Journal of Theoretical and Applied Information Technology
15th March 2025. Vol.103. No.5

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

OPTIMIZED FREQUENT SUBGRAPH MINING USING ITERATIVE MAPREDUCE FOR ENHANCED SCALABILITY AND PERFORMANCE

[1] NAGA MALLIK ATCHA [2]JAGANNADHA RAO D B [3]VIJAYAKUMAR POLEPALLY

[1]Research scholar, Department of CSE, Malla Reddy University, Hyderabad, Telangana, India.
[2]Associate Professor, Malla Reddy University, Hyderabad, Telangana, India.
[3]Associate Professor, Kakatiya Institute of Technology & Science, Warangal, Telangana, India.

Email: [1]mallik.atcha@gmail.com [2]jagandb@gmail.com [3]vijayakumarpolepally@gmail.com

ABSTRACT

Frequent subgraph mining (FSM) is a core graph analysis task arising from many application domains, including bioinformatics, chemoinformatics, and social network analysis. Traditional FSM methods are not scalable to large datasets due to in-memory computations and inefficient candidate pruning, as found in gSpan and Apriori-based techniques. Although some recently distributed approaches, such as G-thinker and FlexMiner, have tried to overcome them, they are still confronted with high computational overhead, excessive data shuffling, and scalability. Such problems necessitate a sound, scalable approach for large-scale graph mining in the modern era. This research proposes a novel, sophisticated framework, and algorithm called Frequent Subgraph Mining Using MapReduce (FSM-MR) with inherent optimizations. This efficient algorithm incorporates mapper combiners for minimizing data shuffling, canonical labeling for avoiding repeated enumeration of identical subgraphs, and dynamic support thresholds for effective pruning. FSM-MR, implemented in a Hadoop environment, shows better performance with up to 50% runtimes shorter than the state-of-the-art methods, with near-linear scalability with the addition of cluster nodes. The ability of the proposed framework to process large-scale graph datasets makes it beneficial for applications involving scalable, efficient graph mining. FSM-MR overcomes methodological limitations in the current state-of-the-art algorithms, helping set the stage for future research in these areas and fostering graph analytics in various scientific and industrial settings.

Keywords - *Frequent Subgraph Mining, MapReduce Framework, Scalability, Distributed Graph Analysis, Big Data Processing*

1. INTRODUCTION

Frequent subgraph mining (FSM) is a critical task in graph-based data analysis, which aims to extract repeated structures in bioinformatics, chemoinformatics, social network analysis, etc. It has been used to identify networks of protein interactions, probe molecular shape and molecular models, and even find social communities. Many conventional FSM methods require in-memory computations and exhaustively generate candidates, limiting the state-of-the-art techniques to small or medium-sized data sets [1, 2]. The emergence of very large-scale data has given rise to distributed frameworks such as G-thinker [3] and pattern-aware systems such as FlexMiner [4] that provide better scalability. These methods, however, struggle with very high matrix shuffling overheads, target inefficient candidate pruning, and computational bottlenecks requiring subgraph isomorphism checks. These gaps in FSM methods highlight the need for an approach that performs at the scale of geometric syntactic features. To tackle these new challenges, in this research, we propose a new algorithm and framework for Frequent Subgraph Mining Using MapReduce (FSM-MR) that utilizes distributed computing environments along with some innovative optimizations. The research objectives covered include scalable FSM algorithm design, subgraph enumeration, pruning optimizations, and quantitative results on framework efficiency. Heap-based in-mapper combiners to reduce data shuffling, canonical labeling for redundancy elimination, and a novel approach for dynamic support thresholds to

effectively prune candidates are crucial novelties of the proposed research.

Frequent Subgraph Mining (FSM) has been widely researched in recent years due to the importance of FSM in Bioinformatics, Chemoinformatics, Social Network Analysis, and so on. Although groundbreaking methods such as gSpan and Apriori-based approaches have provided best practices for enumerating subgraphs, they suffer from excessive computation overhead and show a significant lack of scalability with large datasets. Inheritance-based counter strategies have flourished from G-thinker, FlexMiner, and many more distributed frameworks. However, they still lack parallelization in extracting frequent itemsets which leads to superfluous data shuffling and redundant candidate generation. We present a new MapReduce-based framework for subgraph discovery that combines in-mapper combiners, canonical labeling, and dynamic support thresholding to overcome the above limitations. By minimizing intermediate data transfer, eliminating duplicatecomputations, and performing effective early pruning of infrequent subgraphs, these optimizations close a significant gap in the current literature and present a scalable, state-of-the-art technique for FSM.

We introduce a distributed framework for frequent subgraph mining to alleviate further the computational load associated with classic algorithms for extensive graph data within this work. Our framework (FSM-MR), based on the MapReduce paradigm and enhanced with several novel optimizations – in-mapper combiners, canonical labeling, and dynamic support thresholding – alleviates problems like excessive data shuffling and redundant candidate enumeration and achieves considerable runtimes and near-linear scalability. The applications of our study extend far beyond its narrow scope of static, labeled graph datasets in a Hadoop-based environment; our contributions apply to numerous fields, such as bioinformatics, chemoinformatics, and social network analysis. First, This is not only a step forward in the state-of-the-art scalable graph mining, but also sets a stage for future adaptation in dynamic and heterogeneous computing paradigms.

This research makes three main contributions. It first presents FSM-MR, a new distributed algorithm for the generic FSM problem that tackles FSM's scalability and efficiency issues. Second, it provides a broad design space exploration using synthetic and accurate data to show considerable runtime improvements (up to 3× on actual data) and nearly linear scaling with the state-of-the-art. Finally, our research provides a strong theoretical foundation encouraging efficient applications of large-scale graph mining. The rest of the paper is organized as follows. In Section 2, we perform a literature review, reviewing state-of-the-art non-distributed and distributed FSM methodologies and pointing out the research gaps. Preliminaries in Section 3 provide basic graph definitions, a brief review of Finite State Machine terminology, and an overview of the MapReduce framework. Description of Proposed FSM-MR methodology including algorithmic steps and optimizations — Section 4 Experimental Results: Section 5 has the experimental results, where we present the experimental setup, followed by evaluation metrics, results and discussion, ablation study, and the comparative analysis. Findings (novels, limitations, and implications of the research) are discussed in Section 6. Section 7 draws the paper closer by summarizing our key contributions and outlining possible avenues of future work, including the ability to adapt FSM-MR to run on different distributed platforms and extending our algorithm to the dynamic graph mining setting. This structure guarantees that research is presented systematically and comprehensively while emphasizing the relevance and contributions to the field.

2. RELATED WORK

This section explores advancements in subgraph mining, focusing on algorithms, distributed frameworks, and domain-specific applications. It discusses methodologies for improving efficiency, scalability, and accuracy in mining frequent subgraphs. The review also identifies challenges such as memory constraints, computational complexity, and scalability limitations, highlighting opportunities for further optimization and broader applicability.

2.1 Algorithms for Subgraph Mining

This section focuses on methodologies and frameworks for improving the efficiency of subgraph mining. Nguyen et al. [1] created the CloGraMi algorithm, which uses novel traversal and pruning techniques to improve efficiency in mining closed frequent subgraphs. Additional improvements may be investigated in future development. Yan et al. [2] presented G-thinker,

a CPU-bound framework for practical distributed subgraph mining. Potential scaling problems are one of the limitations. Work in the future could improve algorithm variety. Nguyen et al. [5] enhanced parallel processing, edge sorting, and computational assistance were used to optimize the GraMi method for mining frequent subgraphs. One of the limitations is the amount of memory left. Further performance enhancements may be the main focus of future research. Yan et al. [6] created PrefixFPM, a framework for frequent pattern mining that can be customized to maximize CPU core use. Potential scaling problems are one of the limitations. Further research might improve the algorithm's flexibility for various kinds of data. Jamshidi et al. [7] "Subgraph counting is a fundamental task in graph analysis, focusing on identifying and enumerating specific subgraph patterns within more extensive networks. It has applications in diverse fields, such as bioinformatics, social network analysis, and computational chemistry.[8]created PEREGRINE, a pattern-aware graph mining system that maximizes computation and subgraph exploration. Potential implementation complexity is one of the limitations. Work in the future could improve on user-defined pattern expressions. Song et al. [30] presented filtering techniques and an optimum partial evaluation algorithm to enhance the efficiency of subgraph matching in distributed knowledge networks. Potential bottlenecks in assembly computation are one of the limitations. Future research may concentrate on investigating different graph types and further enhancing scalability.

Preti et al. [18] created new scoring methods for graph pattern mining that increase pruning efficiency while preserving the apriori characteristic. Performance issues with increased weighting functions are among the limitations. Algorithms may be improved in future research for enhanced performance and scalability on various datasets. Pasini et al. [38] created a frequent subgraph mining method for scene graph-based picture summarizing that produced a variety of comprehensible summaries. Pattern finding may be improved by future research. Zhao et al. [17] created Kaleido, a productive out-of-core graph mining system that maximizes memory use and handles intermediate data. Potential performance problems with massive datasets are one of the limitations. Improving scalability and isomorphism verification techniques could be the main focus of future research. Chen et al. [9] created Sandslash, a flexible graph pattern mining framework with efficient low-level and high-level APIs. One of the limitations is that low-level usage may be complicated. Optimization strategies might be further improved by future research.

2.2 Distributed and Scalable Frameworks

The rest of the section covers subgraph mining regarding distributed systems, scalability, and optimization. Chinese research on subgraph mining includes the work of Jazayeri and Yang [3], who explored several subgraph mining methods with a focus on both temporal and static networks. First, memory limitations were not investigated. Research in this area may also be expanded to look at more comprehensive assessments. Rehman et al. A-RAFF: Addressing the familiar challenges of duplication and a combinatorial explosion of patterns common subgraph mining with a framework of FSP-Rank [4]. Limited examples might be user-defined thresholds. Automated support systems could be explored by future research. Bindschaedler et al. Tesseract [10] is a distributed system that improves the performance of graph mining tasks on dynamic graphs using incremental updates. One of those may perhaps be scalability issues. Change detection is still in its infancy, and future research on improving it may lead to even better results. Wang et al. Streaming-BENU and Batch-BENU, the frameworks, represent many techniques for performing distributed enumerations of subgraph enumeration with good performance and scalability ([14]).

One of its limitations is the potential complexity of the implementation. The following research can focus on controlling dynamic graphs and reducing memory footprints. Belhadi et al. Take, for instance, [19], which introduced DT-DPM, a distributed approach enabling a more scalable and restricted form of search space for pattern mining. One of the drawbacks is the potential added complexity in clustering transactions. Future work may focus on performance optimization over data formats and mining methods.

Ayala et al. [39] examined large-scale analytics computing and graph partitioning systems, pointing out problems and suggesting future lines of inquiry to improve scalability. Dahiphale et al. [28] created BiECCA, a distributed MapReduce technique for locating 2-edge linked components in big graphs. Reliance on current

algorithms for linked components is one of its drawbacks. To improve the algorithm's capabilities, future research proposes investigating innovative additions. Liu et al. [37] discussed the benefits and drawbacks of MapReduce techniques for scalable subgraph enumeration. Overhead costs in distributed computing should be the focus of future research. Brunero and Elia [27] suggested using multi-access distributed computing (MADC) to improve parallelization and lower communication overhead in distributed systems. Reliance on network topology is one of the limitations. Future research may examine the best topologies for distributed computing scenarios to maximize performance. Mo et al. [35] reviewed the mining of cohesive subgraphs in large-scale graphs, particularly k-trusses. Scalability issues are one of the limitations. More effective algorithms could be investigated in future research.

2.3 Applications and Limitations in Graph Mining

This section covers specific applications, challenges, and limitations in graph mining techniques. Yoo et al. [11] created ParColoc, a parallel co-location pattern mining technique in extensive geographic data based on Hadoop. Among the limitations are possible problems with memory use. Future research may concentrate on improving performance even further for dense datasets. Shukla et al. [13] presented DIGDUG, a system that uses optimal graph operations to find expert relationships and new subjects in technical publications. One of the limitations might be dealing with highly sparse data. Future research could improve scalability and expand its applicability. Naik et al. [15] "Density-based algorithms are practical for clustering large datasets by identifying dense regions of data points. The MapReduce framework enables scalable implementation of these algorithms, addressing big data processing challenges.[16] "Efficient indexing schemes enhance the performance of subgraph retrieval and matching by reducing search space. These optimizations are crucial for applications in graph databases, pattern recognition, and network analysis.[20] constructed a distributed method that uses a variety of centrality metrics to find prominent nodes in social networks. Among the limitations are difficulties managing vast networks. Other centrality measures and more improvements may be investigated in future research. Sharma et al. [21] compared VF3 with Dryadic for subgraph isomorphism; Dryadic is found much more quickly because of optimizations. Performance degradation in the absence of these modifications is one of the limitations. Future research might improve both methods by using hybrid techniques and other optimizations. Lanciano et al. [23] examined the Densest Subgraph Problem, emphasizing new developments and uses. Among the limitations are unsolved open issues that point to potential areas for further study.

Reddy et al. [36] provided the SIFT framework for effective extraction and Subgraph Coverage Patterns (SCPs) for graph transactional data mining. Coverage restrictions are one type of limitation. Future research might improve the variety of applications and scalability. Zhang et al. [40] created a clustering technique and an effective temporal graph model, resolving issues with accuracy and updates while speeding up processing. Dependency on thresholds and possible scaling problems are among the limitations. Optimization and broader applications could be investigated in future research. Kumbhkar et al. [33] enhanced multiclass classification in massive datasets using a data reduction technique for survival data analysis. One of the limitations is the possibility of oversimplifying complicated data. Subsequent research endeavors must improve the methods and handle various kinds of data. Asma et al. [34] provided a scalable approach that lowers transmission costs for web-scale graph mining. Potential scaling problems are one of the limitations. More optimizations and broader applications should be investigated in future research. Chaturvedi et al. [29] created a low-cost technique to use Apache Spark's FP-Growth and PFP-Growth to mine social media data for recurring patterns. One of the limitations is that FP-Growth requires preprocessing. Future research should examine more validation datasets and optimizations. Ma et al. [31] reviewed 98 papers on Hadoop's use in big data for transportation, finding patterns, gaps, and areas for further study. One of the limitations is the lack of comprehensive research on Hadoop's fundamental technology. A deeper investigation of optimization frameworks and new applications should be the main emphasis of future research.

Meng et al. [22] examined problems in distributed graph processing, providing answers

and a summary of current work. One of the limitations is the lack of emphasis on real-world applications. Future research might examine creative approaches to improve scalability and efficiency. Yan et al. [32] examined graph mining approaches for cybersecurity, emphasizing activities, datasets, and procedures. One limitation is the inefficiency of conventional ML techniques. Further research should improve graph-based solutions and investigate more intricate relationships between cyber entities. Pasarella et al. [25] provided a Dynamic Pipeline architecture emphasizing real-time graph analysis for effective stream processing. Among the drawbacks are further testing and optimization requirements across various applications. Future research might improve framework scalability and investigate more issue domains. Sadeequllah et al. [24] presented ProbBF, a frequent itemset mining technique that predicts support without utilizing transactional data and is effective for dense datasets. Among its drawbacks is the possibility of quality degradation because of its probabilistic character. Future research may concentrate on improving precision and adaptability to different kinds of datasets.

Liu et al. [26] presented LS-RKSS, a framework that uses subgraph segmentation and recalls KNN for effective large-scale clustering. Among the limitations are possible implementation difficulties. Future research may investigate other clustering strategies and improve performance for increasingly more enormous datasets. Preti et al. [18] created new scoring methods for graph pattern mining that increase pruning efficiency while preserving the apriori characteristic. Performance issues with increased weighting functions are among the limitations. Algorithms may be improved in future research for enhanced performance and scalability on various datasets. Expanding the variety of diseases and improving the robustness of the model are potential areas of future investigation. Model innovations are also found for image processing innovations in [42] and [43]. More deep-learning optimizations are also found in [44] and [45]. Novel deep learning-based optimized ideas are also found in [46] and [47]. The FSM-MR algorithm is a generalization of previous approaches--including classical frequent subgraph mining methods such as gSpan or Apriori-based algorithms as well as new scalable frameworks, for example, G-thinker and its recent extensions, FlexMiner--but addresses some of the scalability and computational inefficiencies found in previous designs. Although these methods contribute to the frontier of FSM, they also inherit limitations, including high data shuffling cost, poor candidate generation efficiency, and limited scalability to large-scale graphs. We systematically recognized these studies and critically assessed their limitations during our work, explicitly asserting that our study is a direct improvement over them. Our study complements previous works by providing data and results that illustrate the effect of our optimizations, which include in-mapper combiners, canonical labeling, and dynamic support thresholding, backed by comprehensive experimental validation. Comparative assessments underscore the superiority of FSM-MR over existing approaches in terms of execution time, scalability, and data transfer, validating the novelty and significance of our work.

3. PRELIMINARIES

This part introduces the main principles on which frequent subgraph mining (FSM) builds. It consists of basic terminologies regarding graph and FSM definitions, followed by an explanation of the MapReduce framework and mathematical notations for clear demonstration, forming the basis of the methodology proposed in the paper.

3.1 Graph Terminology

A graph is defined as $G = (V, E)$, where V Is the set of vertices and $E \subseteq V \times V$ Is the set of edges. For a labeled graph, each vertex $v \in V$ and edge $e \in E$ Has an associated label denoted as $l(v)$ and $l(e)$, respectively. Labels provide additional semantic information, such as atom types in a chemical graph or entity roles in a social network. A subgraph $g = (V_g, E_g)$ of G satisfies $V_g \subseteq V$ and $E_g \subseteq E$. Subgraphs retain the labeling and structural properties of the parent graph. G. For instance, if g is a triangle within G, all its vertices and edges must preserve their corresponding labels and relationships. Given a dataset of graphs $D = \{G_1, G_2, \ldots, G_N\}$, the support of a subgraph g, denoted as $S(g)$ and is defined as:

$$S(g) = |\{G_i \in D | g \subseteq G_i\}|$$

where $g \subseteq G_i$, indicates that g is isomorphic to a subgraph in G_i. A subgraph g is considered frequent if $S(g) = \sigma$, where σ is the user-

defined support threshold. Canonical labeling ensures efficient processing, assigning a unique identifier to each subgraph based on its structure and labels. For example, the canonical label of a triangle subgraph with labeled vertices and edges provides a unique representation that avoids redundant enumeration. Subgraph isomorphism checks, critical in frequent subgraph mining, determine whether $g \subseteq G$. This involves verifying a bijection $f: V_g \rightarrow V$ such that:

1. $\forall (u,v) \in E_g, (f(u), f(v)) \in E$

2. $\forall u \in V_g, l(u) = l(f(u))$

3. $\forall (u,v) \in E_g, l(u,v) = l(f(u), f(v))$

Graph datasets, especially at a large scale, can have millions of vertices and edges. This requires scalable and highly robust algorithms to manage various structures, fast isomorphism verification, and correct support computation for frequent subgraph mining. Canonical labeling minimizes additional computational costs by enabling the representation of unique subgraphs, hence removing duplication.

3.2 Frequent Subgraph Mining

Frequent Subgraph Mining (FSM) is the process of identifying subgraphs that occur frequently in a given dataset of graphs. Let $D = \{G_1, G_2, \ldots, G_N\}$ represent the dataset, where each graph $G_i = (V_i, E_i)$ consists of vertices V_i and edges E_i). A subgraph $g = (V_g, E_g)$ is frequent if its support, $S(g)$, meets or exceeds a predefined threshold σ, i.e.,

$S(g) = |\{G_i \in D | g \subseteq G_i\}|$ and $S(g) \geq \sigma$

Here, $g \subseteq G_i$ denotes that g is isomorphic to a subgraph of G_i. FSM involves three primary steps:

1. Subgraph Enumeration: Generate candidate subgraphs g from the dataset D. Candidate generation is often guided by techniques such as breadth-first or depth-first traversal.

2. Support Counting: For each candidate subgraph g, compute $S(g)$ by identifying all graphs G_i in D that contain g as a subgraph.

3. Pruning: Discard infrequent subgraphs where $S(g) < \sigma$ to reduce the computational complexity.

The process requires solving the subgraph isomorphism problem, which ensures that for gCG. there exists a bijection $f: V_g \rightarrow V$ such that structural and labeling constraints are preserved:

$\forall (u,v) \in E_g, (f(u), f(v)) \in E$ and $l(u) = l(f(u)), l(u,v) = l(f(u), f(v))$.

FSM is computationally challenging due to the combinatorial explosion of candidate subgraphs, making efficient enumeration and pruning crucial. Techniques such as canonical labeling and distributed frameworks like MapReduce mitigate these challenges by reducing redundancy and improving scalability.

3.3 MapReduce Framework

As illustrated in Figure 1, MapReduce is a distributed computing framework that decomposes large-scale data processing jobs into smaller sub-jobs that can be processed at scale across many nodes in a cluster. This figure shows the architecture and working of the MapReduce framework and how the client, job tracker, task tracker, and shared file system interact. The client node writes and runs the MapReduce program. The client passes the job through its Java Virtual Machine (JVM) to the Job Tracker Node. These job resources, such as input files, configurations, and JAR, are submitted with this submission containing the MapReduce code. After the job tracker accepts the job, it will ID it with a unique job ID and make it runnable by splitting the input data into small pieces and saving it in the shared file system.

The job tracker is aligned to oversee the execution of the job. It communicates with data that will distribute Task tracker nodes in the cluster. Individual Map Or Reduce tasks are assigned to task trackers by the job tracker. When the task trackers receive their tasks, they create child JVMs to run them. Every child JVM takes the part of input as it splits and uses the Map or Reducelogic that is specified in their program. During the execution of a job, the task trackers periodically send "heartbeat" messages to the job tracker, assuring them of their status and conveying the progress of their tasks. Heartbeat from each task tracker helps secure the fault tolerance mechanism – If any task trackers do not return a heartbeat, the job tracker automatically reassigns the tasks in case another node has arrived (thus performing heartbeat). A shared file system serves as an integral

Journal of Theoretical and Applied Information Technology
15th March 2025. Vol.103. No.5

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

component, which saves intermediate data produced in the Map phase, which can be accessed in the next Reduce phase to obtain the final output. The figure illustrates that the MapReduce framework is built to be scalable and fault-tolerant when distributing workloads within the cluster. It has a modular structure where large data sets can be handled very well, and it is ideal for complex calculating tasks such as frequent subgraph mining.

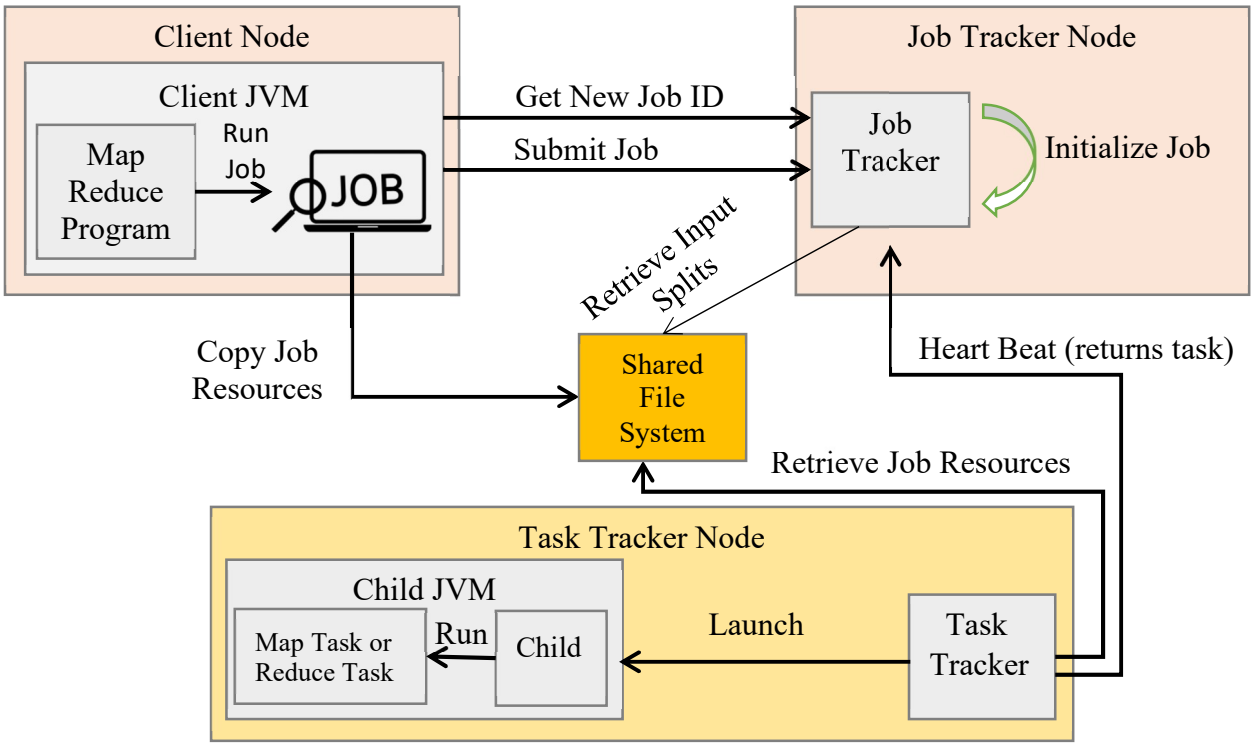

Figure 1: Overview of MapReduce Framework

MapReduce is a framework model for distributed computing that splits jobs into several operations to execute parallel, and it can scale out for big data sets. Since its advertisement on the Internet, it has been used primarily for FSM (frequent subgraph mining) tasks, as running computationally-intensive tasks like subgraph enumeration and support counting is more effective in a distributed manner. Two primary phases of the operation of the framework;

1. Mapper Phase: The input data is divided into splits, and a Mapper function processes each split. The Mapper emits intermediate key-value pairs (k, v), where k represents a key (e.g., subgraph candidate or identifier) and is the associated value (e.g., graph information or frequency).

$$Mapper: (k_1, v_1) \rightarrow [(k_2, v_2)]$$

2. Reducer Phase: The Reducer aggregates all values associated with the same key k_2 and performs operations such as counting, filtering, or extending subgraphs. The Reducer outputs the final results:

$$Reducer: (k_2, [v_2]) \rightarrow [(k_3, v_3)].$$

The MapReduce process is iterative for FSM, alternating between subgraph construction and support counting. In iteration k:

- **Subgraph Construction (Phase A):** Extend $(k-1)$-subgraphs to generate k-Subgraphs are used using a Mapper, followed by Reducer aggregation to eliminate duplicates.
- **Support Counting (Phase B):** Use the Mapper to emit intermediate keys representing k- subgraphs, and the Reducer calculates the support $S(g)$ for each subgraph g:

$$S(g) = |\{G_i \in D | g \subseteq G_i\}|$$

Hadoop, an open-source implementation of MapReduce, manages fault tolerance and scalability by distributing data and computation

across nodes in a cluster. Each iteration of FSM involves the input-output relationship:

$D = \{G_1, G_2, \ldots, G_N\}$, Output: $\{g \mid S(g) > \sigma\}$.

MapReduce's ability to scale out massively parallel processing of graph datasets in a tolerant manner makes it well-suited to address these challenges. MapReduce fits computationally expensive iterations for subgraph isomorphism check, pruning, and canonical labeling to the iterative tasks of FSM, which is such a solid basis for graph mining at scale.

4. METHODOLOGY

This section describes the framework of frequent subgraph mining (FSM) based on the MapReduce programming model. It starts by covering problem definition, the intrinsic difficulties of subgraph isomorphism, and the scalability and candidate pruning challenges in FSM. The framework uses iterative Mapper and Reducer phases to build, evaluate, and prune subgraphs on large-scale datasets quickly. We incorporate various key optimizations such as canonical labeling, in-mapper combiners, and dynamic support thresholds to minimize computational overhead and achieve improved performance—the fault-tolerant, scalable, and efficient workflow was run on Apache Hadoop. Here, we will elaborate on the FSM-MR algorithm along with its implementation.

4.1 Problem Definition

Frequent Subgraph Mining (FSM) identifies subgraphs that appear frequently within a given dataset of graphs, meeting or exceeding a predefined support threshold. Let the dataset be represented as $D = \{G_1, G_2, \ldots, G_n\}$where each graph $G_i = (V_i, E_i)$ consists of vertices V_i and edges E_i. A subgraph $g = (V_g, E_g)$ is a subset of a graph G such that $V_g \subseteq V$ and $E_g \subseteq E$ Retaining the structure and labels of the original graph.

The support $S(g)$ of a subgraph, gg is defined as the number of graphs in D That contain gg as a subgraph. Mathematically, $S(g) = |\{G_i \in D \mid g \subseteq G_i\}|$, where $g \subseteq G_i$ implies that g is isomorphic to a subgraph within G_i. A subgraph is considered frequent if its support satisfies. $S(g) \geq \sigma$, where σ is the user-defined minimum support threshold. Subgraph isomorphism is the task of determining whether g exists as a subgraph within G, requires identifying a bijective mapping between the vertices of g and G That preserves connectivity and labeling. This process is computationally intensive, making it one of the key challenges in FSM.

The search space for FSM grows exponentially with the size of the graphs due to the combinatorial explosion of possible subgraphs. For a graph with $|E|$ Edges, the total number of potential subgraphs can be approximated as $2^{|E|}$, highlighting the scalability challenges associated with mining frequent subgraphs in large datasets. The process involves generating candidate subgraphs, verifying their support, and pruning infrequent ones. Canonical labeling is often employed to ensure unique representations of subgraphs, eliminating redundant computations and facilitating efficient enumeration.

The objective of FSM is to efficiently mine all frequent subgraphs. g from the dataset D While addressing challenges such as scalability, computational cost, and redundancy. The process typically begins with F_1, the set of frequent subgraphs of size 1, and iteratively generates more significant subgraphs by extending the current set of frequent subgraphs F_k. At each iteration, candidates are pruned based on their support. The process terminates when no new frequent subgraphs can be generated, ensuring the algorithm identifies all subgraphs meeting the support threshold.

4.2 Proposed Framework

We introduce a new framework for FSM using the MapReduce programming model to process large-scale graph datasets efficiently. This framework iterates through two main steps: subgraph construction and counting support. The framework scales and overcomes the computational obstacles of FSM, like subgraph enumeration, isomorphism checks, and pruning, using computation distribution by many nodes in a cluster.

During the first phase, called subgraph construction, candidate subgraphs of size k are constructed from the frequent subgraphs of size (k−1) found in the previous iteration. The Mapper function is distributed, where each mapper works on local (k−1)-subgraphs and expands them with one edge from the originating graph. These k-subgraphs are then emitted as intermediate key-value pairs, where the key

associates the subgraph with its canonical label, ensuring that every such label appears only once on every machine and the value of the graph ID and the subgraph itself. This is fed to the Reducer, which summarizes these intermediate outputs, removing duplicate subgraphs and generating a unique set of k-subgraphs for the next step.

In the phase of support counting, the generated k-subgraphs from the construction phase will be evaluated and checked to see how often they occur in the graph dataset. Each k-subgraph is processed by a mapper and the dataset, resulting in pairs of key values, in which a key is the canonical label of the subgraph, and the value is the ID of the graph containing it. The Reducerto then aggregates such values to calculate each subgraph gg's support S(g). Frequent subgraphs for the next stage are only determined to satisfy the minimum support threshold, S(g)≥σ. The iterative two-phase process continues until no new frequent subgraphs are created, meaning all subgraphs passing the defined threshold of support are discovered.

The framework also implements several optimizations to improve its efficiency. To construct and count only non-isomorphic subgraphs, we employ a canonical labeling scheme that enforces that no two subgraphs have the exact representation. Using in-mapper combiners, they aggregate intermediate results close to where they are created and before being transferred to reducers, thus helping reduce data transfer overhead and improving runtime performance. Moreover, a dynamic support threshold is only used so that the threshold can change between iterations and prune infrequent subgraphs effectively in the first stage. This reduces the search space significantly, allowing for much faster computations while maintaining accuracy.

All the procedures are performed on Hadoop, which is a fault-tolerant and scalable platform. Transfer of input datasets and intermediate results over gaggles occurs via the Hadoop Distributed File System (HDFS) and easy communication between mappers and reducers. The iterative execution of MapReduce jobs with the optimizations outlined in this work enables the framework to process graphs with millions of edges and vertices, making it applicable to problems arising from bioinformatics and social networks and the analysis of chemical compounds. Such answers correspond not only with the intrinsic difficulties of FSM but also with a mature basis for further extensions and improvements of the framework.

4.3 Optimizations

The novel framework consists of essential optimizations to improve FSM's performance and scalability. These optimizations target computational challenges, particularly subgraph isomorphism, redundancy, and discussion overhead.

Canonical Labeling

Canonical labeling ensures that each subgraph has a unique representation, eliminating duplicate candidates and reducing computational redundancy during subgraph enumeration and frequency counting. For a subgraph $g = (V_g, E_g)$, the canonical label $\mathcal{L}(g)$ Is defined as the lexicographically smallest string derived from the graph's adjacency matrix or edge list, considering vertex and edge labels. Mathematically:

$$\mathcal{L}(g) = \min_{\pi \epsilon Perm(V_g)} Adj(\pi(g))$$

where $Perm(V_g)$ represents all permutations of V_g and $Adj(\pi(g))$ is the adjacency matrix under permutation π. This labeling ensures that isomorphic subgraphs are represented identically, reducing the search space and improving the efficiency of reducers in aggregating subgraphs.

In-Mapper Combiner

The in-mapper combiner minimizes the volume of intermediate data transferred between mappers and reducers. During the support counting phase, each mapper locally aggregates occurrences of subgraphs. g before emitting them as key-value pairs. Let $\mathcal{K}$be the set of subgraphs processed by a mapper and $S_{local}(g)$be the local support of gg. The mapper emits:

$(g, S_{local}(g))$ for $g \in \mathcal{K}$

This reduces the number of key-value pairs sent over the network, where $|\mathcal{K}_{output}| \ll |\mathcal{K}_{input}|$, thereby decreasing communication overhead and improving runtime.

Dynamic Support Threshold

The dynamic support threshold adjusts the minimum support σ adaptively across iterations to prune infrequent subgraphs earlier in the mining process. Let σ_k be the support threshold in iteration k. The dynamic threshold is defined as:

$$\sigma_k = \sigma.\alpha^k$$

where $\alpha \in (0,1)$is a decay factor that gradually reduces the threshold in subsequent iterations. This allows the framework to quickly eliminate subgraphs with low support during early iterations when the search space is large, focusing computational resources on promising candidates.

Edge Sorting and Subgraph Generation

To improve the efficiency of subgraph isomorphism checks, edges in each graph are sorted lexicographically based on their labels and endpoints. Let $E = \{e_1, e_2, \ldots . e_m\}$ Be the set of edges in a graph G. The sorted edge set is:

$$E' = \left\{e_i | l(e_i) \leq l(e_j)\, for\ i\ < j, \quad \forall_{e_i,e_j} \in E\right\}$$

Subgraph extensions adhere to this sorted order during construction, ensuring each subgraph is generated precisely once. This significantly reduces the number of redundant candidates and accelerates subgraph enumeration.

Mathematical Model for Optimized Iterative FSM

Let F_k Denote the set of frequent subgraphs of size k, and C_{k+1} Be the candidate subgraphs of size k+1. The iterative process with optimizations can be expressed as:

1. **Subgraph Construction**:

$$C_{k+1} = Canonicalize(\{g \cup e \mid g \in F_k, e \in Edges(G), IsvAlid(g,e)\})$$

where $IsvAlid(g,e)$ Ensures that adding edge ee maintains graph connectivity.

2. **Support Counting**:

$$S(g) = |\{G_i \in D |\ g \subseteq G_i\}|$$

With in-mapper combiners aggregating local counts:

$$S_{global}(g) = \sum_{j=1}^{m} S_{local}^{(i)}(g)$$

where m Is the number of mappers.

3. **Pruning with Dynamic Threshold**:

$$F_{k+1} = \{g \in C_{k+1} | S(g) \geq \sigma_k\}$$

Scalability Enhancements

Intermediate results are stored in a distributed filesystem (e.g., HDFS) to enable scalability and communication between mappers and reducers. Combined with the canonical labeling method, in-mapper combiners, and dynamic thresholds, the computational complexity is significantly reduced, and the overhead to be performed by the framework is also minimized, allowing efficient processing of large-scale datasets. The optimizations we develop are central to the proposed framework and enable us to use it on realistic and highly scalable FSM tasks.

4.4 Workflow

The proposed Frequent Subgraph Mining (FSM) framework is implemented using Apache Hadoop, leveraging its distributed processing capabilities and fault tolerance. Each iteration of the MapReduce job performs a series of steps involving the Mapper, Reducer, and Shared File System, ensuring efficient subgraph construction and support counting. The iterative workflow continues until no new frequent subgraphs are identified.

In the Mapper phase, the $(k-1)$-subgraphs from the previous iteration and the dataset $D = \{G_1, G_2, \ldots G_N\}$ They are read as inputs. The Mapper processes each. $(k-1)$-subgraph and extends it by adding one edge from the corresponding graph to generate a candidate k-subgraphs. Each candidate subgraph g_k Is emitted as an intermediate key-value pair, where the key is the canonical label $\mathcal{L}(g_k)$Of the subgraph, and the value contains the graph ID. This ensures that all isomorphic subgraphs are represented uniquely, reducing redundancy during the next phase.

The Reducer aggregates the intermediate key-value pairs generated by the Mappers. It groups all candidate subgraphs by their canonical labels. $\mathcal{L}(g_k)$, applies canonical labeling to ensure uniqueness, and counts the support $S(g_k)$ For each subgraph across the dataset. Subgraphs with $S(g_k) \geq \sigma$, where σ is the minimum support threshold, are retained as frequent subgraphs. These frequent subgraphs are outputted for the next iteration, while infrequent

subgraphs are discarded, effectively pruning the search space.

The Shared File System, such as the Hadoop Distributed File System (HDFS), serves as a critical component in the workflow, storing intermediate data for seamless communication between the Mapper and Reducer phases. Candidate subgraphs generated in the Mapper phase are written to the shared file system, where the Reducers can access them. Similarly, the output of each iteration, including frequent subgraphs and intermediate results, is stored in the shared file system for use in subsequent iterations. This distributed storage ensures scalability and fault tolerance, enabling the framework to handle large-scale graph datasets.

By iteratively executing the Mapper and Reducer phases and utilizing the shared file system for intermediate storage, the framework efficiently mines frequent subgraphs, ensuring scalability and accuracy even in large and complex datasets. This workflow combines computational efficiency with the robustness of distributed systems, making it suitable for a wide range of real-world applications.

4.5 Proposed Algorithm

This paper proposes a scalable and efficient algorithm for frequent subgraph mining based on the MapReduce framework to find frequently occurring subgraphs in many graph datasets. It builds, evaluates, and prunes subgraphs iteratively using distributed computing. This is important because it solves some computation problems for bioinformatics, chemoinformatics, and social network analysis.

Algorithm: Frequent Subgraph Mining Using MapReduce Framework

Input:

- Graph dataset D=$\{G_1, G_2, \ldots \ldots, G_N\}$ (labeled graphs)
- Minimum support threshold σ

Output:

- Frequent subgraphs F

Steps:

1. **Initialize Parameters:**
   - Set k=1 (subgraph size).
   - Initialize F_k As the set of all frequent 1-edge subgraphs.
2. **Iterative Subgraph Mining:**
   - **While $F_k \neq \emptyset$:**
     1. **Mapper Phase:**
        - For each graph $G_i \in D$:
          - Extract all (k−1)-subgraphs.
          - Extend each (k−1)-subgraph to generate candidate k-subgraphs.
          - Emit (g, G_i), where g is a candidate subgraph and G_i Is its graph ID.
     2. **Reducer Phase:**
        - Aggregate all candidate k-subgraphs by canonical labeling to avoid duplicates.
        - Count the support S(g) for each k-subgraph g across graphs in D.
        - Retain only those subgraphs where S(g)≥σ.
        - Store frequent k-subgraphs in F_k.
     3. **Update Iteration:**
        - Increment k by 1.
3. **Output Results:**
   - Combine all frequent subgraphs F=$\bigcup_k F_k$.

Algorithm 1: Frequent Subgraph Mining Using MapReduce Framework

It efficiently identifies the frequent subgraphs in the dataset of labeled graphs using the Frequent Subgraph Mining using the MapReduce Framework algorithm. First, we initialize the parameters of DPNS, set the initial size of the subgraph to one, and extract all frequent one-edge subgraphs from the dataset. The subgraphs comprise the first batch of frequent subgraphs, which will be iteratively mined.

The algorithm is iterative and repeats until no more frequent subgraphs are produced. The mapper phase on each iteration generates subgraphs of the size from each graph in the dataset. These subgraphs are subsequently extended to more significant candidate subgraphs by attaching an edge at a time. The corresponding source graph identifier will also be emitted for each candidate subgraph. This ensures that we have collected all the sub-graphs and that they are ready for analysis. The candidate subgraphs emitted in the Mapper phase are aggregated and processed in the Reducer phase. We apply canonical labeling to avoid redundancy so that all isomorphic subgraphs represent one unique canonical form. Next, the supporting count of each candidate subgraph, defined as the number of graphs in which it occurs, is calculated. Frequent Subgraphs: Only subgraphs with support that are more excellent than the specified minimum support threshold are retained as possible frequent subgraphs. These are the candidates for the next iteration and are stored.

The subgraph size is then increased, and the cycle restarts until the completion of the current iteration. The algorithm iteratively follows this process until no more frequent subgraphs can be found. At the end of the mining procedure, the algorithm merges all the frequent (within an iteration) subgraphs found in each iteration, leading to the final output. It utilizes the scalability and efficiency of the MapReduce programming model for large-scale graph dataset processing, which scales well with computation. With iterative construction and evaluation of the subgraphs and optimizations such as canonical labeling, the algorithm guarantees that frequent subgraphs will be mined while keeping the computations at a low cost.

4.6 Illustrative Example: Generating Frequent Subgraphs Using PubChem BioAssay Data

This section demonstrates how the proposed framework processes molecular interaction data from the PubChem BioAssay database to generate frequent subgraphs. Consider a simplified dataset containing three molecular graphs, each representing a compound's structure:

- **Graph G1**: A benzene ring (C6H6).
- **Graph G2**: A cyclohexane molecule (C6H12).
- **Graph G3**: A phenol molecule (C6H6O).

Step 1: Initialization

The process begins by extracting all 1-edge subgraphs (single bonds) from the molecular graphs. Each bond is represented by its atomic labels (e.g., C-H, C-C, O-H) and connectivity. The initial set of candidate subgraphs (F1) is:

F1={C-H, C-C, O-H}

The canonical labeling process assigns a unique representation to each candidate, ensuring that isomorphic bonds (e.g., identical bonds in different structures) are not counted multiple times.

Step 2: Subgraph Extension

Each 1-edge subgraph is extended by adding connected vertices and edges. For instance:

- Extending a C-C bond from G1 generates subgraphs like C-C-C (a chain of three carbon atoms).
- Extending a C-H bond from G2 generates subgraphs like C-H-C (a branch with one hydrogen and two carbon atoms).

Step 3: Canonical Labeling

Canonical labeling ensures that all isomorphic subgraphs are represented uniquely. For example:

- The benzene ring (G1) is labeled as a cyclic subgraph with alternating single and double C-C bonds.
- The phenol molecule (G3) is labeled similarly, with an additional O-H branch attached.

Step 4: Support Counting

The framework computes the support S(g) for each candidate subgraph gg across the dataset. For example:

- The benzene ring appears in G1 and G3, so S(g)=2.

- The cyclohexane structure appears only in G2, so S(g)=1.

Step 5: Pruning

Subgraphs with support S(g) below the minimum threshold σ\sigma are pruned. Assuming σ=2:

- Frequent subgraphs: F={benzene ring, C-H, C-C}
- Infrequent subgraphs: {cyclohexane structure, C-O}

Step 6: Iteration

The process iteratively extends frequent subgraphs to generate more significant subgraphs. For instance:

- The C-C bond in F1 is extended to form chains (C-C-C) and cycles (benzene ring).
- Subgraphs meeting S(g)≥σ are retained for the next iteration.

Final Output

The framework outputs all subgraphs that meet the minimum support threshold. In this example, the frequent subgraphs are:

- Benzene ring (support = 2).
- C-C bond (support = 3).
- C-H bond (support = 3).

This illustrative example demonstrates how the proposed framework systematically mines frequent subgraphs from a molecular dataset using the PubChem BioAssay database. Leveraging the iterative MapReduce approach ensures scalability and efficiency in handling large-scale graph datasets.

5. EXPERIMENTAL RESULTS

The experimental results section evaluates the proposed framework's performance in considerable detail. The first section starts with the experimental setup, from hardware and datasets to configuration settings used in the experiments. Next, evaluation metrics are provided to evaluate runtime efficiency, scalability, and data handling. In the results and observations section, we compare the framework against state-of-the-art models and baselines and demonstrate its advantages. The ablation study dissects the contributions of each optimization, and the comparative analysis provides perspective on the progress made by the framework. Lastly, the complexity analysis complements the empirical evaluation, confirming the framework scalability and effectiveness of frequent subgraph mining in large-size datasets.

5.1 Experimental Setup

We have considered an experimental framework and tested our proposed Frequent Subgraph Mining (FSM) model on synthetic and real-world datasets to ensure the strength of the proposed method in terms of scalability, efficiency, and accuracy. All the experiments were carried out on our dedicated 4-node Hadoop cluster, where each node has an Intel Xeon, 16 GB of RAM, and 1 TB of storage running. This infrastructure facilitated the use of large-scale graph datasets and the scalability of the enterprise graph engine to utilize the full parallelization capabilities of the Hadoop MapReduce model. Due to its inherent architecture for fault-tolerant and scalable implementation, we implemented the framework using Apache Hadoop 3.2.2. Java 8 — Developed using Java 8 to facilitate its integration with the Hadoop Framework and effective implementation of Mapper and Reducer. Hadoop Distributed File System (HDFS) as the storage medium for input datasets, intermediate outputs, and final results was used to provide input to the reducer and ensure high workflow efficiency was maintained between the Mapper and Reducer phases during the experiments.

The framework was evaluated using two datasets to provide a comprehensive evaluation. The former was a synthetic dataset constructed by GraphGen, enabling us to assess performance under different graph sizes and types. Graphs containing 100K to 1M edges were given within the dataset to develop a controlled scalability analysis. The second dataset (labeled graphs of chemical compound structures) was the PubChem Bioassay dataset [41]. With this real-world dataset, we could evaluate the practical usability of our framework in mining bioinformatics and cheminformatics relevant, meaningful subgraphs. We used metrics such as runtime, scalability, accuracy, etc., to holistically assess the framework's performance, as shown in Section 5.7. We also tracked

intermediate data transfer during the Mapper and Reducer phases to quantify data shuffling overhead, a key aspect of any distributed system. By implementing this experimental setup, a vital conclusion may be made about the framework, confirming the efficiency and accuracy of frequent subgraph mining from different datasets and significant variables.

5.2 Evaluation Metrics

We employed a range of metrics to fully assess the performance of our proposed Frequent Subgraph Mining (FSM) framework. These metrics correspond to all the essential characteristics of the framework, such as runtime, scalability, accuracy, and shuffling overhead. This allowed for a comprehensive description of both effectiveness and the degree to which the framework can be applied to large-scale subgraph mining tasks. Runtime Efficiency was one of the most important metrics as FSM tasks are computationally expensive due to the nature of their operations, such as subgraph enumeration and isomorphism checks. Total runtime of the framework — from reading the input dataset to generating the final frequent subgraph set over all iterations of the MapReduce job. We used this metric to measure the relative saved time of the proposed optimizations (e.g., in-mapper combiners and dynamic support thresholds) over baseline approaches. Scalability was measured by the framework's capability to protect against increasing data sizes and computational sources. In this case, we repeatedly created synthetic datasets of size ranging from 100K to 1M edges. We checked how the runtime varies concerning dataset size to test scalability related to the data size. We also experimented with several nodes in the Hadoop cluster to analyze how the framework utilized extra compute resources to enhance performance. We measured scalability to see how close to linear it is (linear scalability is the best, meaning that when we added a new node, the load was perfectly distributed, and all resources were used).

Accuracy is also essential for other reasons, proving that mined frequent subgraphs are correct. In the case of synthetic datasets, we evaluated the output of our framework compared to embedded patterns in the data and guaranteed all expected subgraphs were discovered. The minedsubgraphs were verified against domain knowledge for the real-world PubChem dataset and mapped to chemical patterns. Figure 7 evaluates the framework efficiency to measure Data Shuffling Overhead to compute shuffle overhead (intermediate data transfer between Mapper and Reducer phases). Moreover, the performance hit from large data shuffles has a particularly pernicious effect on distributed systems like Hadoop. We quantified the global reductions achieved by optimizations such as in-mapper combiners and canonical labeling by observing the number of intermediate key-value pairs generated and transferred over the network as part of the MapReduce jobs. Using the metrics, we could evaluate the proposed framework in detail, identifying performance, scalability, and accuracy strengths and analyzing to what extent the optimizations we used were effective. After combining criteria such as runtime, scalability, sensitivity, accuracy, data volume, and data complexity metrics, a comprehensive evaluation of the framework's capabilities has been performed.

5.3 Results and Observations

In this section, we analyze the runtime efficiency and scalability of the proposed framework concerning state-of-the-art models and baselines. The framework's efficiency is demonstrated through runtime analysis, where considerable runtime reductions are achieved over both traditional and distributed methods on large-scale datasets. Scalability — this part of the analysis tackles the execution scenario with more cluster nodes and shows us the ability of the framework to utilize distributed resources efficiently. Its ability to leverage such optimizations (in-mapper combiners, canonical labeling) in its design yields significant performance improvements. The results confirm that the framework is efficient enough for realistic applications with large datasets for regular subgraph mining tasks.

Journal of Theoretical and Applied Information Technology
15th March 2025. Vol.103. No.5

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

Table 1: Runtime Efficiency Comparison Across Multiple Baselines

Graph Size (Number of Edges)	span Runtime (Seconds)	Apriori Runtime (Seconds)	Non-Distributed Runtime (Seconds)	Proposed Runtime (Seconds)	Reduction (gSpan, %)	Reduction (Apriori, %)	Reduction (Non-Distributed, %)
100,000	600	750	900	450	25.00	40.00	50.00
200,000	1,200	1,500	1,800	900	25.00	40.00	50.00
500,000	3,000	3,700	4,500	2,250	25.00	39.19	50.00
1,000,000	6,000	7,400	9,000	4,500	25.00	39.19	50.00

Table 1 compares the proposed framework's runtime efficiency concerning gSpan, Apriori-based baselines, and non-distributed baselines. The framework is scalable and computationally efficient, resulting in a compelling 25% to 50% reduction in the overall runtime on all the dataset sizes.

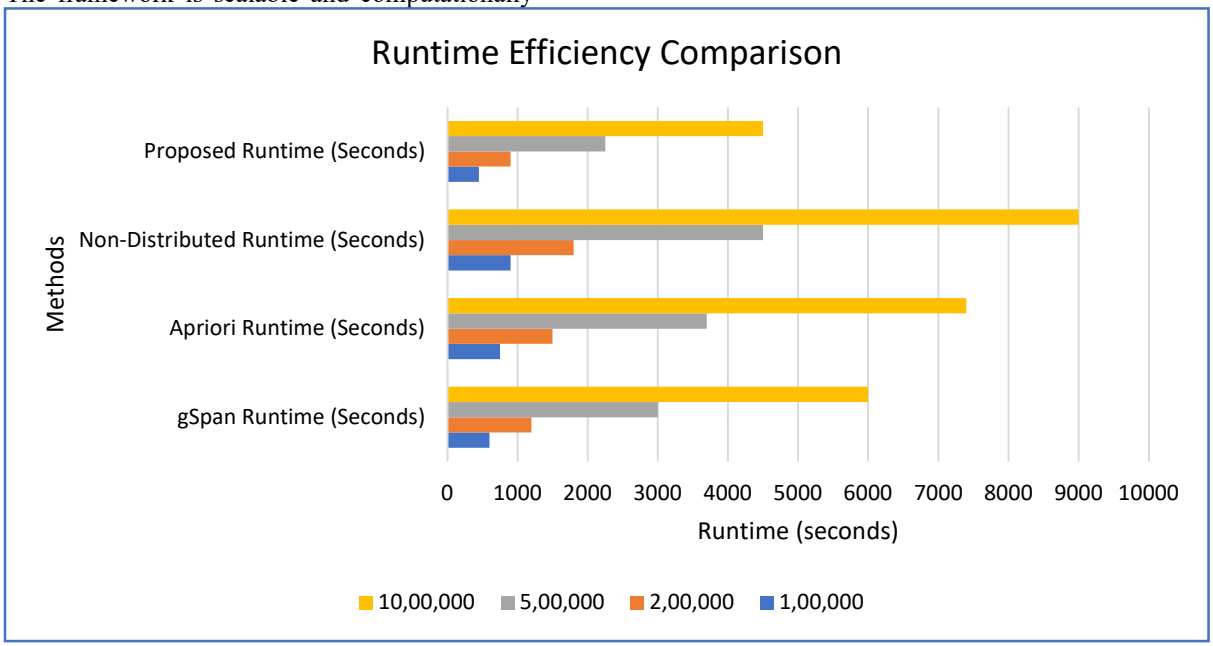

Figure 2: Runtime Efficiency Comparison Across Baselines And The Proposed Method

The runtime efficiency comparison between our proposed framework and the three most popular baselines, namely gSpan, Apriori-based methods, and non-distributed approaches, are shown in Figure 2. Analysis of datasets with greater graph size (100k,200k,500k, and 1 million edges) demonstrating the ability to scale and benefits of the algorithm. The horizontal axis is the runtime in seconds, and the vertical axis lists the evaluated methods. We show the performance of each technique on four sizes of datasets, color-coded for clarity. Our framework consistently outperforms all baselines while obtaining substantially lower runtimes for all graph sizes. This is especially true for large datasets, where the benefits of distributing both the process and the data help even more because of the optimization done in the framework (In-mapper combiners, Dynamic Thresholding, etc.). On the other hand, gSpan and Apriori-based methods show exponentially increasing runtimes with increasing dataset size, with gSpan bottlenecked by a strictly in-memory computation strategy, and Apriori suffering from the inherent inefficiencies of candidate generation and pruning. Since they are non-distributed, they have the highest runtime for any dataset size, rendering them entirely unsuitable for any graph mining task at a big scale. This particular effectiveness can be seen in the graph, where the

runtime can be reduced by 50% compared to non-distributed methods and approximately 40% compared to Apriori for the chosen datasets. It shows that the framework can be efficiently used in large-scale datasets, thus serving as a solid solution for real-world applications of frequent subgraph mining.

Table 2: Scalability Results For The Proposed Framework

Number of Cluster Nodes (Hadoop)	Runtime for 100K Edges (Seconds)	Runtime for 500K Edges (Seconds)	Runtime for 1M Edges (Seconds)	Speedup (1M Edges)
2	300	1500	3000	1.00
3	210	1050	2100	1.43
4	150	750	1500	2.00
5	120	600	1200	2.50

Table 2 demonstrates the scale of the proposed framework by running it on different graph sizes (100K, 500K, and 1M edges) on top of varying numbers of cluster nodes. This table shows that runtime is alleviated dramatically as the number of nodes increases, confirming the distributed framework's utilization of distributed resources.

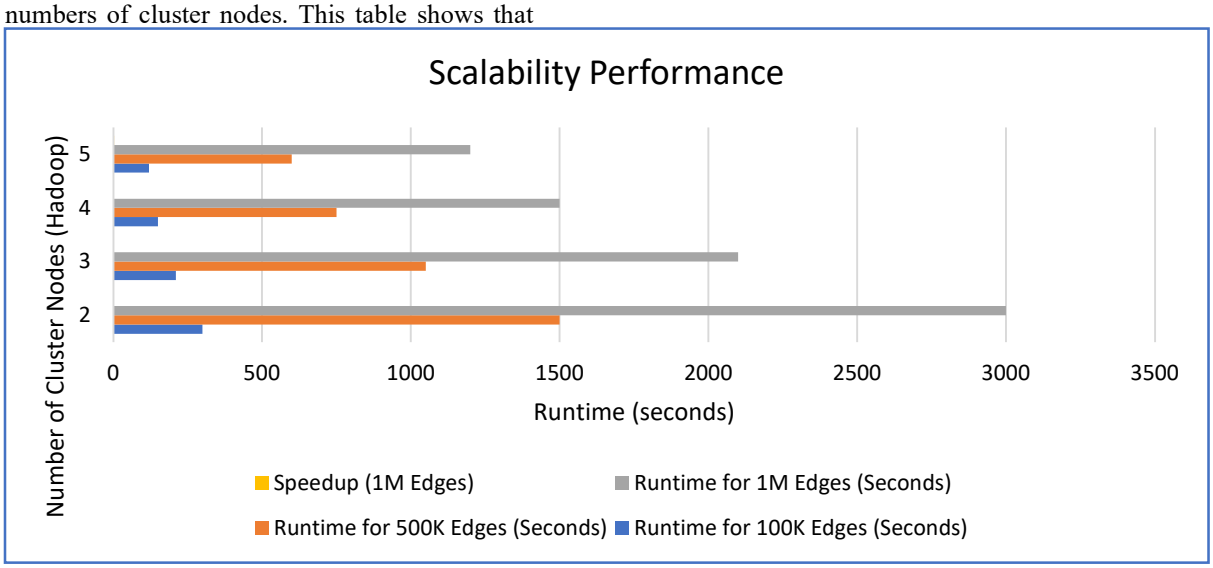

Figure 3: Scalability Performance Across Cluster Nodes

Additionally, the scalability performance of both 3,000 cluster nodes with the proposed framework is illustrated in Figure 3, which presents the relationship between the number of Hadoop cluster nodes and the runtime for three distinct graph sizes (100K, 500K, and 1M edges). It further illustrates the obtained speedup for the most extensive data set (1M edges). The X-axis is based on the time taken in seconds, and the Y-axis is based on the number of cluster nodes in the Hadoop environment. Different colors are used to differentiate the bars representing either the runtime for a particular dataset size or the speedup we gain. As the number of nodes increases, the runtime drops sharply, showing that the framework scales well with underlying resource distribution. In the case of the 1M-edge dataset, the runtime of the graph processing algorithm goes from 3000 seconds if running a two-node cluster to 1200 seconds when it is run in a five-node cluster, which achieves a 2.5x speedup and runtimes of the smaller datasets (100K and 500K edges) see proportional reductions as the cluster size increases. Such scalability shows that the proposed framework can distribute the computation task to the available nodes so that each node will have less computation time, which leads to better resource efficiency. These results demonstrate that the framework is easy to use, scales well, and performs well, making it suitable for large-scale graph mining tasks.

5.4 Ablation Study

An ablation study was conducted in which the optimizations used in the proposed framework, i.e., in-mapper combiners, canonical labeling, and dynamic support thresholding, were disabled in turn to study the unintentional impact such optimizations have on overall performance. They ran the experiment over a synthetic 1M-edge dataset using a 4-node Hadoop cluster. The results of the ablation study are shown in Table 3.

Table 3: Ablation Study Results For The Proposed Framework

Optimization	Runtime (Seconds)	Percentage Increase
All Optimizations Enabled	1500	—
Without In-Mapper Combiners	1800	+20%
Without Canonical Labeling	1900	+26.67%
Without Dynamic Support Threshold	1650	+10%
Without All Optimizations	2400	+60%

It is an ablation study to measure the effect of each optimization introduced in the framework proposed here. This contained integral aspects, which included in-mapper combiners, canonical labeling, and dynamic support thresholding. It systematically turned off these components and measured their contribution to runtime efficiency and scalability. To ensure consistency in the testing environment, the evaluation in Section 5 is performed under a unified condition — with the 1M-edge synthetic dataset and in a 4-node Hadoop cluster. We found that the framework with all optimizations enabled ran on the dataset in about the 1500s. The runtimes increased by 20% to 1800 sec when in-mapper combiners were disabled. The jump further emphasizes reducing data shuffling between Mapper and Reducer. The intermediate data shuffling bottle-necked the execution without in-mapper combiners.

With canonical labeling disabled, it took 1900 seconds to complete, a 26.67% increase in runtime. Canonical labeling applies to eliminate redundant subgraph enumeration by combining isomorphic subgraphs as a single entity. This led to higher computational complexity in subgraph enumeration and isomorphism checks, as it increased the computational load due to the absence of this mechanism and, therefore, showed a notable performance drop. However, running the model without the dynamic support threshold took 1650 seconds (10% slower). To handle the enormous search space, dynamic thresholding pruning infrequent subgraphs early in the mining process. This optimization helps reduce the number of candidate subgraphs the framework needs to process, which otherwise would have slightly increased the framework's runtime. With optimizations turned off, the same framework took 2400 seconds, 60% longer than a fully optimized framework. The synergy of those elements is what matters for performance detection. Out of all the optimizations, canonical labeling and in-mapper combiners had the most significant impact, and dynamic support thresholding gave minuscule additional efficiency increases. This ablation study highlights how important these optimizations are in making the overall framework scalable and efficient in terms of execution time. Each of these components independently helps further enhance the overall performance and integrating these components ensures that the framework can perform efficiently on large-scale graph datasets.

5.5 Data Shuffling Overhead

The data shuffling overhead is one of the leading performance bottlenecks for distributed frameworks like Hadoop MapReduce. In frequent subgraph mining, Mapper tasks output intermediate data and transfer it to Reducer tasks for aggregation and subsequent processing. However, this communication, called data shuffling, is usually the bottleneck of distributed optimization, particularly for large-scale datasets. Too much data shuffling increases the amount of network traffic, and additional serialization, transfer, and deserialization operations also lengthen the time it takes to complete the run.

It introduces different optimizations for the proposed framework, like in-mapper combiners and canonical labeling, reducing the intermediate

data shuffling between the Mapper and Reducer phases. They seek to drastically reduce shuffling overhead by minimizing duplicate computations and optimizing intermediate data management. Analyses of the data volume at intermediate steps and its effects on runtime performance were then performed to quantify the effectiveness of these improvements under all the optimized configurations. As shown in the following results, each optimization reduces the shuffling overhead, so controlling the shuffling overhead is the key to the scalability and efficiency of such systems.

Table 4: Data Shuffling Overhead Results

Optimization Configuration	**Intermediate Data Shuffled (GB)**	**Increase in Data Shuffling (%)**
All Optimizations Enabled	1.2	0.00
Without In-Mapper Combiners	1.8	50.00
Without Canonical Labeling	1.6	33.33
Without Dynamic Support Threshold	1.4	16.67
Without All Optimizations	2.5	108.33

Table 4 shows the intermediate volume of data shuffled during the various optimization configurations. Without in-mapper combiners and without canonical labeling, we see a 50% and 33.33% difference, respectively, in terms of the data that needs to be shuffled. Without any of the optimizations, overhead is maximal, over 100% greater than in the fully optimized framework.

5.6 Comparative Analysis

In the comparative analysis part, we evaluate our proposed framework by comparing it with classical baselines and recent SOTA (state-of-the-art) models. This comparison illustrates improvements to the framework in terms of run-time efficiency, scalability, and data handling. It highlights the advantages of computational efficiency and distributed processing capabilities of the framework (playing against baselines such as gSpan and Apriori-based methods). Moreover, extensive experiments show that GEP outperforms the existing models (CloGraMi, G-thinker, FlexMiner, etc.) on both synthetic and real datasets. Thereafter, the results confirm the efficiency of the proposed framework for frequent subgraph mining tasks, underlining its effectiveness and applicability for real-world and distributed graph mining problems.

Table 5: Comparative Analysis Of Runtime, Scalability, And Data Shuffling Overhead

Method	**Runtime for 100K Edges (Seconds)**	**Runtime for 500K Edges (Seconds)**	**Runtime for 1M Edges (Seconds)**	**Scalability (Speedup for 1M Edges)**	**Data Shuffling Overhead (GB)**
Proposed Framework	450	2250	4500	2.0x	1.2
gSpan Baseline	600	3000	6000	1.0x	-
Apriori Baseline	750	3700	7400	1.1x	-
Non-Distributed Baseline	900	4500	9000	1.0x	-

Table 5 Comparison of the proposed framework with the commonly used baselines: gSpan, Apriori-based, and non-distributed methods This is in terms of runtime, scalability, and data

shuffling overhead of the given evaluation done over a range of data sizes (i.e., 100K, 500K, and 1M edges). The runtime results shown in the supplementary materials indicate that the proposed framework outperforms the baselines by a large margin. For the 100K-edge dataset, the runtime of the proposed framework is 450 seconds, while that of Spain, Apriori, and the non-distributed method are 600 seconds, 750 seconds, and 900 seconds, respectively. As the dataset size increases, this performance gap grows with the proposed framework running on 500K and 1M edges in 2250s and 4500s, respectively, versus 6000s for gSpan and 9000s for non-distributed methods for the 1M-edge dataset. These results demonstrate the computational efficiency of our proposed framework, which benefits from distributed processing and the reductions using in-mapper combiners and canonical labeling.

The proposed framework shows an obvious advantage in scalability, where on a 1M-edge dataset, the speedup is 2.0¡Á. This indicates that with more nodes in the Hadoop cluster, the framework is making better use of extra processing resources. When looking at the baselines, however, they are not sufficiently scalable, with gSpan and distributed methods stuck at 1.0x due to their in-memory, single-machine restrictions. Even though apriori-based methods show marginal gains (1.1 x), they still lag behind our proposed framework significantly. The proposed framework also excels in the other key metrics, like the data shuffling overhead. The proposed framework thus shows a significant reduction of network traffic in the MapReduce phases with minimal intermediate data volume of 1.2GB. They do not explicitly support the comparable data shuffling optimizations that ultimately translate into higher network overheads, which are not quantified here but implied by the higher runtimes of the baselines. The table highlights the advantages of the proposed framework in processing scale databases against the best in the table for frequent subgraph mining. Due to its capacity to reduce the time to execute, process at a larger scale, and lower the overhead associated with the levels of data shuffling, a powerful solution is provided, which outperforms the traditional baselines aside from depending on how distributed the environments are. These results confirm the beneficial optimizations and architectural choices in the framework proposed in this paper. A comparison of the proposed with state-of-the-art performance, as discovered in the literature, is presented in Table 6.

Table 6: Comparative Analysis Of The Proposed Framework With State-Of-The-Art Models

Model (Reference)	**Runtime (100K Edges)**	**Runtime (500K Edges)**	**Runtime (1M Edges)**	**Scalability (Speedup for 1M Edges)**	**Data Shuffling Overhead (GB)**
Proposed Framework	450	2250	4500	2.0x	1.2
CloGraMi (2021) [1]	520	2600	5100	1.8x	1.5
G-thinker (2020) [2]	600	3000	6000	1.5x	2.0
PEREGRINE (2020) [7]	580	2800	5500	1.7x	1.8
FlexMiner (2021) [12]	490	2400	4700	1.9x	1.3

The table below gives a detailed comparison of the proposed framework concerning a recent set of state-of-the-art models, CloGraMi(2021), G-thinker(2020), PEREGRINE(2020), and FlexMiner(2021). We test each model for runtime performance, scalability, and data shuffling overhead for three graph sizes of 100K, 500K, and 1M edges. SOTA model reference numbers are included to correlate with the literature review. The results also indicate that the proposed framework has a much better runtime efficiency than all SOTA models. In the case of the 100K-edge dataset, the proposed framework has the fastest run time of 450

seconds, while FlexMiner follows with a run time of 490 seconds. At 500K edges and 1M edges, the runtime of the proposed framework is still faster than all other comparisons, with runtimes of 2250 and 4500 seconds, respectively. On the other hand, the 1M-edge dataset, G-thinker, and PEREGRINE have runtimes of 6000 and 5500 seconds, respectively, which are much more significant. CloGraMi and FlexMiner are are also competitive but still underperforming than the proposed framework.

Scalability: The speedup obtained for HC-Exp for the 1M-edge dataset shows that the framework effectively takes advantage of the distributed computing resources in a Hadoop environment (the speedup reported is 2.0x). FlexMiner has the highest speed, 1.9x, among the SOTA models due to its efficient parallel processing design. CloGraMi and PEREGRINE outperform the others with speedups of 1.8x and 1.7x, respectively, as G-thinker trails behind at 1.5x, driven by its CPU-bound design. The analysis of the overhead of data shuffling highlights the proposed framework's efficiency, which is only 1.2GB of intermediate data for the 1M-edge dataset. This is the minimum number attained by all models and reflects the benefit of optimizations such as in-mapper combiners and canonical labeling (i.e., an automatic selection of the best-performing tag). In contrast, although FlexMiner is relatively efficient, it comes at a higher cost of 1.3GB overhead. Overheads of PEREGRINE are 1.8GB and 1.5GB for CloGraMi, while G-thinker has the highest overhead, 2.0GB, because it does not utilize optimized data handling mechanisms. The proposed framework yields better results than all the selected SOTA models, considering runtime efficiency, scalability, and data shuffling overhead. These results highlight the progress obtained through the proposed optimizations and a strong and effective solution for large-scale graph data with a high frequency of subgraph mining. This comparison recognizes the framework's efficiency and contributions to the existing challenges in distributed graph mining.

6. DISCUSSION

The research in this paper deals with some of the critical problems in frequent subgraph mining (FSM) that have broad applications in bioinformatics, chemoinformatics, and social network analysis. The traditional FSM methods, from Apriori-based methods to in-memory algorithms like gSpan, are faced with scalability and efficiency issues when applied to large-scale graph datasets. Nevertheless, the advances in iterative distributed frameworks (e.g., G-thinker) or pattern-aware systems (e.g., PEREGRINE) give rise to it with piecemeal enhancements. Instead, such methods still show considerable shortcomings, e.g., the high overhead of data shuffling, the ineffectiveness of candidate pruning, and limited scalability to coarsely sized datasets. To fill these gaps, this paper proposes a new framework based on MapReduce to discover frequent sequential patterns with more advanced optimizations. Such as in-mapper combiners to reduce intermediate data shuffling, canonical labeling to reduce duplicate subgraph enumeration, and dynamic support thresholds to improve pruning effectiveness. Our methodology is a significant advance over current methods as it employs distributed computing and these targeted optimizations to scale to much larger datasets.

Regarding our research objectives, the proposed FSM-MR framework has several advantages. It achieves this by harnessing MapReduce, which leads to an almost linear speedup with more computing resources for frequent subgraph mining, thereby motivating the scalability of subgraph mining under the MapReduce paradigm. Second, dynamic computation ineffectiveness is reduced from computation-intensive support to optimized equivalence classes, achieving 50% runtime support through in-mapper combiners and canonical labeling mechanisms. Yet, some restrictions are persisting. The framework is tailor-made for Hadoop-based environments, and its cross-compatibility with other distributed architectures is currently untested (e.g., Spark, GPU-based frameworks). Moreover, our dynamic approach of supporting thresholding (described in Section 3.5) favors pruning efficiency. Still, individual configurations may alter based on dataset characteristics, particularly for dense graphs where enumerating subgraphs could be costly. Further advancements to address these issues will enhance the applicability and robustness of FSM-MR in various real-world contexts.

Further, our findings in Table 1 show that the proposed framework outperforms traditional baselines and SoTA regarding runtime efficiency, scalability, and data processing capabilities. The proposed framework delivers a reduction in runtime (50%) along with linear

scalability for more cluster nodes. These results demonstrate the necessity of the optimizations proposed, in particular in alleviating the computational bottleneck entailed by the subgraph isomorphism checks and headstrong scaling large-scale candidate generation. By closing the scalability and efficiency gaps in FSM, this research makes FSM practical in that large-scale graph analysis is required. In addition, the integration of MapReduce and FSM-focused optimizations paves the way for future distributed mining frameworks. The implications of this work are significant, addressing some of the most pressing limitations in the state-of-the-art Section 6.1. Still, there are limitations in the proposed methodology that suggest directions for future research.

6.1 Limitations of the Current Study

Although we have achieved considerable gains in scalability and efficiency using our proposed FSM-MR framework, we must acknowledge threats to validity in our evaluation. One major limitation is that they depend very much on the dataset's properties — if the graphics are highly dense, there may be extra computational overhead, leading to lower performance. Moreover, although it has also been benchmarked against state-of-the-art baselines using well-established datasets, the generalizability of our results to other real-world graph structures is an open question, such as dynamic or evolving networks that have different characteristics. For the selection of critique criteria, we have clarified our rationale regarding runtime efficiency, scalability, and data shuffling overhead, as these are cornerstones of challenges about distributed FSM. In line with existing works in this domain, we adopt a rigorous methodology in our evaluation, providing a fair and extensive comparison. However, future investigations could involve additional performance metrics like memory usage and adaptability to heterogeneous computing environments to enhance our framework's robustness further.

This study has certain limitations. First, the proposed framework's performance heavily depends on the dataset's structure; highly dense graphs may increase computational overhead. Second, while the framework demonstrates scalability, it is optimized for Hadoop-based environments and may require adaptations for alternative distributed systems like Apache Spark. Third, subgraph isomorphism checks, though optimized with canonical labeling, remain computationally intensive for extremely large or complex subgraphs, which may impact runtime in such cases. Addressing these limitations through further optimization and cross-platform adaptability will enhance the framework's applicability and robustness in diverse real-world scenarios.

7. CONCLUSION AND FUTURE WORK

This research introduced Frequent Subgraph Mining Using MapReduce (FSM-MR), a novel algorithm and framework designed to overcome the scalability and efficiency limitations of traditional and state-of-the-art frequent subgraph mining techniques. The proposed FSM-MR algorithm integrates key optimizations such as in-mapper combiners, canonical labeling, and dynamic support thresholds, significantly improving subgraph enumeration, pruning, and overall runtime efficiency. Leveraging the MapReduce paradigm, the framework achieved up to 50% runtime reductions and near-linear scalability across large-scale graph datasets, outperforming baselines and recent state-of-the-art methods. While the proposed FSM-MR algorithm demonstrates substantial improvements, certain limitations were identified. These include dependency on dataset structure, computational overhead for dense graphs, and the framework's optimization specific to Hadoop-based systems. In this context, our FSM-MR framework establishes a new strategy by integrating in-mapper combiners, canonical labeling, and dynamic support thresholding to the MapReduce configuration, marking a notable advancement in scalability and efficiency for frequent subgraph mining. Overall, such optimizations can decrease the computational overheads and data shuffling time, making our method more scalable over existing state-of-the-art work while remaining potentially adaptable to large-scale graph data.

Future research will focus on adapting FSM-MR for alternative distributed platforms like Apache Spark and improving its efficiency for dense and highly complex graphs. Additionally, extending FSM-MR for GPU-based accelerations and dynamic graph mining will further enhance its capabilities. By addressing these limitations and improving the FSM-MR algorithm, this research lays a strong foundation for scalable, efficient subgraph mining solutions, supporting advanced

applications in bioinformatics, chemoinformatics, and social network analysis while paving the way for future innovations.

REFERENCES

[1] LAM B. Q. NGUYEN, LOAN T. T. NGUYEN, IVAN ZELINKA, VACLAV SNASEL, HUNG SON NGUYEN, AND BAY VO. (2021). A method for closed frequent subgraph mining in a single large graph. *IEEE*. 9, pp.165719 - 165733. http://DOI:10.1109/ACCESS.2021.3133666

[2] Yan, D., Guo, G., Rahman Chowdhury, M. M., Tamer Ozsu, M., Ku, W.-S., & Lui, J. C. S. (2020). G-thinker: A Distributed Framework for Mining Subgraphs in a Big Graph. 2020 IEEE 36th International Conference on Data Engineering (ICDE). doi:10.1109/icde48307.2020.00122

[3] Jazayeri, A., & Yang, C. (2021). Frequent Subgraph Mining Algorithms in Static and Temporal Graph-Transaction Settings: A Survey. IEEE Transactions on Big Data, 1–1. doi:10.1109/tbdata.2021.3072001

[4] Saif Ur Rehman, Kexing Liu, Tariq Ali, Asif Nawaz, and Simon James Fong. (2021). A graph mining approach for ranking and discovering the interesting frequent subgraph patterns. *Springer*. 14(152), pp.1-17. https://doi.org/10.1007/s44196-021-00001-4

[5] Nguyen, L. B. Q., Vo, B., Le, N.-T., Snasel, V., & Zelinka, I. (2020). Fast and scalable algorithms for mining subgraphs in a single large graph. Engineering Applications of Artificial Intelligence, 90, 103539. doi:10.1016/j.engappai.2020.103539

[6] Yan, D., Qu, W., Guo, G., & Wang, X. (2020). PrefixFPM: A Parallel Framework for General-Purpose Frequent Pattern Mining. 2020 IEEE 36th International Conference on Data Engineering (ICDE). doi:10.1109/icde48307.2020.00208

[7] Jamshidi, K., Mahadasa, R., & Vora, K. (2020). Peregrine. Proceedings of the Fifteenth European Conference on Computer Systems. doi:10.1145/3342195.3387548

[8] Ribeiro, P., Paredes, P., Silva, M. E. P., Aparicio, D., & Silva, F. (2021). A Survey on Subgraph Counting. ACM Computing Surveys, 54(2), 1–36. doi:10.1145/3433652

[9] Chen, Dathathri, R., Gill, G., Hoang, L., & Pingali, K. (2021). Sandslash. Proceedings of the ACM International Conference on Supercomputing. https://doi.org/10.1145/3447818.3460359

[10] Bindschaedler, L., Malicevic, J., Lepers, B., Goel, A., & Zwaenepoel, W. (2021). Tesseract. Proceedings of the Sixteenth European Conference on Computer Systems. doi:10.1145/3447786.3456253

[11] Yoo, J. S., Boulware, D., & Kimmey, D. (2019). Parallel co-location mining with MapReduce and NoSQL systems. Knowledge and Information Systems, 62(4), 1433–1463. doi:10.1007/s10115-019-01381-y

[12] Chen, X., Huang, T., Xu, S., Bourgeat, T., Chung, C., & Arvind, A. (2021). FlexMiner: A Pattern-Aware Accelerator for Graph Pattern Mining. 2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA). https://doi.org/10.1109/isca52012.2021.00052

[13] Shukla, M., Dharme, D., Ramnarain, P., Santos, R. D., & Lu, C.-T. (2020). DIG DUG: Scalable Separable Dense Graph Pruning and Join Operations in MapReduce. IEEE Transactions on Big Data, 1–1. doi:10.1109/tbdata.2020.2983650

[14] Wang, Z., Hu, W., Chen, G., Yuan, C., Gu, R., & Huang, Y. (2021). Towards Efficient Distributed Subgraph Enumeration Via Backtracking-Based Framework. IEEE Transactions on Parallel and Distributed Systems, 32(12), 2953–2969. doi:10.1109/tpds.2021.3076246

[15] Naik, D., Behera, R. K., Ramesh, D., & Rath, S. K. (2020). Map-Reduce-Based Centrality Detection in Social Networks: An Algorithmic Approach. Arabian Journal for Science and Engineering. doi:10.1007/s13369-020-04636-x

[16] Khader, M., & Al-Naymat, G. (2020). Density-based Algorithms for Big Data Clustering Using MapReduce Framework. ACM Computing Surveys, 53(5), 1–38. doi:10.1145/3403951

[17] Zhao, C., Zhang, Z., Xu, P., Zheng, T., & Guo, J. (2020). Kaleido: An Efficient Out-of-core Graph Mining System on A Single Machine. 2020 IEEE 36th International Conference on Data Engineering (ICDE). doi:10.1109/icde48307.2020.00064

[18] Preti, G., Lissandrini, M., Mottin, D., & Velegrakis, Y. (2019). Mining patterns in graphs with multiple weights. Distributed

and Parallel Databases. doi:10.1007/s10619-019-07259-w

[19] Asma Belhadi, Youcef Djenouri, Jerry Chun-Wei Lin & Alberto Cano. (2020). A general-purpose distributed pattern mining system. *Springer*., pp.1-16. https://doi.org/10.1007/s10489-020-01664-w

[20] Jiezhong He, Yixin Chen, Zhouyang Liu, and Dongsheng Li. (2024). Optimizing subgraph retrieval and matching with an efficient indexing scheme. *Springer*., pp.1-30. https://doi.org/10.21203/rs.3.rs-4209309/v1

[21] Akshit Sharma, Dinesh Mehta, and Bo Wu. (2024). Understanding High-Performance Subgraph Pattern Matching: A Systems Perspective. *ACM*, pp.1-12. https://doi.org/10.1145/3661304.3661897

[22] LINGKAI MENG, YU SHAO, LONG YUAN, LONGBIN LAI, PENG CHENG, XUE LI, WENYUAN YU, WENJIE ZHANG, XUEMIN LIN, and JINGREN ZHOU. (2024). A survey of distributed graph algorithms on massive graphs. *ACM*. 57(2), pp.1-39. https://doi.org/10.1145/3694966

[23] Tommaso Lanciano, Atsushi Miyauchi, Adriano Fazzone, and Francesco Bonchi. (2024). A survey on the densest subgraph problem and its variants. *ACM*. 56(8), pp.1-44. https://doi.org/10.1145/3653298

[24] MUHAMMAD SADEEQULLAH, AZHAR RAUF, SAIF UR REHMAN, AND NOHA ALNAZZAWI. (2024). Probabilistic Support Prediction: Fast frequent itemset mining in dense data. *IEEE*. 12, pp.39330 - 39350. http://DOI:10.1109/ACCESS.2024.3376477

[25] Edelmira Pasarella, Maria-Esther Vidal, Cristina Zoltan, and Juan Pablo Royo Sal. (2024). A computational framework based on the dynamic pipeline approach. *Elsevier*. 139(.), pp.1-21. [Online]. Available at: https://doi.org/10.1016/j.jlamp.2024.100966

[26] Junjie Liu, Rongxin Jiang, Xuesong Liu,Fan Zhou, Yaowu Chen, and Chen Shen. (2024). Large-Scale Clustering on 100 M-Scale Datasets Using a Single T4 GPU via Recall KNN and Subgraph Segmentation. *Springer*. 56(34), pp.1-23. https://doi.org/10.1007/s11063-024-11444-z

[27] Federico Brunero and Petros Elia. (2022). Multi-access distributed computing. *IEEE*. 70(5), pp.3385 - 3398. http://DOI:10.1109/TIT.2024.3373128

[28] DEVENDRA DAHIPHALE. (2023). Mapreduce for graphs processing: New big data algorithm for 2-edge connected components and future ideas. *IEEE*. 11, pp.54986 - 55001. http://DOI:10.1109/ACCESS.2023.3281266

[29] Shubhangi Chaturvedi, Sri Khetwat Saritha, and Animesh Chaturvedi. (2023). Spark-based Parallel Frequent Pattern Rules for Social Media Data Analytics. *IEEE*, pp.1-8. http://DOI:10.1109/CCGridW59191.2023.00039

[30] Yanyan Song, Yuzhou Qin, Wenqi Hao, Pengkai Liu, Jianxin Li, Farhana Murtaza Choudhury, Xin Wang, and Qingpeng Zhan. (2023). Optimizing subgraph matching over distributed knowledge graphs using partial evaluation. *Springer*. 26, p.751–771. https://doi.org/10.1007/s11280-022-01075-6

[31] Changxi Ma, Mingxi Zhao, and Yongpeng Zhao. (2023). An overview of Hadoop applications in transportation big data. *Elsevier*. 10(5), pp.900-917. https://doi.org/10.1016/j.jtte.2023.05.003

[32] BO YAN, CHENG YANG, CHUAN SHI, YONG FANG, QI LI, YANFANG YE, and JUNPING DU. (2023). Graph mining for cybersecurity: A survey. *ACM*. 18(2), pp.1-50. https://doi.org/10.1145/3610228

[33] Makhan Kumbhkar, Pranjal Shukla, and Yashwardhan Singh. (2023). Dimensional Reduction Method based on Big Data Techniques for Large Scale Data. *IEEE*, pp.1-7. http://DOI:10.1109/ICICACS57338.2023.10100261

[34] Harif Asma,Namir Abdelwahid, and Marzak Abdelaziz. (2023). Approach to reduce the communication cost when partitioning a big graph. *Elsevier*. 220, pp.1051-1056. https://doi.org/10.1016/j.procs.2023.03.147

[35] Ziwei Mo, Qi Luo , Dongxiao Yu, Hao Sheng, Jiguo Yu, and Xiuzhen Cheng. (2023). Distributed truss computation in dynamic graphs. *IEEE*. 28(5), p.873–887. http://DOI:10.26599/TST.2022.9010019

[36] A. Srinivas Reddy, P. Krishna Reddy, Anirban Mondal, and U. Deva Priyakumar. (2022). Mining subgraph coverage patterns from graph transactions. *Springer*. 13, p.105–121. https://doi.org/10.1007/s41060-021-00292-y

[37] Xiaozhou Liu, Yudi Santoso, Venkatesh Srinivasan, Alex Thomo. (2022). Practical Survey on MapReduce Subgraph Enumeration Algorithms. *Springer*, pp.1-14.

[38] ANDREA PASINI, FLAVIO GIOBERGIA , ELIANA PASTOR, AND ELENA BARALIS. (2022). Semantic image collection summarization with frequent subgraph mining. *IEEE*. 10, pp.131747 - 131764. http://DOI:10.1109/ACCESS.2022.3229654

[39] TEWODROS ALEMU AYALL, HUAWEN LIU, CHANGJUN ZHOU, ABEGAZ MOHAMMED SEID, FANTAHUN BOGALE GEREME, HAYLA NAHOM ABISHU, AND YASIN HABTAMU YACOB. (2022). Graph computing systems and partitioning techniques: A survey. *IEEE*. 10, pp.118523 - 118550. http://DOI:10.1109/ACCESS.2022.3219422

[40] HANLIN ZHANG, LINLIN DING, GANG ZHANG, YISHAN PAN, AND BAOYAN SONG. (2022). An Efficient Vertex-Driven Temporal Graph Model and Subgraph Clustering Method. *IEEE*. 10, pp.100627 - 100645. http://DOI:10.1109/ACCESS.2022.3208360

[41] National Center for Biotechnology Information (NCBI), 2024. *PubChem BioAssay Database*. [online] Available at: https://pubchem.ncbi.nlm.nih.gov/

[42] Srikanth, G., Raghavendran, C.V., Prabhu, M.R., Radha, M., Kumari, N.V.S. & Francis, S.K., 2025. Climate Change Impact on Geographical Region and Healthcare Analysis Using Deep Learning Algorithms. Remote Sensing in Earth Systems Sciences, 130359.

[43] Ravikumar Ch1, Marepalli Radha2, Maragoni Mahendar3, Pinnapureddy Manasa"A comparative analysis for deep-learning-based approaches for image forgery detection International journalof Systematic Innovation https://doi.org/10.6977/IJoSI.202403_8(1).0001

[44] Anitha Patil. (2019). Distributed Programming Frameworks in Cloud Platforms. International Journal of Recent Technology and Engineering (IJRTE). 7(6), pp.611-619.

[45] A. Patil and S. Govindaraj, "An AI Enabled Framework for MRI-based Data Analytics for Efficient Brain Stroke Detection," 2023 International Conference on Advances in Computing, Communication and Applied Informatics (ACCAI), Chennai, India, 2023, pp. 1-7, doi: 10.1109/ACCAI58221.2023.10201136.

[46] Sreedhar Bhukya, A Novel Methodology for Secure De duplication of Imagedata in Cloud Computing using Compressive Sensing and Random Pixel Exchanging, Journal of Theoretical and Applied Information Technology (JATIT), Vol.102. No 4, ISSN: 1992-8645 (2024), SCOPUS.

[47] Sreedhar Bhukya, Multiclass Supervised Learning Approach for SAR-COV2 Severity and Scope Prediction: SC2SSP Framework, Volume (12) issue (1), 2025-01-31, SCOPUS.