on any partitioning schemes) and optimize them further to improve scalability. With the size of data increasing everyday, this dissertation is a step in the direction of scaling graph algorithms (especially substructure discovery) in the context of big data analytics.

REFERENCES

[1] L. B. Holder, D. J. Cook, and S. Djoko, "Substucture Discovery in the SUBDUE System," in *Knowledge Discovery and Data Mining*, 1994, pp. 169–180.

[2] http://snap.stanford.edu/data/com-LiveJournal.html.

[3] http://snap.stanford.edu/data/com-Orkut.html.

[4] http://ailab.wsu.edu/subdue.

[5] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-MAT: A recursive model for graph mining," in *SIAM, Florida, USA, April 22-24, 2004*, 2004, pp. 442–446.

[6] http://www.internetworldstats.com/facebook.htm.

[7] http://blog.linkedin.com/2013/01/09/linkedin-200-million.

[8] M. Elseidy, E. Abdelhamid, S. Skiadopoulos, and P. Kalnis, "GRAMI: frequent subgraph and pattern mining in a single large graph," *PVLDB*, vol. 7, no. 7, pp. 517–528, 2014. [Online]. Available: http://www.vldb.org/pvldb/vol7/p517-elseidy.pdf

[9] A. Inokuchi, T. Washio, and H. Motoda, "An Apriori-Based Algorithm for Mining Frequent Substructures from Graph Data," in *Principles of Data Mining and Knowledge Discovery*, 2000, pp. 13–23.

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

[11] X. Yan and J. Han, "gSpan: Graph-Based Substructure Pattern Mining," in *IEEE International Conference on Data Mining*, 2002, pp. 721–724.

[12] M. Kuramochi and G. Karypis, "Frequent Subgraph Discovery," in *IEEE International Conference on Data Mining*, 2001, pp. 313–320.

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

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

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

[16] S. Padmanabhan and S. Chakravarthy, "HDB-Subdue: A Scalable Approach to Graph Mining," in *DaWaK*, 2009, pp. 325–338.

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

[18] W. Jiang, J. Vaidya, Z. Balaporia, C. Clifton, and B. Banich, "Knowledge discovery from transportation network data," in *ICDE 2005*, April 2005, pp. 1061–1072.

[19] R. Rathi, D. J. Cook, and L. B. Holder, "A serial partitioning approach to scaling graph-based knowledge discovery." in *FLAIRS Conference*, I. Russell and Z. Markov, Eds. AAAI Press, 2005, pp. 188–193.

[20] J. Dean and S. Ghemawat, "MapReduce: Simplied Data Processing on Large Clusters," in *Operating Systems Design and Implementation*, 2004, pp. 137–150.

[21] http://hadoop.apache.org.

[22] J. Lin and M. Schatz, "Design patterns for efficient graph algorithms in MapReduce," in *16th ACM SIGKDD Conference*, 2010, pp. 78–85.

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

[24] M. Zaharia, M. Chowdhury, M. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," ser. HotCloud, 2010.

[25] S. Das and S. Chakravarthy, "Challenges and approaches for large graph analysis using map/reduce paradigm," in *BDA*, 2013, pp. 116–132.

[26] ——, "Partition and conquer: Map/reduce way of substructure discovery," in *DaWaK 2015, Valencia, Spain, September 1-4, 2015*, 2015, pp. 365–378.

[27] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, ser. HotCloud'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 10–10. [Online]. Available: http://dl.acm.org/citation.cfm?id=1863103.1863113

[28] B. Bringmann and S. Nijssen, "What is frequent in a single graph?" in *Advances in Knowledge Discovery and Data Mining, 12th Pacific-Asia Conference, PAKDD 2008, Osaka, Japan, May 20-23, 2008 Proceedings*, 2008, pp. 858–863. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-68125-0_84

[29] T. Washio and H. Motoda, "State of the art of graph-based data mining," *SIGKDD Explorations*, vol. 5, no. 1, pp. 59–68, 2003. [Online]. Available: http://doi.acm.org/10.1145/959242.959249

[30] A. Buluç, H. Meyerhenke, I. Safro, P. Sanders, and C. Schulz, "Recent advances in graph partitioning," in *Algorithm Engineering*. Springer, 2016, pp. 117–158.

[31] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM, 2010, pp. 135–146.

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

[33] J. Huan, W. Wang, and J. Prins, "Efficient Mining of Frequent Subgraphs in the Presence of Isomorphism," ser. ICDM '03, Washington, DC, USA, 2003, pp. 549–552.

[34] S. Nijssen and J. N. Kok, "A quickstart in frequent structure mining can make a difference," in *Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining*, ser. KDD '04. New York, NY, USA: ACM, 2004, pp. 647–652.

[35] X. Yan, P. S. Yu, and J. Han, "Graph Indexing: A Frequent Structure-based Approach," in *SIGMOD Conference*, 2004, pp. 335–346.

[36] C. Jin, S. S. Bhowmick, X. Xiao, J. Cheng, and B. Choi, "GBLENDER: towards blending visual query formulation and query processing in graph databases," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, ser. SIGMOD '10. New York, NY, USA: ACM, 2010, pp. 111–122.

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

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

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

[40] M. M. Alam, M. Khan, and M. V. Marathe, "Distributed-memory parallel algorithms for generating massive scale-free networks using preferential attachment model," in *SC*, 2013, p. 91.

[41] F. N. Afrati, V. R. Borkar, M. J. Carey, N. Polyzotis, and J. D. Ullman, "Map-reduce extensions and recursive queries," in *Extending Database Technology*, 2011, pp. 1–8.

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

[43] G. D. Fatta and M. R. Berthold, "Dynamic load balancing for the distributed mining of molecular structures," vol. 17, no. 8, 2006, pp. 773–785.

[44] S. Hill, B. Srichandan, and R. Sunderraman, "An iterative mapreduce approach to frequent subgraph mining in biological datasets," in *Proceedings of the ACM Conference on Bioinformatics, Computational Biology and Biomedicine*, ser. BCB '12, 2012, pp. 661–666.

[45] W. Lin, X. Xiao, and G. Ghinita, "Large-scale frequent subgraph mining in mapreduce," in *IEEE 30th International Conference on Data Engineering, Chicago, ICDE 2014, IL, USA, March 31 - April 4, 2014*, 2014, pp. 844–855.

[46] W. Lu, G. Chen, A. K. H. Tung, and F. Zhao, "Efficiently extracting frequent subgraphs using mapreduce," in *Proceedings of the 2013 IEEE International Conference on Big Data, 6-9 October 2013, Santa Clara, CA, USA*, pp. 639–647.

[47] X. Ren and J. Wang, "Multi-query optimization for subgraph isomorphism search," vol. 10, no. 3, 2016, pp. 121–132.

[48] F. Katsarou, N. Ntarmos, and P. Triantafillou, "Performance and scalability of indexed subgraph query processing methods," vol. 8, no. 12, 2015, pp. 1566–1577.

[49] L. Lai, L. Qin, X. Lin, Y. Zhang, and L. Chang, "Scalable distributed subgraph enumeration," vol. 10, no. 3, 2016, pp. 217–228.

[50] http://giraph.apache.org/.

[51] J. Bodra, "Processing Queries over graph databases, An approach and its evaluation," Master's thesis, The University of Texas at Arlington, May 2016.

[52] Y. Tian and J. M. Patel, "Tale: A tool for approximate large graph matching," in *Proceedings of the 2008 IEEE 24th International Conference on Data Engineering*, ser. ICDE '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 963–972. [Online]. Available: http://dx.doi.org/10.1109/ICDE.2008.4497505

[53] M. Elseidy, E. Abdelhamid, S. Skiadopoulos, and P. Kalnis, "GRAMI: frequent subgraph and pattern mining in a single large graph," *PVLDB*, vol. 7, no. 7, pp. 517–528, 2014. [Online]. Available: http://www.vldb.org/pvldb/vol7/p517-elseidy.pdf

[54] S. Das, A. Goyal, and S. Chakravarthy, "Plan before you execute: A cost-based query optimizer for attributed graph databases," in *DaWaK 2016, Porto, Portugal, September 6-8, 2016*, 2016, pp. 314–328.

BIOGRAPHICAL STATEMENT

Soumyava Das was born in Kolkata, India in 1987. He received his Bachelor of Engineering degree in Computer Science and Engineering from Jadavpur University, India in July 2009. He worked as a research intern in the computer vision and pattern recognition lab at Indian Statistical Institute Kolkata, India in summer 2008. After graduation, he worked as a Software Developer for Atrenta, India (now Synopsys) from 2009 to 2011 before starting his doctoral research at the University of Texas at Arlington in September 2011. He has served as a Graduate Teaching Assistant and an Assistant Instructor in the Department of Computer Science and Engineering at The University of Texas at Arlington from 2011 till 2017. He is the recipient of the Enhanced GTA Fellowship from 2011 to 2016 and the STEM fellowship from 2016 to 2017. During his doctoral research, he visited StumbleUpon Inc., San Francisco, California in summer 2015 as a data science intern and also worked as a research collaborator with a San Francisco based startup, Promote in Fall 2016. He has served as an external reviewer for DASFAA, DEXA, DAWAK and ER conferences and also serves in the program committee of the International Conference on Big Data Analytics (BDA) conference. Following the completion of his Ph.D. Soumyava will start working at Teradata Aster Labs, Santa Clara, USA as an Analytics Software Engineer.