Scalable Frequent Subgraph Mining

Dissertation by
Ehab Abdelhamid Mohamed Abdelhamid

In Partial Fulfillment of the Requirements

For the Degree of

Doctor of Philosophy

King Abdullah University of Science and Technology

Thuwal, Kingdom of Saudi Arabia

June, 2017
The dissertation of Ehab Abdelhamid Mohamed Abdelhamid is approved by the examination committee.

Committee Chairperson: Panos Kalnis - Professor
Committee Members: Diogo Gomes - Professor, Xin Gao - Associate Professor, Mohammed J. Zaki - Professor (external)
ABSTRACT

A graph is a data structure that contains a set of nodes and a set of edges connecting these nodes. Nodes represent objects while edges model relationships among these objects. Graphs are used in various domains due to their ability to model complex relations among several objects.

Given an input graph, the Frequent Subgraph Mining (FSM) task finds all subgraphs with frequencies exceeding a given threshold. FSM is crucial for graph analysis, and it is an essential building block in a variety of applications, such as graph clustering and indexing. FSM is computationally expensive, and its existing solutions are extremely slow. Consequently, these solutions are incapable of mining modern large graphs. This slowness is caused by the underlying approaches of these solutions which require finding and storing an excessive amount of subgraph matches.

This dissertation proposes a scalable solution for FSM that avoids the limitations of previous work. This solution is composed of four components. The first component is a single-threaded technique which, for each candidate subgraph, needs to find only a minimal number of matches. The second component is a scalable parallel FSM technique that utilizes a novel two-phase approach. The first phase quickly builds an approximate search space, which is then used by the second phase to optimize and balance the workload of the FSM task. The third component focuses on accelerating frequency evaluation, which is a critical step in FSM. To do so, a machine learning model is employed to predict the type of each graph node, and accordingly, an optimized method is selected to evaluate that node. The fourth component focuses on mining dynamic graphs, such as social networks. To this end, an incremental index is maintained during the dynamic updates. Only this index is processed and updated for the majority of graph updates. Consequently, search space is
significantly pruned and efficiency is improved.

The empirical evaluation shows that the proposed components significantly outperform existing solutions, scale to a large number of processors and process graphs that previous techniques cannot handle, such as large and dynamic graphs.
I would like to express my special thanks to my advisor Professor Panos Kalnis for the continuous support of my Ph.D. study. I would also like to thank my committee members, Professor Mohammed Zaki, Professor Diogo Gomes and professor Xin Gao for their time serving as committee members and for their insightful comments. I thank all my friends, especially the Infocoulders, for the fun and fruitful time I spent with them. I would also like to thank KAUST for giving me the opportunity to be part of an engaging and stimulating research environment.

Special thanks to my family. Words cannot express how grateful I am to my father and mother for all of the sacrifices that you have made on my behalf. Also to my small family, my wife Shimaa, for her time and continuous support, thanks for everything you have done for me. Of course, I thank my little two kids, Omar and Yasin, for keeping me young, active and for the great moments full of fun.
# TABLE OF CONTENTS

Examination Committee Page 2  
Copyright 3  
Abstract 4  
Acknowledgements 6  
List of Figures 10  
List of Tables 13  
List of Algorithms 14  

1 Introduction 15  
1.1 FSM Applications 17  
1.2 Existing Methods and Limitations 18  
1.3 Contributions 21  
1.3.1 GRAMi: Fast Single-Threaded FSM 21  
1.3.2 ScaleMine: Parallel FSM 22  
1.3.3 PSI: Improved Frequency Evaluation 23  
1.3.4 IncGM+: Incremental FSM 24  
1.4 Outline 25  

2 Background and Related Work 26  
2.1 Background 26  
2.1.1 Definitions 26  
2.1.2 FSM Algorithm 28  
2.1.3 MNI-based Frequency Computation 29  
2.2 Related Work 31  
2.2.1 Exact FSM 31  
2.2.2 Approximate FSM 34  
2.2.3 Parallel FSM 35
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.2.4</td>
<td>Dynamic graph mining</td>
<td>37</td>
</tr>
<tr>
<td>2.2.5</td>
<td>Frequent Itemset Mining Over a Stream</td>
<td>38</td>
</tr>
<tr>
<td>2.2.6</td>
<td>Subgraph isomorphism</td>
<td>39</td>
</tr>
<tr>
<td>2.2.7</td>
<td>Graph Indexing</td>
<td>41</td>
</tr>
<tr>
<td>2.2.8</td>
<td>Graphlets Mining</td>
<td>43</td>
</tr>
<tr>
<td>3</td>
<td><strong>GRAMI: Single-Threaded Frequent Subgraph and Pattern Mining</strong></td>
<td>45</td>
</tr>
<tr>
<td>3.1</td>
<td>Introduction</td>
<td>45</td>
</tr>
<tr>
<td>3.2</td>
<td>Preliminaries</td>
<td>49</td>
</tr>
<tr>
<td>3.3</td>
<td>The GRAMI Approach</td>
<td>51</td>
</tr>
<tr>
<td>3.3.1</td>
<td>The CSP Model</td>
<td>51</td>
</tr>
<tr>
<td>3.3.2</td>
<td>Frequent Subgraph Mining</td>
<td>53</td>
</tr>
<tr>
<td>3.3.3</td>
<td>Optimizing Frequency Evaluation</td>
<td>56</td>
</tr>
<tr>
<td>3.4</td>
<td>GRAMI Extensions</td>
<td>65</td>
</tr>
<tr>
<td>3.5</td>
<td>Experimental Evaluation</td>
<td>67</td>
</tr>
<tr>
<td>3.6</td>
<td>Conclusion</td>
<td>77</td>
</tr>
<tr>
<td>4</td>
<td><strong>ScaleMine: Scalable Parallel Frequent Subgraph Mining</strong></td>
<td>78</td>
</tr>
<tr>
<td>4.1</td>
<td>Introduction</td>
<td>78</td>
</tr>
<tr>
<td>4.2</td>
<td>Approximate Phase</td>
<td>82</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Sampling-based Subgraph Evaluation</td>
<td>83</td>
</tr>
<tr>
<td>4.2.2</td>
<td>Search Space Estimation</td>
<td>86</td>
</tr>
<tr>
<td>4.3</td>
<td>Exact Phase</td>
<td>87</td>
</tr>
<tr>
<td>4.3.1</td>
<td>System Description</td>
<td>87</td>
</tr>
<tr>
<td>4.3.2</td>
<td>Subtasking</td>
<td>89</td>
</tr>
<tr>
<td>4.3.3</td>
<td>Optimizations</td>
<td>90</td>
</tr>
<tr>
<td>4.3.4</td>
<td>Correctness</td>
<td>92</td>
</tr>
<tr>
<td>4.4</td>
<td>Experimental Evaluation</td>
<td>93</td>
</tr>
<tr>
<td>4.4.1</td>
<td>Experimental setup</td>
<td>93</td>
</tr>
<tr>
<td>4.4.2</td>
<td>Comparison with Existing Systems</td>
<td>94</td>
</tr>
<tr>
<td>4.4.3</td>
<td>Optimizations</td>
<td>95</td>
</tr>
<tr>
<td>4.4.4</td>
<td>Approximation Phase Performance</td>
<td>97</td>
</tr>
<tr>
<td>4.4.5</td>
<td>Scalability</td>
<td>98</td>
</tr>
<tr>
<td>4.5</td>
<td>Conclusion</td>
<td>100</td>
</tr>
<tr>
<td>5</td>
<td><strong>Pivoted Subgraph Isomorphism</strong></td>
<td>101</td>
</tr>
<tr>
<td>5.1</td>
<td>Introduction</td>
<td>101</td>
</tr>
</tbody>
</table>
5.2 Preliminaries .................................................. 104
5.3 Pivoted Subgraph Isomorphism ................................. 108
  5.3.1 System Architecture .................................... 108
  5.3.2 Data Pre-processing ..................................... 109
  5.3.3 Training .................................................. 110
  5.3.4 Model-Based Pivoted Subgraph Isomorphism ............. 111
5.4 Two PSI Methods ............................................... 111
  5.4.1 Pessimistic Pivoted Subgraph Isomorphism ............... 112
  5.4.2 Optimistic Pivoted Subgraph Isomorphism ............... 113
5.5 Evaluation ..................................................... 114
  5.5.1 Experimental setup ..................................... 114
  5.5.2 Comparison with Existing Systems ...................... 116
  5.5.3 Optimizations ........................................... 118
  5.5.4 Applying PSI to FSM .................................... 121
5.6 Conclusion .................................................... 121

6 Incremental Frequent Subgraph Mining on Dynamic Graphs 122
  6.1 Introduction ................................................ 122
  6.2 Preliminaries ............................................... 124
  6.3 Incremental Graph Mining ................................... 127
    6.3.1 Search Space Pruning .................................. 128
    6.3.2 Embeddings-based optimization ....................... 130
    6.3.3 Fast Embeddings Lookup Store (FELS) ................. 134
    6.3.4 Reordering ............................................ 136
    6.3.5 Batching ............................................... 137
  6.4 Experimental Evaluation .................................. 139
    6.4.1 Efficiency ............................................. 141
    6.4.2 Memory overhead ...................................... 142
    6.4.3 Batching ............................................... 144
  6.5 Conclusions ............................................... 146

7 Concluding Remarks ............................................. 147
  7.1 Summary of Contributions ................................ 147
  7.2 Future Research Directions ................................. 148

References ...................................................... 150
LIST OF FIGURES

1.1 (a) An input graph $G$. (b) a candidate subgraph $S$. .......................... 16

2.1 A collaboration graph $G$ and a subgraph $S$; nodes correspond to authors
(labeled with their field of interest) and edges represent co-authorship (la-
beled with the number of co-authored papers). ............................... 27

2.2 FSM search space ............................................................. 29

3.1 (a) A collaboration graph $G$; nodes correspond to authors (labeled with
their field of work) and edges represent co-authorship (labeled with number
of co-authored papers).  (b) and (c) Subgraphs $S_1$ and $S_2$. ............ 46

3.2 (a) The distance $\Delta_p$ for the graph $G$ of Figure 3.1. (b) A pattern $P_1$. .... 50

3.3 (a) Construction of the subgraph tree.  (b) Variables and domains of the
corresponding subtrees. Marked nodes represent the pruned assignments
which are pushed down the tree. .................................................. 59

3.4 Automorphisms.  (a) Input graph $G$. (b) Subgraph $S$ and its valid assignments. 62

3.5 (a) Subgraph $S$ is generated by extending $S'$ with edge $C - K$.  (b) $S$ is
decomposed into overlapping subgraphs $S_1$ to $S_3$ containing the newly ex-
tended edge $C - K$. ................................................................. 63

3.6 Performance of GRAMI and GROWSTORE ............................... 70

3.7 (a) Memory requirements for GRAMI and GROWSTORE and (b) Using
MIS metric .............................................................................. 71

3.8 Performance evaluation for mining frequent patterns in CiteSeer dataset
comparing between GROWSTORE and GRAMI($\delta$) where $\delta$ is the distance
threshold .............................................................. 72

3.9 Comparing (a,c) the minimum support threshold and (b,d) the maximum
number of frequent patterns that can be achieved within an allotted time
budget.  For (a,b) we used GRAMI($\delta$) and for (c,d) we use CGRAMI($\delta$)
constrained to reject patterns with more than four nodes with the same label 73

3.10 Performance evaluation of GRAMI and AGRAMI with different values for
the approximation parameter. .................................................. 75
3.11 The effect of optimizations. No opt: Algorithm ISFREQUENTHEURISTIC (Section 3.3.2). Lazy: Lazy search and decomposition optimizations enabled. Pruning: Only pruning push-down optimization enabled. Unique: Only unique labels optimization enabled. GRAMI: All optimization enabled (Algorithm ISFREQUENT). ........................................ 76

3.12 Performance comparison between GRAMI and GGQL; a modified version of GRAMI that replaces ISFREQUENT with a counting function based on GraphQL .................................................. 77

4.1 Strong scalability of Baseline and TaskDivision. Total response time in seconds. Twitter dataset; \( \tau = 160K \) (see Section 4.4 for details) ........................................ 80

4.2 The distribution of means of the samples ........................................ 84

4.3 ScaleMine System Architecture ........................................ 88

4.4 Performance of ScaleMine vs. existing FSM systems on a cluster of 16 machines (256 workers) using two datasets: (a) Patents and (b) Twitter ........ 95

4.5 Effect of ScaleMine’s optimizations using 512 cores on both Twitter (\( \tau = 155k \)) and Weibo (\( \tau = 490k \), maximum size = five edges) ........................................ 96

4.6 Approximation phase accuracy for Patents and Twitter datasets. The used support thresholds are the same thresholds used in Figure 4.4 .................. 97

4.7 Approximation phase time w.r.t the exact time .................. 98

4.8 Scalability and speedup efficiency of ScaleMine using different real datasets: Patents with \( \tau = 15k \), Twitter with \( \tau = 155k \), Weibo with \( \tau = 460k \) and maximum subgraph size = five edges and MicoBig with \( \tau = 8m \) and maximum subgraph size = five edges ................................. 99

5.1 (a) Input graph \( G \) (b) A Subgraph \( S \) ........................................ 102

5.2 PSI Architecture ........................................ 109

5.3 Query Performance of PSI vs. state-of-the-art Subgraph Isomorphism Systems ........................................ 117

5.4 Effect of PSI’s Optimizations using Twitter and Weibo datasets ........... 118

5.5 Prediction Accuracy of PSI ........................................ 119

5.6 Model training and prediction time vs. the total time using different real datasets ........................................ 120

6.1 FSM search space starting from small subgraphs at the bottom to larger subgraphs towards the top ........................................ 126
6.2 (a) Evolving graph $G$ at different time points (b) Subgraph $S_1$ (c) Subgraph $S_2$ .................................................. 127
6.3 (a) Input graph $G$ (b) A Subgraph $S$ (c) the MNI table of $S$ embeddings in $G$ when $\tau = 3$ ........................................ 131
6.4 Fast Embeddings Lookup Store (FELS). (a) Embeddings inverted index and (b) MNI table ...................................................... 134
6.5 Performance comparison: Naive, MomentFSM, IncGM, and IncGM+. Time represents the graph loading time plus the time required for processing all graph updates. ...................................................... 141
6.6 The fringe size: the number of materialized subgraphs in $MFS$ and $MIFS$. ................................. 143
6.7 Memory overhead. ...................................................... 144
6.8 Twitter batching. Time represents the time required to process the given workload using different batch sizes compared to edge by edge updates. .................. 144
6.9 Effect of different batch size. Time represents the time required for processing all update batches. ...................................................... 145
**LIST OF TABLES**

3.1 Definitions of the anti-monotonic structural constraints for pattern $P$, implemented in CGRAMI ............................................. 66
3.2 Definitions of the anti-monotonic semantic constraints for pattern $P$, implemented in CGRAMI ............................................. 66
3.3 Datasets and their characteristics ........................................... 68

4.1 Existing parallel systems and the largest graph used in their evaluations (ScaleMine is the proposed system). .................................. 79
4.2 Datasets and their characteristics ........................................... 94

5.1 Datasets and their characteristics ........................................... 115
5.2 Number of Subgraph Query Matches in Millions ................. 116
5.3 Comparison between ScaleMine and ScaleMine+ .................. 121

6.1 Datasets and their characteristics ........................................... 140
6.2 Comparing the performance of the different batching optimization techniques ......................................................... 146
LIST OF ALGORITHMS

1. Frequent Subgraph Mining ........................................... 28
2. CSP-based Frequent Subgraph Mining ............................... 53
3. Subgraph Extension ....................................................... 54
4. Subgraph Evaluation based on CSP ................................. 55
5. Heuristic-based frequent Subgraph Mining ....................... 57
6. Optimized Frequent Subgraph mining ............................... 58
7. Sampling-Based Subgraph Evaluation ............................... 85
8. Optimized Pivoted Subgraph Isomorphism ......................... 112
9. Optimistic Pivoted Subgraph Isomorphism ....................... 114
10. Fringe-Based Incremental Frequent Subgraph Mining .......... 129
11. Optimized Incremental Frequent Subgraph Mining Based on Storing Minimal Number of Embeddings ............. 132
Chapter 1

Introduction

Graphs model complex relations among objects in a variety of application domains such as chemistry, bioinformatics, computer vision, social networks, text retrieval and web analysis. For example, a social network can be represented as a graph, where nodes represent users and edges represent relationships among these users. Similarly, chemical compounds, protein-protein-interaction networks, communication networks and the web, can be modeled as graphs. Modern applications require analysis, mining, and indexing of these graphs, which are usually very large. For example, Facebook has 1.86 billion monthly active users [1]. Another example is the world wide web, where there exist almost 1.8 billion websites [2]. Processing such huge graphs is not trivial even for simple tasks [3].

One of the important operations on graphs is Frequent Subgraph Mining (FSM). Given an input graph $G$ and a user-defined frequency threshold $\tau$, the goal of FSM is to find all subgraphs in $G$ that have frequency larger than or equal to $\tau$. The frequency of a subgraph $S$ depends on the occurrences of $S$ in the input graph $G$. There are two settings for FSM; The first is the transactional setting, where mining is conducted on a database of many, relatively small graphs, each graph is a transaction. The other setting, which is the focus of this thesis, is the single graph setting, where mining is conducted on a single large graph.

In the transactional setting, the frequency of a subgraph $S$ is calculated as the number of input graphs that have at least one occurrence of $S$. A subgraph is considered frequent if it exists in at least $\tau$ graphs. This frequency computation metric is anti-monotonic in the transactional setting. An anti-monotonic metric guarantees that if $S_1$ is a subgraph of $S_2$ then the frequency of $S_1$ is always larger than or equal to the frequency of $S_2$. 
The single graph setting is a general case of the transactional one since a set of small graphs can be considered as connected components within a single large graph. Moreover, discovering frequent subgraphs in a single graph is more complicated because multiple occurrences of identical subgraphs may overlap, which complicates the frequency computation task. More importantly, this setting is more computationally demanding since the complexity of finding subgraph occurrences is exponential in the input graph size. For the single graph setting, using the same metric used in the transactional case is meaningless since the input dataset is one graph. Using another simple metric like counting the number of occurrences is not anti-monotone since there are cases where a subgraph has less number of occurrences than its extended supergraph. For instance, Figure 1.1 shows an example of an input graph $G$ and a candidate subgraph $S$. The number of occurrences of $S = \{v_1, v_2\}$ is six and they are: $\{(u_{21}, u_{19}), (u_{22}, u_{19}), (u_1, u_2), (u_{23}, u_{18}), (u_{17}, u_{18}), (u_{17}, u_{16})\}$, whereas the number of occurrences of its subgraph with a single node labeled with 'B' is only four: $\{(u_{19}), (u_2), (u_{18}), (u_{16})\}$. This difference in the number of occurrences contradicts with the anti-monotone property. As such, this metric is not anti-monotonic. Having an anti-monotone frequency metric is crucial since it allows the development of methods that can
effectively prune the search space. Without an anti-monotone metric, exhaustive search is unavoidable [4, 5]. The literature defines several anti-monotone frequency metrics for the single graph case, such as minimum image based (MNI) [4], harmful overlap (HO) [5], and maximum independent sets (MIS) [6]. Out of these metrics, MNI is the most efficient since the computation of MIS and HO are NP-complete, while the computation is linear for MNI. More details on MNI will be given in Section 2.1.3.

1.1 FSM Applications

FSM is a well known operation and it is widely applied in many applications, such as graph analysis, security, privacy and data mining. For example, in bioinformatics, the discovery of frequent structured patterns is essential to analyze and understand the biological data [7]. Also, in the social networks domain, FSM is used to predict the relationship types among users [8]. These predictions are then used to model users’ behavior and provide recommendations tailored to their needs. FSM is also an essential building block of many applications in multidisciplinary domains. Frequent subgraphs can be considered as the set of features for an input graph. Then, classical data mining techniques, such as clustering and classification, can be applied on the input graphs using theses features. For example, frequent subgraphs are used for graph clustering [9], where the set of input graphs are grouped based on the common frequent subgraphs that they contain. Also, frequent subgraphs are used to assign chemical compounds to various classes of interest [10]. Such classification has many applications in pharmaceutical research and are extensively used at various phases during the drug development process. Moreover, frequent subgraphs are used for image classification [11]. Given a set of images, each one is transformed to a graph, then frequent subgraphs are extracted from each graph and used to classify these images. FSM is also used in bioinformatics applications, such as finding biological mechanisms [12]. This task is achieved by mining a graph that represents all possible RNA structures and interactions. Subgraphs that are significantly more frequent in this graph than in a random graph are re-
turned as the possible biological mechanisms. Similarly, FSM is applied to protein-protein interaction networks (PPI) [13]. These networks are graphs where nodes represent proteins (and are labeled with their functionality) and edges represent interactions between these proteins. Such graphs are constantly updated to include new proteins and their interactions. A critical task for biologists is to predict the functionality (and add the corresponding label) of a new protein without experimental testing. The above task can be performed by matching frequent subgraphs with the neighboring structure of a protein with an unknown label, then the unknown label can be inferred from a matching frequent subgraph. FSM finds its way in data privacy as well. In many applications, information about users is represented as graphs. It is critical for the success of these applications to preserve the anonymity of their users. K-automorphism [14], which is a general framework for privacy preservation in graphs, utilize FSM to anonymize information stored in these graphs. Finally, graph indexing relies on finding the set of frequent subgraphs [15]. Given a query subgraph $q$ and input dataset $D_G$, an index of $D_G$ is built in order to efficiently find matches of $q$ in $D_G$. In graph databases, frequent subgraphs are used as the basic indexing objects since they are able to describe the aggregate relationships among graph objects.

Based on the above discussion, it is clear that FSM is an important tool, and it is used in many applications from a variety of domains. Current FSM algorithms have severe scalability limitations and cannot scale to large graphs. The next section reviews existing FSM solutions, and discusses their limitations.

### 1.2 Existing Methods and Limitations

Typical FSM systems follow a series of evaluation-extension steps. Candidate subgraphs are generated and evaluated to check which ones are frequent subgraphs. Then, those found to be frequent subgraphs are extended into larger candidates. This process repeats until no more candidate subgraphs are found. Some systems execute evaluation and extension as two disjoint steps [16, 17, 18], other systems combine them in a single operation [19, 20].
AGM [17] and FSG [18] are among the first attempts for FSM in the transactional setting. Both systems find new candidate subgraphs by following a level-by-level extension technique. A subgraph extension is based on joining two frequent subgraphs in order to create a larger candidate subgraph. Each candidate subgraph is evaluated and is extended if it is frequent. Two problems emerge as a result of this approach. The first is related to how candidates are generated. Different join paths may lead to the same subgraph, resulting in duplicates of the same subgraph that is evaluated repeatedly. The second problem is related to the efficiency of subgraph evaluation, which relies on finding all matches of the subgraph under consideration. Finding these matches is performed by one of the expensive subgraph matching algorithms, such as subgraph isomorphism, which is known to be NP-Complete. As a result, subgraph evaluation becomes excessively expensive.

GSPAN [19] alleviates the problems mentioned above. It utilizes an effective technique to avoid redundant evaluations for the same candidate subgraph. This technique relies on a depth-first search strategy that visits each candidate subgraph at most once. To improve the performance of the subgraph evaluation step, GSPAN utilizes occurrences of small frequent subgraphs to evaluate larger candidate subgraphs. As such, it avoids the expensive calls of subgraph isomorphism. Consequently, evaluating candidate subgraphs is significantly improved. Following a similar approach, SiGRAM [6] is proposed as a solution for FSM in the single graph setting. The main difference between SiGRAM and GSPAN is how each one computes the frequency. Since SiGRAM is designed for the single graph setting, it utilizes MIS for frequency computation, which is an anti-monotonic frequency metric. GSPAN and SiGRAM both require finding and maintaining all matches for each candidate subgraph. The number of matches of a given subgraph is known to increase exponentially with the graph size. This number becomes huge for large and dense graphs, and hence, the available storage becomes insufficient for storing these occurrences. Moreover, finding all these occurrences, which requires a huge number of subgraph isomorphism iterations, cannot be done in a reasonable time. As a result of these problems, these systems cannot
scale to real-world large graphs.

All of the mentioned approaches are single-threaded solutions. Since finding frequent subgraphs is expensive, a rational extension is to employ parallel and distributed solutions. Some attempts were proposed to parallelize FSM on a shared memory system [21, 22, 23]. However, these solutions are limited by the capabilities of a single machine. Other than the distributed solutions for the transactional setting [24, 25, 26, 27], there exist some attempts for the single graph setting [28, 29, 30, 31]. The single graph solutions are based either on the MapReduce [28] or the vertex centric [32] frameworks. These systems benefit from the capabilities of a cluster of machines, though, they suffer from the overhead of finding and maintaining all occurrences for each candidate subgraph. Furthermore, their performance is severely affected by problems related to their underlying frameworks. As a result of these limitations, existing distributed solutions cannot scale to a large number of machines nor process very large graphs.

One important aspect of modern applications is the requirement to manage graphs that are continuously evolving, such as social networks and web graphs. For example, in a social network, new users join the network and others leave. New relations among users such as friendship relations are added continuously. These dynamic changes can be represented by nodes/edges additions/deletions. Despite the need to manage such dynamic graphs, there is yet no enough effort devoted for mining this type of graphs. Current FSM techniques are not designed for mining dynamic graphs. To support dynamic graph mining using existing static FSM techniques, a straightforward approach is to run an FSM algorithm from scratch after each graph update. It is prohibitively expensive to follow such an approach given the fact that one iteration of FSM requires significant overhead. Due to the high computational complexity, existing approaches on evolving graphs either target the simpler case of a stream of small graphs [33], or produce approximate results [34]. There does not exist an efficient exact solution that targets mining large dynamic graphs.

The above discussion shows that existing solutions are limited and cannot cope with the
increase in the size of modern large graphs. There is a need to support graphs that are large and dense. Moreover, real applications require an incremental FSM solution to mine large dynamic graphs. In the following section, we discuss our proposed techniques for scalable FSM solutions.

1.3 Contributions

The goal of the work presented in this thesis is to scale FSM to large graphs. In particular, we propose a comprehensive solution that is composed of four components: 1- GramI [35], an efficient FSM solution that mines large graphs significantly faster than existing single-machine solutions without consuming excessive amounts of memory. 2- ScaleMine [36], a scalable FSM solution that utilizes a large number of compute nodes to mine large graphs. 3- PSI, an efficient method for enhancing the performance of evaluating candidate subgraphs, and finally 4- IncGM+ which supports FSM over dynamic graphs. Based on these four components, the overall efficiency of FSM is significantly improved. In the following subsection we briefly discuss each component:

1.3.1 GramI: Fast Single-Threaded FSM

The main time-consuming step for FSM is the frequency evaluation of candidate subgraphs. This step relies on finding subgraph matches using expensive algorithms such as subgraph isomorphism. Existing solutions [19, 6] relies on intensive usage of such expensive algorithms, and hence, they do not scale to large graphs. To alleviate the limitations of these approaches, we propose GramI, a single-threaded FSM algorithm that limits the use of subgraph matching algorithms and minimizes memory overhead. GramI needs to find a minimal number of occurrences that are required to satisfy a user-defined frequency threshold, other occurrences are ignored. A key part of the underlying idea of GramI is to evaluate the frequency of a subgraph as a constraint satisfaction problem (CSP). Three components are needed to evaluate a CSP; variables, domains and constraints. For each candidate
subgraph, its nodes represent the variables, the input graph nodes are the domains of these variables and the constraints of these variables are the edges of the candidate subgraph. GRAMI tries to solve the CSP until it finds a minimal set of appearances that are enough to evaluate subgraph frequency. Furthermore, GRAMI does not store intermediate results, thus, it consumes minimal memory overhead. Solving the CSP can still take exponential time in the worst case. GRAMI utilizes a set of optimizations that significantly improve its efficiency. These optimizations utilize properties of the input graph as well the candidate subgraphs. GRAMI is also extended to support a new type of matching that relaxes the connectivity between graph nodes. This relaxation allows the discovery of more interesting frequent patterns. By employing the proposed optimizations and by refraining from iterating over and storing all occurrences of a given subgraph, GRAMI scales to an order of magnitude larger graphs and achieves up to two orders of magnitude better performance compared to existing solutions.

## 1.3.2 ScaleMine: Parallel FSM

ScaleMine is the extension of GRAMI that supports mining massive graphs using large infrastructure. It distributes the workload of FSM on many processing cores, each core has access to the whole graph. ScaleMine scales to a large number of cores by balancing the workload among the available processors. In order to do so, ScaleMine utilizes a novel two-phases approach consisting of one approximate phase followed by an exact phase. First, ScaleMine executes a novel approximate FSM algorithm that uses sampling to: (i) identify a set of subgraphs that are frequent with high probability; (ii) collect various statistics about the input graph; and (iii) build a model to predict the execution time for evaluating each candidate subgraph. The approximate phase relies on the central limit theorem to decide when to stop sampling while attaining accurate decisions. The approximate phase comprises a small fraction of the total computational cost of the whole FSM task.

In the subsequent exact phase, ScaleMine utilizes the collected information to balance
the workload among the available processors. Since ScaleMine has information about the possible frequent subgraphs, it can provide enough tasks for available workers. Moreover, using the collected statistics, ScaleMine decides which tasks are expensive and need to be divided among several workers. In order to distribute the load of expensive tasks, ScaleMine uses the cost model built in the approximate phase to simulate various scenarios of intra-task parallelism, and decides whether it pays off to split expensive tasks into smaller subtasks. Compared with existing parallel solutions, our results show that ScaleMine is at least an order of magnitude faster, scales to an order of magnitude larger graphs, and is able to utilize 12X more compute cores.

1.3.3 PSI: Improved Frequency Evaluation

To minimize the overall overhead of FSM, it is essential to improve the frequency evaluation step, which is the focus of this component. This component uses pivoted subgraph isomorphism (PSI), which is a more efficient algorithm than subgraph isomorphism for evaluating the frequency of candidate subgraphs. Each pivoted subgraph isomorphism query is composed of a graph and a selected node inside this graph, which is called a pivot node. The goal of this operation is to find which input graph nodes match the pivot node in at least one occurrence of the query graph. Graph nodes that match the pivot node are called valid nodes, while the other nodes are called invalid nodes. This operation resembles how the MNI-based frequency is calculated in FSM.

A pessimistic-optimistic approach is presented as an efficient solution for the PSI problem. Two methods are designed, the first is optimized for invalid nodes, which is called pessimistic. For this method, the goal is to quickly prune graph nodes until no match is found. This pruning is achieved by comparing the neighbors around query nodes to those around corresponding graph nodes, and a graph node with insufficient neighbors is pruned. In order to do this comparison, a neighborhood signature is employed. This signature is able to capture information about the neighborhood of a given node beyond
direct neighbors, which allows effective pruning. The other method is called optimistic and is more suited for valid nodes. The optimistic method tries to reach a solution; a match to the query node, as quickly as possible. This is achieved by prioritizing graph nodes that, with higher chances, will lead to a solution.

We transform the PSI problem into a classification task and propose a solution which is based on the optimistic-pessimistic methods. The basic idea is that, given a list of candidate graph nodes, a small subset of these nodes is used for building a classification model. For the remaining nodes, the built model is used to predict which one is valid and which is invalid. For a node that is predicted to be valid, the optimistic matching method is used. For invalid nodes, the pessimistic method is used. Our pruning, classification and optimistic approaches are all based on an indexing technique which uses the neighbors around each graph node. This index is populated during a pre-processing step.

Our experiments show that the proposed approach significantly outperforms the state-of-the-art subgraph isomorphism solutions. We also show that by employing the proposed PSI technique, the efficiency of ScaleMine is improved by up to one order of magnitude.

1.3.4 \textit{IncGM+}: Incremental FSM

Existing FSM systems assume that the used graphs never change over time. Such assumption contradicts with real world applications such as social networks. For this type of graphs, we propose \textit{IncGM+}, our fourth component. \textit{IncGM+} is an incremental FSM solution that allows mining of dynamic graphs by utilizing novel pruning and indexing techniques. To this end, the “fringe” concept is introduced for the subgraph mining domain. Fringe subgraphs are subgraphs that lie on the border between frequent and infrequent subgraphs. Besides being representative of the whole search space, these subgraphs are the most sensitive to dynamic graph updates. \textit{IncGM+} maintains the fringe subgraphs and exploits them to prune the search space. Although this pruning allows significant improvement, more improvement can be achieved by enhancing or avoiding repeated evaluations
of fringe subgraphs. To do this, \textit{IncGM+} maintains a set of carefully selected occurrences of the fringe subgraphs. These occurrences are then utilized to avoid or improve the performance of evaluating the fringe subgraphs. Furthermore, \textit{IncGM+} utilizes information collected during past iterations to guide the processing in future iterations. Finally, \textit{IncGM+} is further extended to support batched updates in order to gain additional efficiency.

Our evaluation shows that \textit{IncGM+} is up to three orders of magnitude faster and consumes significantly less memory than competitors relying on existing techniques.

### 1.4 Outline

In this thesis, we highlight the bottlenecks of existing FSM solutions and show how to scale FSM to large static and dynamic graphs by solving the mentioned problems. The rest of this document is organized as follows: Chapter 2 gives an essential background needed to understand the FSM problem, then it reviews the related work. Chapter 3 gives detailed discussion on GRAMI [35], the single-threaded solution, and shows how it outperforms other solutions. In Chapter 4, we present ScaleMine [36], the parallel solution for FSM. The proposed technique for improving frequency evaluation \(^1\) is then discussed in Chapter 5. Finally, the dynamic graphs solution \(^2\) is presented in Chapter 6. Finally, the conclusion of this research is given in Chapter 7.

\(^1\)Paper under preparation

\(^2\)Paper under submission
Chapter 2

Background and Related Work

This chapter is composed of two parts. The first part gives a brief background about FSM, including the introduction of important definitions and the discussion of how FSM algorithms work. In the second part, we give an overview about the work related to FSM that covers exact, approximate and parallel solutions. Furthermore, we discuss other techniques related to incremental FSM.

2.1 Background

In this section we discuss the basics of FSM. First, we give necessary formal definitions, such as a graph, subgraph isomorphism and FSM. Then, we discuss how a typical FSM algorithm works. Finally, we discuss the MNI metric, which is an efficient metric for calculating the frequency of candidate subgraphs.

2.1.1 Definitions

A graph is a data structure that is composed of a set of nodes and a set of edges connecting these nodes. A graph is defined as follows:

Definition 1. A graph $G = (V, E, L)$ consists of a set of nodes $V$, a set of edges $E$ and a labeling function $L$ that assigns labels to nodes and edges.

Figure 2.1a illustrates an example of a graph. This graph represents a research collaboration network. Nodes represent authors, each edge represents a collaboration between two authors. Labels have important semantics in this graph; node labels represent author’s field
Figure 2.1: A collaboration graph $G$ and a subgraph $S$; nodes correspond to authors (labeled with their field of interest) and edges represent co-authorship (labeled with the number of co-authored papers).

of work (e.g., Databases) and edge labels represent the number of co-authored papers between two connected authors. To simplify presentation, this example illustrates undirected graphs with a single label for each node and a single label for each edge. However, the methods proposed in this thesis also support directed graphs with multiple labels per node and edge.

**Definition 2.** Let $S = (V_S, E_S, L_S)$ be a subgraph of a graph $G = (V, E, L)$. A subgraph isomorphism of $S$ to $G$ is an injective function $f : V_S \rightarrow V$ satisfying (a) $L_S(v) = L(f(v))$ for all nodes $v \in V_S$, and (b) $(f(u), f(v)) \in E$ and $L_S(u, v) = L(f(u), f(v))$ for all edges $(u, v) \in E_S$.

Intuitively, a subgraph isomorphism is a mapping from $V_S$ to $V$ such that each edge in $E$ is mapped to a single edge in $E_S$ and vice versa. This mapping preserves the labels on nodes and edges. These mappings are also called occurrences, matches or embeddings. For example in Figure 2.1, a subgraph $S (v_1 \rightarrow v_2 \rightarrow v_3)$ has three isomorphisms with respect to graph $G$, namely $u_1 \rightarrow u_3 \rightarrow u_4, u_5 \rightarrow u_4 \rightarrow u_3$ and $u_6 \rightarrow u_8 \rightarrow u_9$.

Given a user-defined frequency threshold, the goal of FSM is to find all subgraphs that appear frequently inside an input graph. The FSM problem is defined as follows:
**Input:** $G$ the input dataset, $\tau$ frequency threshold

**Output:** $R$ the set of all frequent subgraphs

1. $R \leftarrow \{\}$
2. $Candidates \leftarrow$ the set of all distinct edges
3. **while** $Candidates$ has elements **do**
   4. $C \leftarrow$ the first element of $Candidates$
   5. Remove $C$ from $Candidates$
   6. **if** IsFrequent($G, \tau, C$) **then**
      7. Add $C$ to $R$
      8. $Extensions \leftarrow$ Extend $C$ into larger subgraphs
      9. Add $Extensions$ to $Candidates$

**Algorithm 1:** Frequent Subgraph Mining

**Problem 1.** Given a graph $G$ and a minimum frequency threshold $\tau$, the frequent subgraph mining problem (FSM) is defined as finding all subgraphs $S$ in $G$ such that the frequency of $S$ is larger than or equal to $\tau$.

### 2.1.2 FSM Algorithm

The FSM algorithm follows a series of evaluation/extension steps. It starts by evaluating the smallest subgraphs (i.e., subgraphs with a single edge or node). Each subgraph evaluation involves finding whether the given subgraph is frequent or infrequent. The next step is to extend those subgraphs that are found to be frequent, then evaluate the new extensions. The process of evaluation and extension repeats until no more frequent subgraphs are found. Such process results in a lattice-like shape, where a path starts from a small subgraph of size one edge, followed by a set of subgraphs each is a one-edge extension of its predecessor.

Algorithm 1 describes the steps of a typical FSM algorithm. This algorithm begins by adding the set of edges to the candidates set (Line 2). Each candidate is later checked for either being frequent or infrequent by the IsFrequent method (Line 6). Those that are found to be frequent are added to the results set (Line 7) and extended into larger subgraphs (candidates) that will be checked in the next iterations (Line 8). These steps are repeated to generate the search space lattice until no more candidates can be generated.

Figure 2.2 illustrates the search space for a typical FSM task. Each element (circle)
represents a subgraph that exists at least once in the input graph. The elements at the bottom represent subgraphs with one edge. As we move up, each subgraph is extended by one edge. The topmost element represents the input graph (the largest possible subgraph in the search space). The number of elements at each level increases as we move up. This increase is due to the larger number of possible edge combinations of subgraphs at the higher levels. Once a certain level is reached, the input graph constrains these extensions and the number of elements decreases for next levels, resulting in a diamond shaped search space. Figure 2.2 shows that the search space is divided into two sets; the set $R$ of frequent subgraphs (solid circles) and the set of infrequent subgraphs (empty circles). The goal of FSM is to return the set $R$.

### 2.1.3 MNI-based Frequency Computation

As mentioned earlier, frequency evaluation in the single graph setting is more complicated compared to the transactional setting. The main difference is the definition of an appropriate anti/monotone frequency metric. Having an anti-monotone frequency metric requires that a subgraph has a frequency greater than or equal to its supergraph. The most intuitive way to measure the frequency of a subgraph is to count its occurrences in the input graph. Unfortunately, such metric is not anti-monotone since there are cases where a sub-
graph appears less times than its extension. For instance, in Figure 2.1 the single node subgraph DB appears three times while its extension DB $\downarrow$ IR appears four times. Having an anti-monotone frequency metric is of crucial importance. The literature defines several anti-monotone frequency metrics such as minimum image based (MNI) [4], harmful overlap (HO) [5], and maximum independent sets (MIS) [6]. These metrics differ in the degree of overlap they allow between subgraph isomorphisms, and the complexity of their computation. Compared to the other metrics, MNI [4] is considered the most efficient metric mainly because it:

- is the only metric that can be efficiently computed; the computation of MIS and HO are NP-complete [5, 6]
- provides a superset of the results of the alternative metrics; if we are interested in the MIS or HO metric we may pay their expensive computational cost and exclude the unqualified subgraphs [5].
- allows us to compute a limited number of appearances compared to MIS and HO that require computing all appearances [37].

**Definition 3.** Let $f_1, \ldots, f_m$ be the set of isomorphisms of a subgraph $S = (V_S, E_S, L_S)$ in a graph $G$. Also let $\text{MNI}_{col}(v_i) = \{f_1(v), \ldots, f_m(v)\}$ be the set of the distinct nodes that are mapped from $v \in V_S$ to $G$ based on these isomorphisms. The minimum image based frequency (MNI) of $S$ in $G$, is defined as $s_G(S) = \min \{ t \mid t = |\text{MNI}_{col}(v)| \text{ for all } v \in V_S \}$.

An MNI$_{tab}$ is a table that consists of $|V_S|$ columns, each column corresponds to a particular MNI$_{col}$. The MNI metric returns the length of the smallest MNI$_{col}$. A frequent subgraph must have an MNI$_{tab}$ where each MNI$_{col}$ has number of valid nodes greater then the user-given frequency threshold $\tau$. For an infrequent subgraph, its MNI$_{tab}$ contains at least one column with size less than $\tau$. For this case, the MNI$_{col}$ with size less than $\tau$ is the reason for the subgraph to be infrequent, and this column is called an invalid column. For instance, for subgraph $S$ and graph $G$ of Figure 2.1, $F(v_1) = \{u_1, u_5, u_6\}$, $F(v_2) = \{u_3, u_4, u_8\}$.
and \( F(v_3) = \{u_3, u_4, u_9\} \), thus \( s_G(S_1) = 3 \). To compare, the respective MIS metric is 2 since isomorphisms \( u_1 \xrightarrow{4} u_3 \xrightarrow{10} u_4 \) and \( u_5 \xrightarrow{4} u_4 \xrightarrow{10} u_3 \) overlap and the MIS metric counts overlaps only once.

### 2.2 Related Work

This section gives an overview about previous work related to FSM. First, (Section 2.2.1) reviews exact solutions in both the transactional and the single graph FSM settings. Then, a discussion on existing approximate FSM techniques is presented in subsection 2.2.2. Parallel FSM techniques are described in subsection 2.2.3. Subsections 2.2.4 and 2.2.5 discuss the previous work on mining streams of data, the first subsection is devoted for graphs and the later for itemsets. As FSM relies on subgraph isomorphism, the different subgraph isomorphism techniques are discussed in subsection 2.2.6.

#### 2.2.1 Exact FSM

There are two main settings for FSM, namely, the transactional setting and the single graph setting. The following discussion lists prior work for the two settings:

**Transactional mining.** This setting is concerned with mining frequent subgraphs on a dataset of many, usually small, graphs. Candidate subgraphs are generated then their frequencies are evaluated. Frequency is defined as the number of input graphs that contain the candidate subgraph. A candidate subgraph is frequent if its frequency is greater than or equal to the user-defined frequency threshold \( \tau \). FSQ [16], AGM [17] and FSG [18] are the first attempts to solve this problem. In these systems, mining starts with evaluating small subgraphs then new candidates are constructed by joining the frequent ones. This process repeats at each level until no more frequent subgraphs are found. The drawback of this approach is the costly join operation and the pruning of false positives. Such pruning requires finding matches of each subgraph, then prune those subgraphs with matches that cannot satisfy the given threshold \( \tau \). GSPAN [19] proposes a variation of the pattern growth
approach. It uses an extension mechanism, where subgraphs grow directly from a single subgraph instead of joining two previous subgraphs. Moreover, GSPAN introduces the use of canonical labels, as unique identifiers to subgraphs, to prevent repeated evaluation of candidate subgraphs that are generated from different extension paths. FFSM [20] follows the same approach of GSPAN with a set of optimizations for candidates generation and evaluation. FFSM is shown to perform better than GSPAN.

Due to the huge number of frequent subgraphs, many approaches focus on particular subsets of frequent subgraphs. For example, MARGIN [38] returns only maximal subgraphs. A maximal frequent subgraph is the one that is frequent and there is no other frequent subgraph that is a supergraph of this subgraph. Maximal frequent subgraphs are considered a compressed representation of the set of all frequent subgraphs. MARGIN starts with one maximal frequent pattern then using that pattern, the system smartly traverses the search space to construct the list of other frequent subgraphs. CLOSEGRAPH [39] focuses on another set of frequent subgraphs. It generates subgraphs that have strictly smaller frequency than any of their parts. The set of closed frequent subgraphs is more informative when knowing frequencies of the frequent subgraphs is important. LEAP [40] and GRAPHSIG [41], on the other hand, discover important subgraphs that are not necessarily frequent, but are significant representatives of the input graphs. ORIGAMI [42] follows a different approach; it mines the input graph by summarizing the search space. Such summarization allows for more efficient mining. ORIGAMI uses a randomized algorithm to randomly traverse the search space, seeking new regions and return a set of maximal subgraphs. Then, it extracts the representative subgraphs from the mined maximal subgraphs by considering the distance between them. Similarly, an output space sampling technique is used to discover frequent interesting discriminative subgraphs [43]. This framework relies on representing the search space as a partial order graph (POG). A random walk is conducted on the POG and the next state in the random walk depends on a user-given distribution. Thus, the user is allowed to specify what interests him by providing such distribution.
Single graph mining. There exists less work on the equally important single graph setting. The main difference between this setting and the transactional setting is the definition of an appropriate anti/monotone frequency metric. The literature defines several anti-monotone frequency metrics such as minimum image based (MNI) [4], harmful overlap (HO) [5], and maximum independent sets (MIS) [6]. These metrics differ in the degree of overlap they allow between subgraph isomorphisms, and the complexity of their computation. Compared to the other metrics, MNI is considered the most efficient for mining large graphs.

Several systems have been introduced for the single graph setting. SGRAM [6] uses the MIS metric, which computes the frequency as the maximal number of disjoint matches of a subgraph. SIGRAM focuses on labeled, sparse and undirected graphs. Vanetik et al. [44] presented a system that utilizes MIS as well, but it relies on edge-disjoint matches. Both approaches rely on the expensive computation of the maximal independent set problem, which is $NP$-complete. Bringmann et al. [4] proposed a system that utilizes MNI, which has a linear complexity. All of the mentioned approaches need to enumerate all subgraph occurrences. Such enumeration is drastically expensive, in terms of both memory and computation overheads, which hinders scaling to large graphs.

Sometimes, the number of frequent subgraphs is huge, and there is a need to focus only on the important ones. Milo et al. [45] proposed an approach to mine subgraphs (network motifs) that appear more frequently than in randomized networks. Their approach is based on calculating the probability of a pattern to appear in a randomly generated graph compared to the given graph, if the difference is significant, then this pattern is of interest. In order to minimize the overhead of finding all frequent subgraphs, other systems focus on specific types of frequent subgraphs. SpiderMine [46] mines only large frequent subgraphs. As a result, it misses many frequent subgraphs. SkinyMine [47] focuses on a different set of subgraphs. It targets skinny subgraphs; those subgraphs with long backbone from which short twigs branch out, those subgraphs are only useful in very specific applications.
2.2.2 Approximate FSM

Many approximate techniques were proposed for FSM since the complexity of exact solutions limits their ability to mine large graphs. GREW [48] is an approximate FSM solution that rely on heuristic techniques to prune large parts of the search space. Consequently, GREW discovers a small subset of the frequent subgraphs. SEuS [49] is another approximate method that constructs a compact summary of the input graph. This summary is obtained by collapsing together all vertices of the input graph that have the same label. This summarization facilitates pruning many infrequent candidates. However, it is only useful when the input graph contains few and very frequent subgraphs. GAPPROX [50] employs an approximate version of the MIS metric. It relies on enumerating all intermediate isomorphisms, but it allows approximate matches. SUBDUE [51] is a branch-and-bound technique which mines subgraphs that can be used to compress the original graph. For evaluating the extent to which a particular pattern can compress the original graph, it uses the minimum description length (MDL) principle, and employs a heuristic beam search to narrow the search-space. These approximations improve the overall computational efficiency but at the same time prevent SUBDUE from finding subgraphs that are indeed frequent.

Khan et al. [52] defined proximity patterns as the set of labels that co-occur in the neighborhood of a particular node. Based on this definition, they proposed a mining approach that relaxes the connectivity constraint of frequent subgraphs, introducing a new type of patterns that cannot be found by other approaches. Allowing approximate matches, like in [50, 51, 52], may result in producing patterns that do not exist in the input graph, which might be problematic for applications that require exact matches of the frequent subgraphs. Finally, graph sampling is used to construct a smaller version of the input graph, and then FSM is applied on the sampled graph [53]. Since mining is conducted on a relatively small graph, the computation overhead is significantly minimized. Several sampling techniques are examined, and the best performing is based on sampling a connected area of the input graph rather than sampling random nodes, edges or walks.
2.2.3 Parallel FSM

In recent years, much effort has been devoted for parallelizing FSM since it is an expensive graph operation. Early attempts were proposed for the transactional setting. For example, [54, 23] utilize multi-core and shared memory architectures. By using a shared memory architecture, the need to partition the dataset is completely avoided. Instead, more focus is given for load balancing, data locality, subtask granularities, task scheduling and locking mechanisms.

The shared memory-based techniques are limited by the available memory and processing power within a single machine. Another direction is to partition and distribute the dataset over several machines. The work in [55] partitions the input graph into a set of fragments, each fragment has to fit in the memory of a single machine. Each machine reports its local frequent subgraphs, and the complete set of frequent subgraphs is computed by combining the reported subgraphs using some heuristic techniques. Another set of techniques utilize the MapReduce framework [28, 27]. Original graphs and intermediate results are distributed among workers based on a hashing function. These systems were evaluated on small and sparse graphs of maximum size of 1M graphs with an average of 50 edges per graph. A more recent MapReduce-based work [25] utilizes a two steps filter and refinement approach. The first step is to divide the input graphs into a set of disjoint partitions, do some local frequent subgraph mining on each partition, and send the locally frequent subgraphs plus other properly selected infrequent ones to the next step. The second step is to process and refine the set of candidate frequent subgraphs to get the complete set of frequent subgraphs. This system scales to a graph dataset of 64M graphs with an average of 52 edges per graph. Another direction is to use message passing. ParCocain [56] is proposed to parallelize Cocain [57]; a coherent quasi clique mining technique. Their contribution is to dynamically distribute subtasks when the static distribution cannot work efficiently. Although the problem definition is different, the proposed ideas are generic enough to support other frequent subgraph mining techniques.
Other systems are proposed for the single graph setting. A system based on shared memory architecture is introduced in [58], it parallelizes SIGRAM by exploiting parallelism in two levels: 1-parallelizing the candidates extension, and 2-parallelizing the frequency computation. This system inherits all of the limitations associated with SIGRAM. It is designed only for small sparse graphs. The experiments show that this system scales well on 30 cores, after that its scalability severely suffers. Other efforts [22, 28, 31, 30, 29] proposed distributed systems for the single graph setting. PARALLEL SUBDUE [22] is a distributed FSM system that scales to graphs of size up to 150K edges. Three partitioning schemes were introduced: functional partitioning, dynamic partitioning and static partitioning. For functional partitioning, the master is responsible for generating subtasks and dividing them among processors. For dynamic partitioning, each processor is responsible for evaluating and expanding an initial disjoint set of candidates; these sets will eventually overlap as candidate subgraphs becomes larger. As for static partitioning, Metis [59] is used to partition the input graph, then each partition is independently mined by one processor. Experiments were conducted on a cluster of 16 machines and show that the best performing approach with the least memory consumption is static partitioning. ParGraph [60] is an MPI solution for parallelizing FSM. It assumes the input graph fits in memory of each used machine. The basic idea is to partition the search space among the available workers. When a worker finishes its assigned tasks, it request new tasks from other workers that have extra unprocessed tasks. MRPF [28] and MR-SUB [31] rely on the MapReduce framework. They both require a user-given parameter for the maximum size of the frequent subgraphs, such requirement limits their applicability. Furthermore, since these systems are based on MapReduce, they inherit its limitations. Since FSM is an iterative process, these systems incur significant disk access cost due to flushing/reading the intermediate data between the subsequent MapReduce jobs.

Arabesque [30] and Pegi [29] leverage the vertex-centric programming model to provide a solution for FSM on large graphs. Arabesque supports FSM through a set of internal
abstractions, these abstractions facilitate retrieval, processing and extension of subgraph matches. Pegi, on the other hand, utilizes a combination of coarse-grain and fine-grain processing, where the master node controls the flow of the mining process and individual vertices are in charge of embedding discovery. Pegi utilizes aggregators to synchronize the flow of information between the workers and the master node. The main drawback of both systems is that they need to find all subgraph embeddings for computing frequencies as well as for subgraph extension. The number of embeddings is known to increase exponentially with the graph size, which significantly increases the communication volume between machines and hinders the scalability of the system. Moreover, since both systems rely on a fine-grain programming model, only vertices related to frequent subgraphs will be computationally active while others will be inactive. As a result, these systems may suffer from load imbalance since they cannot equally distribute active vertices across the workers.

DistGraph [61] is a more recent approach that parallelizes FSM by partitioning the input graph among the available compute nodes. Each node extracts the possible frequent subgraphs within its partition. Then, results from each partition are communicated back to a master node. The problem of missing subgraphs that span across different partitions always happens when a graph is partitioned among different compute nodes. DistGraph solves this issue by following an external neighbor expansion approach. The basic idea is to expand each partition with external edges after mining all subgraphs of a particular number of edges. DistGrah is also supported by several optimizations to minimize the communication overhead. DistGraph is shown to support a graph with four billion edges, which is the largest reported graph for FSM. This graph is artificially created by replicating a real graph for 64 times.

2.2.4 Dynamic graph mining

Different problems have been studied under the umbrella of dynamic graph mining, each with its own definition. An approximate technique for mining nodes that frequently co-
occur in a stream of small subgraphs is proposed in [62]. Another definition of dynamic graph mining is adopted by [63, 64], where a snapshot is taken from the input graph, then each edge is labeled with a time-based sequence of labels. These edge labels are the result of dynamic changes in the edge label over time. This snapshot is then mined to find frequent structures that share similar sequences of edge labels. Another work [65] tries to answer the following question: “Given that a subgraph is currently frequent, which subgraph is expected to be frequent in the future?” Closer to the topic of this thesis but for the transactional setting, IncGraphMiner [33] mines closed frequent subgraphs. In this setting, the unit of update is a graph. StreamFSM [34] is a sampling-based FSM technique. First, the neighborhoods around the added nodes/edges are sampled. Then, these sampled neighborhoods are mined for frequent subgraphs. This approach supports only batches of addition updates; moreover, it is approximate.

None of the above techniques proposes an exact solution for the problem of incremental FSM in dynamic graphs. In the next subsection, we give more details on the related topic of frequent itemset mining over a stream.

2.2.5 Frequent Itemset Mining Over a Stream

Frequent itemset mining over data streams has witnessed huge attention from the research community. Approximate [66, 67, 68] and exact solutions [69, 70, 71, 72, 73, 74] were proposed for incremental mining of a stream of itemsets.

There are two main challenges associated with mining frequent itemsets over a stream; memory and processing overheads. In order to minimize such overheads, many approaches give approximate results [66, 67, 68]. The basic approximation technique is to incrementally maintain a list of the currently frequent itemsets. To discover future frequent itemsets, another list of infrequent itemsets is maintained. This list contains itemsets having frequency > ϵ, where ϵ < τ. Increasing the difference between ϵ and τ increases the accuracy, but has an impact on memory and processing requirements. Besides having dif-
ferent error bounds, the introduced systems differ in how they consider the time of feeding
the system with these transactions. They range from ignoring the time information [75, 76],
to focusing only on the most recent transactions [67, 68]. Another challenge is to decrease
the memory overhead. Some approaches use compressed prefix trees [76], while other
approaches focus on maximal itemsets [68].

For many applications, approximate solutions are not acceptable. Exact solutions are
introduced for such cases. Moment [69] is an exact solution that mines closed frequent
itemsets. It utilizes a fringe composed of four sets of itemsets. One set represents the
frequent closed itemsets while the other three sets represent the boundary between fre-
quent closed itemsets and other itemsets. Following this, the StreamGen system [77] is
proposed to mine the frequent itemset generators. An itemset generator is an itemset that
does not contain an itemset having the same frequency. In StreamGen, three sets are main-
tained for incremental stream mining. One is the set of frequent itemset generators, and
the other two sets contain the boundary itemsets between generators and other itemsets.
INSTANT [70] and INSTANT+ [71] are exact solutions that support only insertion up-
dates and mine maximal frequent itemsets. The two systems utilize an approach based on
maintaining all itemsets with frequency $\leq \tau$. CFI-Stream [72] focuses on mining closed
frequent itemsets. NewMoment [73] and TMoment [74] extend Moment and maintain a set
of all frequent closed itemsets as well as all 1-itemsets. It is not straightforward to apply
the mentioned techniques for incremental subgraph mining due to the differences between
the two settings.

2.2.6 Subgraph isomorphism

FSM relies on the computation of subgraph isomorphisms, which is NP-Complete. Sub-
graph isomorphism is the bottleneck of FSM due to its excessive usage and its high com-
plexity. Despite its theoretical complexity, many research efforts had been done to reduce
its overhead in practice. The first practical algorithm that addresses this problem follows
a backtracking approach [78]. Since then, several performance enhancements were proposed, ranging from CSP-based techniques [79], search order optimization [80], indexing [15] and parallelization [81]. As reported in [82], GRAPHQL [80] is considered one of the best subgraph isomorphism techniques, though its performance is not stable. GRAPHQL prunes the search space by using local and global pruning techniques. Moreover, GRAPHQL utilizes a search order optimization technique which is based on a global cost model. This cost model is based on the probability of having edges between two labels and the number of occurrences for each label. GRAPHQL uses this cost model to select the best plan among the set of possible plans. In order to minimize the number of possible plans, GRAPHQL utilizes a greedy-based approach for plan construction. TURBOISO [83] is a more recent work, it outperforms all its previous techniques. TURBOISO superiority is the result of two contributions: 1-grouping similar parts of the query and process them at once, and 2-selecting an optimal search order plan based on the local neighborhood of each input graph node. BOOSTISO [84] exploits the relationship among the input graph nodes to avoid duplicate computation of intermediate results rather than utilizing the query structure as in TURBOISO. As such, significant computation overhead is pruned. The beauty of this technique is that it can be plugged into other techniques, resulting in improved performance. CFL-Match [85] is the state-of-the-art for subgraph isomorphism. It significantly minimizes the number of intermediate results by decomposing a given query into a core subgraph and a forest such that core is processed first. Moreover, an auxiliary data structure is designed to accurately estimate the number of embeddings of query paths and is also used for efficient generation of those embeddings.

Although the mentioned techniques led to significant improvements [80, 15, 82, 80, 83], they are not as effective in the FSM problem for two reasons: First, subgraph isomorphism techniques are effective in finding all appearances of a subgraph, while for the frequent subgraph mining task, it is sufficient to find the minimum appearances that satisfy the frequency threshold. This difference affects the way graph nodes are traversed, and it allows
minimizing the number of node visits during search. Additionally, modern techniques em-
ploy global pruning and indexing techniques. Forming such structures on large graphs
results in a huge and often unacceptable overhead. Thus, it is important to employ smarter
techniques to find subgraph occurrences that are more suitable to FSM.

Another related topic is subgraph listing. It is the task of listing matches of a subgraph
inside an unattributed input graph. Early work focused on specific types of subgraphs,
such as triangles, quadrangle and cliques [86] and more recently a more efficient solution
for triangles [87]. Parallel solutions were proposed to improve the efficiency. Some solu-
tions [88, 89] were built on top of Map-reduce frameworks. These solutions suffer from
load imbalance, huge number of intermediate results and require excessive number of join
operations. More recently, P$g$L [90] avoids the expensive join operations and follows a
divide and conquer approach to efficiently find subgraph matches.

2.2.7 Graph Indexing

Since subgraph isomorphism is NP-Complete, graph indexes are used to improve the eval-
uation of subgraph matching queries. The literature contains many efforts that use indexes
for the transactional setting. The basic idea is to use a filter-then-verify approach. The index
is used to filter out input graphs that cannot match the given query, then the more expensive
matching task is applied on the remaining graphs. An early example is GraphGrep [91], it
uses paths to index the input graph. The labels sequence of all possible paths (up to a given
length) are maintained and used to index the input graphs. This is followed by gIndex [92],
which is an index that maintains a list of frequent subgraphs. These subgraphs are com-
pared to the given query and are used to prune non-qualifying graphs. gIndex shows up to
an order of magnitude improvement over GraphGrep. Both of these approaches require ex-
tensive processing and storage overhead. Closure-tree [93] tries to avoid such problems by
employing graph closures. A graph closure is a normal graph, but each node and edge have
multiple labels. A single graph closure is a generalization of multiple graphs and is used
to represent multiple input graphs. Based on that, a closure-tree represents all input graphs in a hierarchy where each node is a graph closure of its children. This index is shown to outperform GraphGrep by an order of magnitude. FG-index [94] is another index that aims at improving the verification step. This index maintains a set of frequent subgraphs, if a query belongs to the frequent subgraphs set, then verification can be done by only using the index. Otherwise, verification is required. Instead of using frequent subgraphs, (Tree+Delta≥Graph) [95] uses frequent trees plus a small set of discriminative graphs. The conducted experiments show that Tree+Delta≥Graph has much smaller indexes and outperforms both gIndex and Closure-tree indexes. A more recent approach [96] relies on indexing a simple feature structure that compresses a number of features. By having this compressed representation, more efficient and effective pruning is achieved.

Less work has been done for the single graph setting. These approaches also follow the filter-then-verify approach. Since there is only a single graph, vertex filtering is used instead of graph filtering. This approach is called vertex-at-a-time. GADDI [97] is the first approach that supports single graph indexing. GADDI employs an index that captures graph structures that exist between any pair of vertices. Based on this index, a two way pruning technique is used to efficiently prune candidate matches. Moreover, a dynamic matching algorithm is used to avoid redundant computations. GADDI suffers from the excessive cost needed for creating its index. NOVA [98] utilizes an index called nIndex to efficiently enumerate subgraph matches. This index is based on having a vector signature for each graph node. Each vector has multiple features and each feature represents a particular characteristic of the node. Then, node ordering and pruning are applied using this representation. NOVA significantly outperforms GADDI in both index construction cost and query evaluation efficiency. DSI [99] follows a similar approach, it indexes the neighborhood of all input graph vertices. The index is then used to filter out unqualified graph nodes. SPath [100] is an index which is based on maintaining shortest paths around graph nodes. SPath follows a path-at-a-time approach instead of vertex-at-a-time. This
approach allows more efficient processing. A given query is decomposed into multiple paths, then pruning and matching are applied using those paths. Finally, SMS [101] is a subgraph matching algorithm that utilizes a simple yet effective index. Each graph node is represented by the labels appearing in its neighborhood. Furthermore, SMS utilizes a graph partitioning technique to improve the efficiency. SMS is shown to significantly outperform NOVA, SPath and GADDI.

2.2.8 Graphlets Mining

Given a value $K$, a graphlet is a subgraph of an input graph and it consists of $K$ nodes. Most of the research focuses on connected induced graphlets due to their importance in exploring global and local structure properties of different graph types. Graphlets are used in many domains ranging from social network analysis to other applications in biology [102], chemoinformatic [103] and image analysis [104]. Due to their importance in such applications, it is required to have an efficient solution for discovering and counting frequencies of those graphlets.

There exists several attempts for graphlet mining solutions. Early work [105, 106, 107] focused on finding exact frequencies of graphlets of a particular size. But, these approaches suffer from inefficiencies especially for large graphs. Following this, RAGE [108] is an approach that counts induced and non-induced graphlets with size up to four. It presented different algorithms, each one tackles a particular graphlet type. Orca [109] groups the set of automorphic nodes into a set of orbits. Then, frequency computation is conducted by enumerating small graphlets and utilizing relationships among the different orbits. This approach outperforms previous solutions by an order of magnitude. More recently, Ahmed et. al [110] proposed a framework that is on average 460X faster than previous approaches. Also, it supports single-threaded and parallel execution. This technique relies on counting a few graphlets for each graph edge, based on these counts they propose an approach to obtain exact counts of other graphlets in constant time.
Approximate solutions are important for situations where the cost of exact methods is intolerable and approximate results are acceptable. Moreover, there are many cases where the whole graph cannot be loaded in memory for full traversal. Examples of such situations are when full access to the graph is restricted or when the dataset comes from a data stream. For such situations, approximation, usually using sampling and random walk is a good solution. A number of approaches [111, 112, 113] rely on sampling to obtain an estimate of graphlets counts. These approaches assume the whole graph is fully accessible. Other approaches work on graphs with restricted access [114, 115], they estimate the relative frequencies among a set of graphlets. Later, a random walk-based solution [116] is also proposed for the restricted access setting. Its goal is to obtain an unbiased estimation of graphlets frequencies.

Although both FSM and graphlets counting have strong similarities, they have different goals. FSM discovers subgraphs that are frequent, while graphlets mining discovers the distribution of frequencies for a particular set of subgraphs (graphlets) in an input graph.
Chapter 3

**GRAMI: Single-Threaded Frequent Subgraph and Pattern Mining**

### 3.1 Introduction

Considering the collaboration graph $G$ of Figure 3.1, assume there is a user interested to mine important collaborations among authors. Typically, in such graphs, frequent subgraphs are most likely to show collaborations among authors having the same field of work (i.e., collaborations among DB researchers). In order to reveal more interesting subgraphs, the user would progressively reduce the frequency threshold until subgraphs showing interdisciplinary collaborations are discovered (i.e., among AI, DB and IR researchers). Lowering the frequency threshold increases the number of qualified intermediate results and intensifies the already expensive computations of the mining process. For example, a state-of-the-art method for frequent subgraph mining crashes after a day consuming 192GB for an input graph of 100K nodes and 1M edges. Therefore, the development of efficient frequent subgraph mining algorithms that support large graphs and low frequency thresholds is very crucial.

Existing literature considers two settings: transactional and single graph. In this work, the focus is on the *single/graph* setting that considers one large graph [52, 48, 6]. For this setting, a subgraph is frequent if it has support value of at least $\tau$ in the input graph. Such a context is required in many modern applications, including social, collaboration and PPI networks. The most straightforward method to evaluate the frequency of a subgraph $S$ in a graph $G$ is to look for *isomorphisms* of $S$ in $G$ [5, 80, 48, 6]. Isomorphisms are exact matches of $S$ in $G$ that pair nodes, edges and labels. For example, in the collaboration
A typical method to mine frequent subgraphs in a single graph, is a grow/and/store method that proceeds with the following steps:

1. Find all nodes that appear at least \( \tau \) times and store all of their appearances.
2. Extend the stored appearances to construct larger candidate subgraphs, evaluate their frequency, and store all appearances of the new frequent subgraphs.
3. Repeat Step 2 until no more frequent subgraphs are found.

Existing systems such as SiGRAM [6] use variations of this grow/and/store method. These approaches take advantage of the stored appearances of smaller subgraphs to efficiently evaluate the frequency of larger candidates. The main bottleneck of such algorithms is the creation and storage of all appearances of each subgraph. The number of such appearances depends on the size and the properties of the input graph and the subgraph; it can be prohibitively large to compute and store, rendering grow/and/store solutions infeasible in practice.

In this chapter, we propose GRAMI (Graph Mining); a novel framework that addresses the frequent subgraph mining problem. GRAMI undertakes a novel approach differentiating it from grow/and/store methods. First, it stores only the templates of frequent subgraphs, but not their appearances on the graph. This eliminates the limitations of the grow-
/and/store methods, and allows GRAMI to mine large graphs and support low frequency
thresholds. Also, it employs a novel method to evaluate the frequency of a subgraph.
More specifically, GRAMI models the frequency evaluation as a constraint satisfaction
problem (CSP). At each iteration, GRAMI solves the CSP until it finds the minimal set of
appearances that are enough to evaluate subgraph frequency, and it ignores all remaining
appearances. The process is repeated by extending the subgraphs until no more frequent
subgraphs can be found.

Solving the CSP can still take exponential time in the worst case. In order to support
large graphs in real-life applications, GRAMI employs a heuristic search and a series of op-
timizations that significantly improve performance. More specifically, GRAMI introduces
novel optimizations that (a) prune large portions of the search space, (b) prioritize fast and
postpone slow searches and (c) take advantage of special graph types and structures. By
avoiding the exhaustive enumeration of appearances and using the proposed optimizations,
GRAMI supports larger graphs and smaller frequency thresholds than existing approaches.
For example, to compute the frequent patterns of the 100K nodes/1M edges graph that the
state-of-the-art grow-and-store method crashed after a day, GRAMI needs only 16 minutes.

Additionally, we propose three extensions to the original GRAMI framework. The first
one considers graphs such as social or research networks, that may contain incomplete in-
formation and transitive relationships. In such cases indirect relationships (like a friend of
a friend) reveal neighborhood connectivity and proximity information. To explore these
relationships, patterns were introduced [117, 52, 118]. Patterns establish a more pow-
derful definition of matching, than subgraphs, that captures indirect connections by replacing
edges with paths. To mine frequent patterns, we have appropriately extended GRAMI.
For instance in Figure 3.1, GRAMI may also consider $u_5 \rightarrow u_8 \rightarrow u_9$ to be a match of $S_1$
since $u_5$ (labeled DB) is indirectly connected to $u_8$ (labeled IR). The second extension,
CGRAMI, allows the user to define a set of constraints, both structural (e.g., the subgraph
is allowed to have up to $\alpha$ edges) and semantic (e.g., a particular label cannot occur more
than $\alpha$ times in the subgraph). The constraints are used to prune undesirable matches and limit the search space. The final extension, AGRAMI, is an \textit{approximate} version, which approximates subgraph frequencies. The approximation method may miss some frequent subgraphs (i.e., has false negatives), but the returned results are \textit{not} approximate (i.e., does not have false positives).

Noteworthy, GRAMI and its extensions support directed and undirected graphs and may be applied to both single and multiple labels (or weights) per node and edge.

In summary, our main contributions are:

- We propose GRAMI, a novel framework to mine frequent subgraphs in a large single graph. GRAMI is based on a novel idea that refrains from computing and storing large intermediate results (appearances of subgraphs). A key part of the underlying idea is to evaluate the frequency of subgraphs using CSP.
- We offer a heuristic search with novel optimizations that significantly improve GRAMI’s performance by pruning the search space, postponing expensive searches, and exploring special graph types.
- We develop a variation of GRAMI that is able to mine frequent patterns, a more powerful version of matching that is required in several modern applications.
- We present CGRAMI, a version that supports structural and semantic constraints, and AGRAMI, an approximate version which produces results with no false positives.
- We experimentally evaluate the performance of GRAMI and demonstrate that it is up to 2 orders of magnitude faster than existing methods in large real-life graphs.

The rest of the chapter is organized as follows. Section 3.2 formalizes the problem. Section 3.3 presents GRAMI and its optimizations. Section 3.4 discusses the extensions of GRAMI. Section 3.5 presents the experimental evaluation. Finally, section 3.6 concludes.
3.2 Preliminaries

In this chapter we propose GRAMI to solve the problem of FSM (Problem 1). This problem does not consider finding the actual number of appearances (i.e., frequency) provided that it is greater than $\tau$. This is very useful in several applications [11, 6], but there are other applications that demand the exact number of appearances (like graph indexing [15]). Also note that Problem 1 is computationally expensive since it relies on the $NP$-Complete subgraph isomorphism problem [119].

The subgraph isomorphism definition (Definition 2) requires matching of both node and edge labels. For instance in Figure 3.1, subgraph $S_2$ has only one isomorphism (formed by nodes $u_1, u_2$ and $u_3$). Recent research argues that this matching is rather restrictive, and relaxes it by allowing indirect relationships and differences between the edges of the graph and the subgraph [117, 52, 118]. Such frameworks may also consider subgraph $u_6 \xrightarrow{DM} u_8 \xrightarrow{20} u_7$ to be a match of $S_2$ since DM and DB are indirectly connected. We refer to this match as a pattern. For mining frequent patterns, we adopt the pattern matching definition as outlined in [118]. Specifically, we employ a distance metric to measure the distance between two nodes. To this end, we may use any metric function, i.e., a function that satisfies the triangle inequality [118]. Typically, the distance function is computed based on the edge labels (or weights) but it may also be defined on other graph properties (e.g., the number of hops between two nodes).

For graph $G$ of Figure 3.1, we may use a distance function $\Delta_h(u, v)$ defined as the number of hops in the shortest path that connects $u$ and $v$. For instance, $\Delta_h(u_0, u_3) = 2$. Alternatively, we may use $\Delta_p(u, v)$ defined as the minimum sum of the inverse of edge weights among the paths that connect $u$ and $v$. For an example, $\Delta_p(u_6, u_7) = 1/4 + 1/20 = 0.3$. Intuitively, a shorter distance denotes a stronger collaboration. Figure 3.2 illustrates the values of $\Delta_p$ for the graph $G$ of Figure 3.1. Solid lines correspond to the original edges of the graph, while dotted lines illustrate some additional transitions (for figure clarity, we do not show all transitions).
**Definition 4.** A graph $P = (V_P, E_P, L_P)$ is a pattern of a graph $G(V, E, L)$ iff $V_P \subseteq V$, $L_P(v) = L(v)$ for all $v \in V_P$ and $L_P(e) = \emptyset$ for all $e \in E_P$.

In other words, a pattern is analogous to a subgraph but without considering edge labels. For instance, a pattern $P_1$ of the graph $G$ is presented in Figure 3.2b.

**Definition 5.** Let $P = (V_P, E_P, L_P)$ be a pattern of a graph $G = (V, E, L)$, $\Delta$ be a distance metric function, and $\delta$ be a user-defined distance threshold. A pattern embedding of $P$ to $G$ is an injective function $\phi : V_P \rightarrow V$ satisfying (a) $L_P(v) = L(\phi(v))$ for all nodes $v \in V_P$ and (b) $\Delta(\phi(u), \phi(v)) \leq \delta$ for all edges $(u, v) \in E_P$.

The minimum image based support for a pattern, denoted by $\sigma_G(P)$, can be computed as in Definition 3 by replacing the isomorphisms $f_1, \ldots, f_m$ with the pattern embeddings $\phi_1, \ldots, \phi_\mu$. For example consider Figure 3.2; setting a distance threshold $\delta = 0.3$, we have $\sigma_G(P_1) = 2$. The corresponding embeddings are illustrated by the gray areas. Note that there are other possible matches to $P_1$ but only the indicated two matches satisfy the constraint $\Delta(\phi(u), \phi(v)) \leq \delta$.

**Problem 2.** Given a graph $G$, a distance function $\Delta$, a distance threshold $\delta$, and a minimum support threshold $\tau$, the frequent pattern embedding mining problem is defined as finding all patterns $P$ of $G$ such that $\sigma_G(P) \geq \tau$.

We defined two problems; frequent subgraph mining (Problem 1) and frequent pattern
mining (Problem 2). In the following, we discuss GRAMi; a single-threaded solution that supports both problems.

3.3 The GRAMi Approach

GRAMi introduces a novel technique that addresses the frequent subgraph mining problem without exhaustively enumerating all isomorphisms in the graph. To this end, GRAMi models the underlying problem as a constraint satisfaction problem (Subsection 3.3.1). Following, Subsection 3.3.2 applies the model to solve the frequent subgraph mining problem. Subsection 3.3.3 proposes several optimizations to enhance the performance. The frequent pattern mining problem together with other interesting extensions are discussed in Subsection 3.4.

3.3.1 The CSP Model

A constraint satisfaction problem (CSP) is represented as a tuple $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ where (a) $\mathcal{X}$ is an ordered set of variables, (b) $\mathcal{D}$ is a set of domains corresponding to variables $\mathcal{X}$, and (c) $\mathcal{C}$ is a set of constraints between the variables in $\mathcal{X}$. A solution for the CSP is an assignment to the variables in $\mathcal{X}$, such that all constraints in $\mathcal{C}$ are satisfied. The subgraph isomorphism problem (Definition 2) can be mapped to a CSP as follows.

Definition 6. Let $S(V_S, E_S, L_S)$ be a subgraph of a graph $G(V, E, L)$. The subgraph $S$ to graph $G$ CSP, is a CSP $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ where:

1. $\mathcal{X}$ contains a variable $x_v$ for every node $v \in V_S$.
2. $\mathcal{D}$ is the set of domains for each variable $x_v \in \mathcal{X}$. Each domain is a subset of $V$.
3. Set $\mathcal{C}$ contains the following constraints:
   a) $x_v \neq x_{v'}$, for all distinct variables $x_v, x_{v'} \in \mathcal{X}$.
   b) $L(x_v) = L_S(v)$, for every variable $x_v \in \mathcal{X}$.
   c) $L(x_v, x_{v'}) = L_S(v, v')$, for all $x_v, x_{v'} \in \mathcal{X}$ such that $(v, v') \in E_S$. 
To simplify notation, whenever it is clear from the context, we use \( v \) to refer to a node of the subgraph and to the corresponding variable \( x_v \) of the CSP as we do in the following example.

**Example 1.** Consider Figure 3.1. The subgraph \( S_1 \) to graph \( G \) CSP is defined as:

\[
\begin{pmatrix}
(v_1, v_2, v_3), \{\{u_0, \ldots, u_9\}, \ldots, \{u_0, \ldots, u_9\}\}, \\
\{v_1 \neq v_2 \neq v_3, L(v_1) = \text{DB}, L(v_2) = L(v_3) = \text{IR}, \\
L(v_1, v_2) = 4, L(v_2, v_3) = 10\}
\end{pmatrix}
\]

The following proposition relates the subgraph to a graph CSP with the subgraph isomorphism \( f \) (Definition 2).

**Proposition 1.** A solution of the subgraph \( S \) to graph \( G \) CSP corresponds to a subgraph isomorphism of \( S \) to \( G \).

Intuitively, a solution assigns a different node of \( G \) to each node of \( S \), such that the labels of the corresponding nodes and edges match. For instance, a solution to the CSP of Example 1 is the assignment \((v_1, v_2, v_3) = (u_1, u_3, u_4)\).

**Definition 7.** An assignment of a node \( u \) to a variable \( v \) is valid if and only if there exists a solution that assigns \( u \) to \( v \). Each valid assignment corresponds to an isomorphism.

In Example 1, \( v_2 = u_3 \) is a valid assignment; \( v_2 = u_0 \) is invalid.

**Proposition 2.** Let \((X, D, C)\) be the subgraph \( S \) to graph \( G \) CSP. The MNI support of \( S \) in \( G \) satisfies \( \tau \), i.e., \( s_G(S) \geq \tau \), iff every variable in \( X \) has at least \( \tau \) distinct valid assignments (i.e., isomorphisms of \( S \) in \( G \)).

Proposition 2 is a key part of this work since it provides a method to determine if a subgraph \( S \) is frequent in \( G \). To this end, we may consider the \( S \) to \( G \) CSP and check the number of valid assignments of every variable. If for every variable there exists \( \tau \) or more valid assignments, then \( s_G(S) \geq \tau \) and \( S \) is considered frequent. Continuing Example 1,
Algorithm: **FREQUENTSUBGRAPHMINING**

**Input:** A graph $G$ and the frequency threshold $\tau$

**Output:** All subgraphs $S$ of $G$ such that $s_G(S) \geq \tau$

1. $\text{result} \leftarrow \emptyset$
2. Let $fEdges$ be the set of all frequent edges of $G$
3. foreach $e \in fEdges$ do
   4. $\text{result} \leftarrow \text{result} \cup \text{SUBGRAPHEXTENSION}(e, G, \tau, fEdges)$
   5. Remove $e$ from $G$ and $fEdges$
4. return $\text{result}$

**Algorithm 2: CSP-based Frequent Subgraph Mining**

we have $s_G(S_1) \geq 3$ since all domains contain at least three valid assignments (more specifically, the domains of variables $v_1$, $v_2$ and $v_3$ are $\{u_1, u_5, u_6\}$, $\{u_3, u_4, u_8\}$ and $\{u_4, u_3, u_9\}$ respectively).

### 3.3.2 Frequent Subgraph Mining

We now apply the CSP model presented in Section 3.3.1 to solve the frequent subgraph mining problem (Problem 1). We start by presenting Algorithms **FREQUENTSUBGRAPHMINING** and **SUBGRAPHEXTENSION** that are used in many related methods to generate candidate subgraphs [19, 6] and are illustrated here for completeness. Then, we consider methods to measure the number of appearances (frequency) of these subgraphs. Algorithm **ISFREQUENTCSP** shows how we may address frequency evaluation without computing and storing all intermediate results. Algorithm **ISFREQUENTHEURISTIC** offers a heuristic approach and Algorithm **ISFREQUENT** supplements it with optimizations that highly improve performance. The frequent pattern embedding mining problem (Problem 2) is discussed in Section 3.4.

**FREQUENTSUBGRAPHMINING** is described in Algorithm 2. It starts by identifying a set $fEdges$ that contains all frequent edges (i.e., with frequency greater than or equal to $\tau$) in the graph. Based on the anti/monotone property, only these edges may participate in frequent subgraphs. For each frequent edge, **SUBGRAPHEXTENSION** is executed. The
Algorithm: \texttt{SubgraphExtension}

\textbf{Input:} A subgraph $S$ of a graph data $G$, the frequency threshold $\tau$ and the set of frequent edges $fEdges$ of $G$

\textbf{Output:} All frequent subgraphs of $G$ that extend $S$

1. $result \leftarrow S$, $candidateSet \leftarrow \emptyset$
2. \textbf{foreach} edge $e$ in $fEdges$ and node $u$ of $S$ \textbf{do}
3.  \hspace{1em} if $e$ can be used to extend $u$ then
4.  \hspace{2em} Let $ext$ be the extension of $S$ with $e$
5.  \hspace{2em} if $ext$ is not already generated then $candidateSet \leftarrow candidateSet \cup ext$
6. \textbf{foreach} $c \in candidateSet$ \textbf{do}
7. \hspace{1em} if $s_G(c) \geq \tau$ then
8. \hspace{2em} $result \leftarrow result \cup \texttt{SubgraphExtension}(c, G, \tau, fEdges)$
9. \textbf{return} $result$

\textbf{Algorithm 3:} Subgraph Extension

details of $\texttt{SubgraphExtension}$ is given in Algorithm 3. This algorithm takes as input a subgraph $S$ and tries to extend it with the frequent edges of $fEdges$ (Lines 2-5). All applicable extensions that have not been previously considered are stored in $candidateSet$. To exclude already generated extensions (Line 5) we adopt the $DFScode$ canonical form as in GSPAN [19]. Then, $\texttt{SubgraphExtension}$ (Lines 6-8) eliminates the members of $candidateSet$ that do not satisfy the support threshold $\tau$ since according to the anti-monotone property, their extensions are also infrequent. Finally, $\texttt{SubgraphExtension}$ is recursively executed (Line 8) to further extend the frequent subgraphs.

According to Proposition 2, a subgraph $S$ is frequent in $G$ (i.e., $s_G(S) \geq \tau$) if there exist at least $\tau$ nodes in each domain $D_1, \ldots, D_n$ that are valid variable assignments (i.e., are part of a solution) for the corresponding variables $v_1, \ldots, v_n$. To evaluate frequency, we may use $\texttt{ISFrequentCsp}$ (Algorithm 4) that returns $true$ iff $S$ is a frequent subgraph of $G$. Initially, $\texttt{ISFrequentCsp}$ enforces node and arc consistency [120]. Node consistency excludes unqualified nodes from the domains (like nodes with different labels or with lower degree) and arc consistency ensures the consistency between the assignments of two variables. Specifically, for every constraint $C(v, v')$, arc consistency ensures that for every node in the domain of $v$ there exists a node in the domain of $v'$ satisfying $C(v, v')$. 
Algorithm: ISFREQUENTCsp

Input: Graphs $S$ and $G$ and the frequency threshold $\tau$
Output: $true$ if $S$ is a frequent subgraph of $G$, $false$ otherwise

1. Consider the subgraph $S$ to graph $G$ CSP
2. Apply node and arc consistency
3. If the size of any domain is less than $\tau$ then return $false$
4. foreach solution $Sol$ of the $S$ to graph $G$ CSP do
   5. Mark all nodes of $Sol$ in the corresponding domains
   6. If all domains have at least $\tau$ marked nodes then return $true$
7. return $false$ // Domain is exhausted

Algorithm 4: Subgraph Evaluation based on CSP

The AC-3 algorithm [121] is used to ensure efficient and effective application of arc consistency. It progressively applies arc consistency on each domain. If one of its neighbors change, AC-3 revises that domain again. This process repeats until no more domain pruning can be applied. The literature discusses many advanced techniques for solving CSP. We use basic CSP approaches, other advanced techniques are beyond the scope of this work. If, after node and arc consistency enforcement, the size of a domain is smaller than $\tau$ the algorithm returns $false$ (Line 3). Following, ISFREQUENTCsp considers every solution $Sol$ and marks the nodes assigned to variables to the corresponding domains (Line 5). If all domains have at least $\tau$ marked nodes then (according to Proposition 2) $S$ is frequent in $G$. Otherwise, ISFREQUENTCsp continues with the following solution.

Complexity. Let $N$ and $n$ be the number of nodes of graph $G$ and subgraph $S$ respectively. The complexity of FREQUENTSUBGRAPHMINING is determined by the complexity of SUBGRAPHEXTENSION and ISFREQUENTCsp. The former computes all subgraphs of $G$, which takes $O(2^{N^2})$ time. The latter evaluates frequency which is reduced to the computation of subgraph isomorphisms (a well-known NP-hard problem) and takes $O(N^n)$ time. Overall, the complexity of the mining process is $O(2^{N^2} \cdot N^n)$ time which is exponential in the problem size. Thus, it is of crucial importance to devise appropriate heuristics and optimizations that improve execution performance. Several works study the subgraph generation process and propose techniques that significantly improve performance [19, 6].
These techniques are implemented in Algorithm \textsc{SubgraphExtension}. In the following section, we consider the optimization of Algorithm \textsc{IsFrequentCsp} that computes subgraph isomorphisms.

3.3.3 Optimizing Frequency Evaluation

Algorithm \textsc{IsFrequentCsp} naively iterates over the solutions of the subgraph $S$ to graph $G$ CSP trying to find $\tau$ valid assignments for every variable. To guide this search process, we propose \textsc{IsFrequentHeuristic}; the heuristic approach illustrated in Algorithm 5. Intuitively, the algorithm considers each variable at a time and searches for $\tau$ valid assignments. If these are found, it moves to the next variable and repeats the process. In more details, \textsc{IsFrequentHeuristic} starts by enforcing node and arc consistency. Then, the algorithm considers every variable and counts the valid assignments in its domain (stored in variable $\text{count}$). If, during the process, any variable domain remains with less than $\tau$ candidates, then the subgraph cannot be frequent, so the algorithm returns $\text{false}$ (Line 6 and 14). To count the valid assignments, \textsc{IsFrequentHeuristic} iterates over all nodes $u$ in the domain $D$ of a variable $x$ and searches for a solution that assigns $u$ to $x$. If the search is successful then $\text{count}$ is incremented by 1, and the process continues to the next node in $D$ until the number of valid assignments ($\text{count}$) becomes $\tau$, in which case the algorithm proceeds to the next domain (Line 13). On the other hand, if search is unsuccessful then $u$ is removed from $D$ and the algorithm continues with the next node in $D$. Updating $D$ may trigger new inconsistencies in other domains, thus, arc consistency (Line 5) is checked again. \textsc{IsFrequentHeuristic} also implements the following optimization. Assume that for a domain $D$ a solution was found for some node $u \in D$. Then, $\text{count}$ is incremented by 1 and all nodes (including $u$) that belong to this solution are marked in the respective domains (Line 10). Hence, if these nodes are considered in a later iteration of the algorithm, they are recognized as already belonging to a solution (Line 8). By doing this, any further search is precluded.
Algorithm: ISFREQUENTHEURISTIC

Input: Graphs \( S \) and \( G \) and the frequency threshold \( \tau \)
Output: true if \( S \) is a frequent subgraph of \( G \), false otherwise

1. Consider the subgraph \( S \) to graph \( G \) CSP
2. Apply node and arc consistency
3. \textbf{foreach} variable \( v \) with domain \( D \) \textbf{do}
   4. \hspace{1em} \( \text{count} \leftarrow 0 \)
   5. \hspace{1em} Apply arc consistency
   6. \hspace{1em} \textbf{if} the size of any domain is less than \( \tau \) \textbf{then return} false
   7. \hspace{1em} \textbf{foreach} element \( u \) of \( D \) \textbf{do}
      8. \hspace{2em} \textbf{if} \( u \) is already marked \textbf{then} \( \text{count}++ \)
      9. \hspace{2em} \textbf{else if} a solution \( \text{Sol} \) that assigns \( u \) to \( v \) exists \textbf{then}
          10. \hspace{3em} Mark all values of \( \text{Sol} \) in the corresponding domains
          11. \hspace{3em} \( \text{count}++ \)
      12. \hspace{2em} \textbf{else} Remove \( u \) from the domain \( D \)
      13. \hspace{2em} \textbf{if} \( \text{count} = \tau \) \textbf{then} Move to the next \( v \) variable (Line 3)
   14. \hspace{1em} \textbf{return} false // Domain is exhausted and \( \text{count} < \tau \)
15. \textbf{return} true

Algorithm 5: Heuristic-based frequent Subgraph Mining

Algorithm 6 illustrates ISFREQUENT that enhances ISFREQUENTHEURISTIC through several optimizations that significantly improve execution performance. ISFREQUENT uses three novel optimizations, namely, Push/down pruning, Lazy search and Unique labels. Finally, ISFREQUENT specializes, for frequent mining, Decomposition pruning and Automorphisms, that are known to speed up search [122] and frequent subgraph mining [37] respectively. In the sequel, we present the optimization techniques according to their execution order in the ISFREQUENT algorithm.

Push-down pruning. The subgraph generation tree is constructed by extending a parent subgraph with one edge at a time. Since the parent is a substructure of its children, those assignments that were pruned from the domains of the parent, cannot be valid assignments for any of its children. For example, Figure 3.3a illustrates a part of a subgraph generation tree consisting of subgraph \( S_1 \) which is extended to \( S_2 \), \( S_3 \) and then to \( S_4 \) (via \( S_2 \)). Assume that when considering subgraph \( S_1 \), ISFREQUENT excludes elements \( a_3, b_1 \), and
Algorithm: ISFREQUENT

**Input:** Graphs \( S \) and \( G \) and the frequency threshold \( \tau \)

**Output:** true if \( S \) is a frequent subgraph of \( G \), false otherwise

Consider the subgraph \( S \) to graph \( G \) CSP and apply node and arc consistency

// Push-down pruning

foreach edge \( e \) of \( S \) do

Let \( S/e \) be the graph after removing \( e \) from \( S \)

Remove the values of the domains in \( S \) that correspond to invalid assignments of \( S/e \)

// Unique labels

if \( S \) and \( G \) satisfy the unique labels optim. conditions then

if the size of any domain is less than \( \tau \) then return false

else return true

// Automorphisms

Compute the automorphisms of \( S \)

foreach variable \( x \) and its domain \( D \) do

\( \text{count} \leftarrow 0 \), \( \text{timedoutSearch} \leftarrow \emptyset \)

if there is an automorphism with a computed domain \( D' \) then

\( D \leftarrow D' \) and move to the next \( x \) variable (Line 9)

Apply arc consistency

if the size of a domain is less than \( \tau \) then return false

// Lazy search

foreach element \( u \) of \( D \) do

if \( u \) is already marked then \( \text{count}++ \)

else

Search for a solution that assigns \( u \) to \( x \) for a given time threshold

if search timeouts then Save the search state in a structure \( \text{timedoutSearch} \)

if a solution \( S\text{ol} \) is found then

\( \text{count}++ \)

else Remove \( u \) from the domain \( D \) and add \( u \) to the invalid assignments of \( D \) in \( S \)

if \( \text{count} = \tau \) then Move to the next variable (Line 9)

// Resume timed-out search if needed

if \( |\text{timedoutSearch}| + \text{count} \geq \tau \) then

// Decompose

Decompose graph \( S \) into a set of graphs \( \text{Set} \) that contain the newly added edge

foreach \( s \in \text{Set} \) do

Remove invalid assignments of \( s \) from the respective domains of \( S \)

foreach \( t \in \text{timedoutSearch} \) do

Resume search from the saved state \( t \)

if a solution \( S\text{ol} \) is found then

Mark all values of \( S\text{ol} \) to the corresponding domains

\( \text{count}++ \)

else Remove \( u \) from the domain \( D \) and add \( u \) to the invalid assignments of \( D \) in \( S \)

if \( \text{count} = \tau \) then Move to the next variable (Line 9)

return false // Domain is exhausted and \( \text{count} < \tau \)

return true

Algorithm 6: Optimized Frequent Subgraph mining

\( a_3 \) from the domain of variables \( v_1, v_2, \) and \( v_3 \) respectively (depicted by light gray ovals in Figure 3.3b). This information can be pushed down such that \( a_3,b_1,a_3 \) are also pruned from all descendants of \( S_1 \). This happens recursively; for instance, the assignments pruned because of \( S_2 \) are depicted by dark gray dotted ovals.
Figure 3.3: (a) Construction of the subgraph tree. (b) Variables and domains of the corresponding subtrees. Marked nodes represent the pruned assignments which are pushed down the tree.

The same substructure may also appear in subgraphs that do not have an ancestor/descendant relationship. In the example of Figure 3.3, $S_4$ is not a descendant of $S_3$; however, both contain substructure $A - B - A - C$. Since $S_3$ and $S_4$ are in different branches, pushing down the pruned assignments is not applicable. Instead, we use a hash table to store the pruned assignments of previously checked subgraphs. The hash key is the $DFScode$ canonical representation of $S_3$ [19]. When $S_4$ is generated, the hash table is searched for matching substructures. If one is found, the corresponding invalid assignments are pruned from the domains of $S_4$. ISFREQUENT applies this optimization (Lines 2-4) using the invalid assignments populated while searching for valid nodes (Lines 23 and 33).

Saving the invalid assignments of subgraphs results in a significant performance gain for the following two reasons.

- Subgraphs (like $S_4$) take advantage of the respective pruning of smaller subgraphs (like...
to prune invalid assignments. Thus, the domains of the subgraph variables are reduced avoiding the expensive search procedure (Lines 18 and 29). In many cases, a subgraph may be eliminated without search. For instance, in Figure 3.3, assuming that \( \tau = 3 \), \( S_4 \) can be eliminated, because there are only two valid assignments of variable \( v_1 \) remaining in its domain.

- This domain reduction also speeds up the search process since it highly depends on the domain size. For instance, in Figure 3.3, assuming that \( \tau = 2 \), when considering variable \( v_1 \), the search space has a size of \( 2 \cdot 2 \cdot 3 \cdot 4 = 48 \) combinations (bottom of Figure 3.3b), while without using this optimization the respective search space size is \( 5 \cdot 3 \cdot 5 \cdot 6 = 450 \) combinations.

To perform push-down pruning, Line 3 constructs \( O(n^2) \) subgraphs \( S/e \) by removing an edge from \( S \), (\( n \) is the number of nodes in \( S \)) and uses a hash lookup to remove the invalid assignment (Line 4). Thus, the overall complexity is \( O(n^2) \) time.

**Unique labels.** In the case of data graphs with a single label per node and subgraphs having a tree-like structure and unique node labels, the following optimization can be applied:

**Proposition 3.** Let \( G \) be a graph with a single label per node, \( S(V_S, E_S, L_S) \) be a subgraph of \( G \), \( S \)'s underlying undirected graph is a tree, and all of its node labels are unique, i.e., \( L_S(v) \neq L_S(v') \) for all \( v \) and \( v' \) in \( V_S \) such that \( v \neq v' \). To calculate \( s_G(S) \) directly, it suffices to consider the \( S \) to \( G \) CSP and refine the domains of variables by enforcing node and arc consistency.

**Proof:** Since each graph node has a single label and the query has unique labels, no node can appear in more than one domain. For any \( S \), we will use induction to prove that each value \( N \) in each domain of \( S \) (after applying the node and arc consistency constraints) is part of a valid solution. Let \( Q \) be a copy of \( S \) where all of \( S \)'s directed edges are replaced with undirected ones. \( Q \) is connected, undirected, and acyclic, therefore it is a tree. Let \( Q \) be rooted at the node corresponding to \( N \)'s domain.
• For $Q$ with $height = 1$, $N$ is guaranteed to be part of a valid solution (by definition of the node and arc consistency constraints and by considering the fact that the same node cannot appear in other domains).

• For $Q$ with $height = R$, let $T$ be a subgraph of $S$ and its underlying undirected graph is a subtree of $Q$ sharing the same root but with $height = R - 1$. Let $L$ be the set of $T$’s leaf nodes and assume that $T$ has a solution. $Q$ is composed of $T$ and the set of trees $Z$ with height equals one (or zero) each rooted at a distinct node from $L$. Since each element in $Z$ has a solution in $G$, and each solution joins with $T$’s solution only by its corresponding root in $Z$, hence, a valid solution for $S$ exists.

Note that the final step cannot be applied when the underlying undirected graph $Q$ contains a cycle. For example if $S$ is an undirected triangle of three nodes labeled $(A, B, C)$ and the data graph $G$ is undirected and contains six nodes forming a cycle: $(A, B, C, A, B, C)$. When considering the $S$ to $G$ CSP after enforcing node and arc consistency the count $s_G(S)$ is 2, but, the correct result is 0.

Example 2. Consider the subgraph $DB−IR$ and the graph $G$ of Figure 3.1. Let $v_1$ (resp. $v_2$) be the variable that corresponds to nodes labeled with $DB$ (resp. $IR$). The initial domains are $D_{v_1} = D_{v_2} = \{u_0, \ldots, u_9\}$. After applying node and arc consistency we have $D_{v_1} = \{u_1, u_5, u_6\}$ and $D_{v_2} = \{u_0, u_3, u_4, u_8\}$ which encodes the actual isomorphisms of the subgraph to graph $G$.

If the above conditions hold (Line 5), GRAM1 uses the current domain sizes to directly decide whether $S$ is frequent or not (Lines 6-7). The overall process can be performed in $O(n)$ time.

Automorphisms. Automorphism is an isomorphism of a graph to itself. Automorphisms appear because of symmetries. Following [37], such symmetries in the subgraph can be used to prune equivalent branches and reduce the search space. For example, consider subgraph $S$ of graph $G$ presented in Figure 3.4; $S$ has automorphisms. To determine if $S$
Figure 3.4: Automorphisms. (a) Input graph $G$. (b) Subgraph $S$ and its valid assignments.

is frequent in $G$, while iterating over the domain of $v_1$, ISFREQUENT finds the assignment $(v_1, v_2, v_3) = (u_1, u_4, u_2)$ to be a solution (i.e., an isomorphism of $S$ to $G$). Due to the symmetry of the subgraph $S$, assignment $(v_1, v_2, v_3) = (u_2, u_4, u_1)$ is also a solution. The benefits of this observation are twofold. First, we may identify the valid assignments of a variable more efficiently. Second, when we compute all valid assignments of a variable (i.e., $v_1$) we also compute the valid assignments for its symmetric counterpart (i.e., $v_3$).

ISFREQUENT detects automorphisms in Line 8. This requires $O(n^n)$ time where $n$ is the number of nodes in subgraph $S$. In practice, despite the exponential worst-case bound, the cost of automorphisms is very low since the size of subgraph $S$ is negligible compared to the size of the graph $G$.

**Lazy search.** Intuitively, to prove that a partial assignment does not contribute to any valid solution, the search algorithm has to exhaust all available options; a rather time consuming process. Thus, if a search for a solution that pertains to a specific partial assignment takes a long time, then this is probably because the partial assignment cannot contribute to a complete valid assignment. To address such cases, initially ISFREQUENT searches for a solution only for a limited time threshold (Line 18). The intuition of the optimization is that other assignments may produce much faster results that will help indicate if the subgraph is frequent ($s_G(S) \geq \tau$). In such a case, the result of the timed out search would be irrelevant, hence, there is no reason to waste time in further search. Nevertheless, this cannot guarantee that a timed out partial assignment will not eventually be essential for
proving the frequency of the subgraph. Thus, if search is timed out, the algorithm stores the search state in the \textit{timedoutSearch} set of nodes with incomplete check. These searches will only be resumed when the non-timed out cases are not sufficient to show that a subgraph is frequent. More specifically, timed-out searches are considered if after the time limited search, \( \text{count} < \tau \) and \( \text{count} \) plus the size of \textit{timedoutSearch} (i.e., the number of timed out searches) surpasses the threshold \( \tau \) (Line 25). Only then, the algorithm resumes each timed out search \( t \in \text{timedoutSearch} \) from its saved state but without a time-out option until enough assignments are found to prove frequency (Line 34). Note that, if necessary, \texttt{IsFrequent} eventually searches the entire search space for each variable to provide the exact solution.

The complexity of Lazy search (Lines 15–24) can be done in \( \mathcal{O}(N) \) time (note that the search of Line 18 takes constant time since it is performed for a specific time frame).

**Decomposition pruning.** The final optimization is performed in Lines 26 and 27. At this point, the algorithm is about to resume the timed out searches. To reduce the problem size, the algorithm decomposes the input subgraph \( S \) into a set of distinct subgraphs \( \text{Set} \). Recall

![Diagram](image-url)
that algorithm SUBGRAPH EXTENSION extends subgraphs by adding an edge \( e \) from the set of frequent edges \( fEdges \). Set \( \text{Set} \) is constructed by removing one edge at a time from \( S \) and adding to \( \text{Set} \) the connected component that includes edge \( e \). Any other decomposition has already been considered by the Push-down pruning optimization. Finding and removing invalid assignments from the domains of the elements of \( \text{Set} \) is a much easier task because they are smaller than the original subgraph \( S \).

For example, consider Figure 3.5. Subgraph \( S \) extends \( S' \) with edge \( C - K \) and, thus, it is decomposed into \( \text{Set} \) that contains subgraphs \( S_1 \) to \( S_3 \). Let us assume that the variable corresponding to the new node labeled with \( K \) is \( v_k \) and the initial domain of \( v_k \) contains values \( k_1 \) to \( k_7 \). Further, assume that using subgraphs \( S_1, S_2 \) and \( S_3 \) we can exclude values \( \{k_1, k_5\}, \{k_2, k_6\} \) and \( \{k_3\} \) respectively. The decomposition optimization removes all these values from the domain of \( v_k \), therefore, it only contains the values \( k_4 \) and \( k_7 \).

Decomposition pruning can be done in \( O(n^2) \). Resuming timed-out searches (Lines 28-34) requires solving a CSP on \( n - 1 \) variables with domain of size \( N \) and can be done in \( O(N^{n-1}) \) time.

**Complexity analysis of IsFREQUENT.** Let \( N \) and \( n \) be the number of nodes in \( G \) and \( S \) respectively. Push-down pruning, unique labels and automorphisms can be done in \( O(n^2) \), \( O(n) \) and \( O(n^n) \) respectively. Subgraph size is negligible in comparison to the data graph size, and thus these procedures are not expensive. IsFREQUENT applies arc consistency, lazy search and resumes timed-out search that can be done in \( O(Nn) \), \( O(N) \) and \( O(N^{n-1}) \) respectively. Thus, the complexity of IsFREQUENT is determined by the resumed timed-out searches. More specifically, if \( p \) is the possibility expressing that a node in a domain of a variable is valid, then to find the required \( \tau \) valid assignments we need to consider \( \tau/p \) nodes and solve \( \tau/p \) CSPs of size \( n - 1 \) for each one of the \( n \) variables. In total, the complexity bound is \( O(n \cdot \tau/p \cdot N^{n-1}) \).
3.4 GRAM1 Extensions

Generalization to pattern mining. Section 3.3 models the subgraph isomorphism problem (Definition 2) as a subgraph to graph CSP (Definition 6). Similarly, a pattern embedding $\phi$ (Definition 5) can be mapped to a CSP by replacing Condition 3c of Definition 6 with the following condition:

3c) $\Delta(x_v, x_{v'}) \leq \delta$, for every $x_v, x_{v'} \in \mathcal{X}$ such that $(v, v') \in E_P$ (where $\Delta$ is the distance metric and $\delta$ is the distance threshold).

Whenever it is clear from the context, we use $v$ to refer to a node of the pattern and $x_v$ to refer to the corresponding variable of the CSP as we do in the following example.

**Example 3.** Considering Figure 3.2, for $\delta = 0.3$, the pattern $P_1$ of graph $G$ CSP is defined as follows:

$$
\left( (v_1, v_2, v_3), \{\{u_0, \ldots, u_9\}, \ldots, \{u_0, \ldots, u_9\}\},
\begin{aligned}
&\{ v_1 \neq v_2 \neq v_3, L(v_1) = DM, L(v_2) = IR, L(v_3) = DB, \\
&\Delta(v_1, v_2) \leq 0.3, \Delta(v_2, v_3) \leq 0.3, \Delta(v_1, v_3) \leq 0.3 \}
\end{aligned}
\right)
$$

The notations for a solution (Proposition 1) and valid (or invalid) assignments (Definition 7) are easily extended to support pattern to graph CSPs. For instance, assignment $(v_1, v_2, v_3) = (u_7, u_8, u_6)$ is a solution of the CSP of Example 3 and a pattern embedding of $P_1$ to $G$. Moreover, $v_2 = u_3$ is a valid assignment while $v_2 = u_0$ is invalid (and thus, cannot be extended to a solution).

**Proposition 4.** Let $(\mathcal{X}, D, C)$ be the pattern $P$ to graph $G$ CSP. The MNI support of $P$ in $G$ satisfies $\tau$, i.e., $\sigma_G(S) \geq \tau$, iff every variable in $\mathcal{X}$ has at least $\tau$ distinct valid assignments (i.e., embeddings of $P$ in $G$).

Continuing Example 3, we have $\sigma_G(P_1) \geq 2$ since all domains contain at least two valid assignments (the domains of variables $v_1, v_2$ and $v_3$ are $\{u_2, u_7\}$, $\{u_3, u_8\}$ and $\{u_1, u_6\}$).
Table 3.1: Definitions of the anti-monotonic structural constraints for pattern $P$, implemented in CGRAIMI

| $|V_P| \leq \alpha$ | Number of nodes should not exceed $\alpha$ |
|---------------------|---------------------------------------------|
| $|E_P| \leq \alpha$ | Number of edges should not exceed $\alpha$ |
| $\max(\text{degree}(V_P)) \leq \alpha$ | The maximum node degree is $\alpha$ |

Table 3.2: Definitions of the anti-monotonic semantic constraints for pattern $P$, implemented in CGRAIMI

| $(\forall v \in V_P)(L(v) \in \mathcal{L})$ | $P$ contains only labels from $\mathcal{L}$ |
| $(\forall v \in V_P)(L(v) \notin \mathcal{L})$ | $P$ does not contain any label from $\mathcal{L}$ |
| $(\forall v, v' \in E_P)(L(v, v') \in \mathcal{E})$ | $P$ contains only edges from $\mathcal{E}$ |
| $(\forall v, v' \in E_P)(L(v, v') \notin \mathcal{E})$ | $P$ does not contain any edges from $\mathcal{E}$ |
| $(\neg \text{subgraph}(P', P))$ | Pattern $P$ must not contain a specific subgraph $P'$ |
| $(\forall v \in V_P)(\text{count}(L(v)) \leq \alpha)$ | A node label cannot appear more than $\alpha$ times in $P$ |

To address the frequent pattern mining problem (Problem 2), we can also employ Algorithms ISFREQUENTHEURISTIC and ISFREQUENT, with the following additional preprocessing step. For each frequent node, we precompute the set of nodes that are reachable within distance $\delta$. We run a distance-bound Dijkstra algorithm from each frequent node to find the shortest path to the reachable nodes, where the path distance is defined by the distance function $\Delta$. The Dijkstra algorithm terminates when the distance of the shortest path exceeds $\delta$. All optimizations of Subsection 3.3.3 apply directly in this setting as well. To avoid confusion, we use GRAIMI for the subgraph mining problem and $\text{GRAIMI}(\delta)$ for the pattern mining problem.

**User-defined constraints.** Typically, frequent patterns show interactions between nodes bearing the same label. For instance, in citation graphs, most collaborations are among authors working in the same field. In many applications, interactions among nodes of different types (like interdisciplinary collaborations) are more interesting and important [123]. To allow the user to focus on the interesting patterns, we developed CGRAIMI, a version of GRAIMI that supports two types of user-defined constraints: (a) **Structural**, such as “the number of vertices in pattern $P$ should be at most $\alpha$” and (b) **Semantic**, such as “$P$ must not contain specific labels”. 
Although not a requirement, it is desirable that the user-defined constraints are anti-monotonic. In such cases, the constraints can be pushed down in the subgraph extension search tree to early prune large parts of the search space, thus accelerating the process. Tables 3.1 and 3.2 present a set of useful structural and semantic anti-monotonic constraints that are supported by CGRAI.

**Approximate mining.** Frequent subgraph mining is a computationally intensive task since it is dominated by the NP-complete subgraph isomorphism problem. Thus, its performance is prohibitively expensive when applied to large graphs. Motivated by this, we introduce AGRAMI, an approximate version of our framework, which is able to scale to larger graphs.

To maintain the quality of results, AGRAMI does not return any infrequent pattern (i.e., does not have false positives), although it may miss some frequent ones (i.e., may have false negatives). To achieve this, we modify the way ISFREQUENT handles time-outs (Line 18) as follows: we set the time-out to occur after $f(\alpha)$ iterations of the search. If a solution is found before this time-out, the *count* is updated as normal. On the other hand, if a time-out occurs, it is assumed that the search was unsuccessful. If enough time-outs occur during the search of a specific domain such that its *count* remains less than $\tau$, the pattern is considered to be infrequent. Parameter $f(\alpha) = \alpha^n \prod_{i=1}^{n} |D_i| + \beta$, where $\beta$ is a constant, $D_i$ are the domains of the variables, $n$ is the number of variables and $0 < \alpha \leq 1$ is a user-defined approximation parameter. $\prod_{i=1}^{n} |D_i|$ grows exponentially; thus it has to be bounded by an exponential weight $\alpha^n$. Increasing $\alpha$ decreases the approximation error at the expense of longer execution time. When $\alpha = 1$, AGRAMI becomes equivalent to GRAMI.

In contrast to the existing approximate approaches [48, 50, 49, 51, 52], AGRAMI may miss some frequent subgraphs, but the returned results do not have false positives.

### 3.5 Experimental Evaluation

In this section, we experimentally evaluate GRAMI and its extensions. For comparison, we have implemented GROWSTORE, which follows a pattern *grow-and-store* approach [6,
Table 3.3: Datasets and their characteristics

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Nodes</th>
<th>Distinct node labels</th>
<th>Edges</th>
<th>Density</th>
</tr>
</thead>
<tbody>
<tr>
<td>Twitter</td>
<td>11,316,811</td>
<td>100</td>
<td>85,331,846</td>
<td>Dense</td>
</tr>
<tr>
<td>Patents</td>
<td>2,745,761</td>
<td>418</td>
<td>13,965,409</td>
<td>Medium</td>
</tr>
<tr>
<td>Aviation</td>
<td>101,185</td>
<td>6,173</td>
<td>133,087</td>
<td>Sparse</td>
</tr>
<tr>
<td>MiCo</td>
<td>100,000</td>
<td>29</td>
<td>1,080,298</td>
<td>Dense</td>
</tr>
<tr>
<td>CiteSeer</td>
<td>3,312</td>
<td>6</td>
<td>4,732</td>
<td>Medium</td>
</tr>
</tbody>
</table>

19]. **GROWSTORE** uses the original code of GSPAN [19] and takes advantage of all of its optimizations. The only difference is that GROWSTORE, similar to **GRAMi**, uses the efficient \( MNI \) metric. Both GROWSTORE and GRAMi are completely memory based. All experiments are conducted using Java JRE v1.6.0 on a Linux (Ubuntu 12) machine with 8 cores running at 2.67GHz with 192GB RAM and 1TB disk. Our experimental machine used an exotic memory size to accommodate the memory requirements of GROWSTORE; GRAMi may also run on ordinary machines with 4GB RAM for all datasets but Twitter.

**Datasets.** We experiment on several different workload settings by employing the following real graph datasets; their main characteristics are summarized in Table 6.1.

**Twitter** \(^1\). This graph models the social news of Twitter and consists of \( \sim 11 \)M nodes and \( \sim 85 \)M edges. Each node represents a Twitter user and each edge represents an interaction between two users. The original graph does not have labels, so we randomly added labels to the nodes. The number of distinct labels was set to 100 and the randomization follows a Gaussian distribution.

**Patents.** This dataset models U.S. patents’ citations and consists of a directed graph with \( \sim 3 \)M nodes and \( \sim 14 \)M edges. Each node represents a patent and each edge represents a citation. The graph is maintained by the National Bureau of Economic Research [124]. As a preprocessing step, we remove all unlabeled nodes.

**MiCo.** This dataset models the co-authorship information collected by Microsoft and consists of an undirected graph with 100K nodes and \( \sim 1 \)M edges. Nodes represent authors and are labeled with the author’s field of interest. Each edge represents a collaboration between

\(^1\)socialcomputing.asu.edu/datasets/Twitter
two authors and is labeled with the number of co-authored papers. MiCo is the graph resulted from crawling the computer science collaboration graph from Microsoft academic research service\(^2\).

CiteSeer\(^3\). CiteSeer represents a directed graph consisting of \(\sim 3\)K publications (nodes) and \(\sim 4\)K citations between them (edges). Each node has a single label representing a Computer Science area. Each edge has a label (0 to 100) that measures the similarity distance between the corresponding pair of publications.

Aviation\(^4\). This dataset contains a list of records extracted from the aviation safety database and was used in [51, 6] for evaluation. Each record corresponds to an event and has several attributes (like event type, location, flight condition). This information is represented by a graph having two types of nodes and edges. The first type of nodes represents the events (and are labeled with the IDs of the event) while the second represents attribute values (and are labeled with the actual value). The first type of edges links events and is labeled with their relationship (e.g., near to) while the second type links events with attribute values and is labeled with the attribute name. Aviation consists of 100K nodes and 133K edges. Note that Aviation is a fundamentally different dataset when compared with the previous ones. The Aviation graph has on average one edge per node, thus, it is very sparse. Also it has a very large number of distinct node labels.

**Metrics.** The support threshold \(\tau\) is the key evaluation metric as it determines when a subgraph or a pattern is frequent. Decreasing \(\tau\) results in an exponential increase in the number of possible candidates and thus exponential decrease in the performance of the mining algorithms. For a given time budget, an efficient algorithm should be able to solve mining problems for low \(\tau\) values. When \(\tau\) is given, efficiency is determined by the execution time.

To evaluate a result set, we consider the number and the maximum size of subgraphs/patterns in the set. Obviously, these values should be as large as possible.

---

\(^2\)academic.research.microsoft.com

\(^3\)cs.umd.edu/projects/linqs/projects/lbc

\(^4\)ailab.wsu.edu/subdue
Computing frequent subgraphs. Initially, we consider Problem 1 that mines frequent subgraph isomorphisms. Figure 3.6 shows the performance of GROWSTORE and GRAMI on Twitter, Patents, MiCo and Aviation datasets. The number of results (intermediate and actual) grows exponentially when the support threshold $\tau$ decreases. Thus, the running time of all algorithms also grows exponentially. Unlike GROWSTORE, GRAMI does not need to enumerate all intermediate results, thus, it is more efficient. Our results indicate that GRAMI outperforms GROWSTORE by at least two orders of magnitude for Patents and MiCo datasets and by at least an order of magnitude for Twitter and Aviation datasets. For the larger datasets (Twitter and Patents) and for the lower $\tau$ (3K and 65K respectively), GROWSTORE was not able to produce results even when it was allotted two orders of magnitude more time than GRAMI.
Figure 3.7: (a) Memory requirements for GRAMI and GROWSTORE and (b) Using MIS metric

**Memory requirements.** Figure 3.7a illustrates the memory requirements for GROWSTORE and GRAMI for the MiCo dataset. Since GROWSTORE needs to store all intermediate results, it consumes about an order of magnitude more memory. For $\tau = 10$, 400 the size of the intermediate results exceed the available memory (192GB), and hence GROWSTORE crashes. For this frequency, there is an increase in the number of the frequent subgraphs and thus an exponential increase in the number of intermediate candidates that need to be stored and checked for frequency. This trend also appears for the other datasets. GRAMI on the other hand is not affected by the increase in the output size. Most of the memory GRAMI uses, is required for the storage of the input graph $G$. The most costly data structure of ISFREQUENT is the hash table used by push-down pruning, but, still it does not exceed 2% for the overall required memory. Also the space needed to store timed-out searches (set timedoutSearch) was never above 1% of the total memory. For all our experiments, GRAMI could also be executed in machines with memory size of 4GB except for Twitter.

**Using MIS metric.** In this experiment, we compare GROWSTORE$^{MIS}$ the original version of GROWSTORE that uses the MIS metric with GRAMI$^{MIS}$, the modified version of GRAMI that also supports MIS. For the Aviation dataset, GRAMI$^{MIS}$ takes slightly more time than GRAMI while GROWSTORE$^{MIS}$ could not produce results even if it was alloted three orders of magnitude more time than GRAMI$^{MIS}$. Interestingly, GROWSTORE$^{MIS}$ cannot produce
Figure 3.8: Performance evaluation for mining frequent patterns in CiteSeer dataset comparing between GROWSTORE and GRAMI(\(\delta\)) where \(\delta\) is the distance threshold results in reasonable time even for the much smaller Citeseer dataset. To achieve a comparison, we have constructed a new dataset by randomly sampling 1400 edges from the Citeseer dataset. The results are illustrated in Figure 3.7b. Clearly, GRAMI\(^{MIS}\) outperforms GROWSTORE\(^{MIS}\) by up to three orders of magnitude.

**Computing frequent patterns.** We now consider Problem 2 that mines frequent pattern embeddings. We evaluate the performance of GROWSTORE and GRAMI(\(\delta\)) for several values of the distance threshold \(\delta\). We use the CiteSeer dataset and distance function \(\Delta_h(u, v)\) defined as the number of hops in the shortest path that connects \(u\) and \(v\). For GRAMI(\(\delta\)), we test on two different distance thresholds namely one and four. Intuitively, for \(\delta = 1\) (respectively \(\delta = 4\)) two pattern nodes that are connected with an edge may be matched with two graph nodes that are one hop (respectively four hops) away. GROWSTORE can only find
Figure 3.9: Comparing (a,c) the minimum support threshold and (b,d) the maximum number of frequent patterns that can be achieved within an allotted time budget. For (a,b) we used GRAMI(δ) and for (c,d) we use CGRAMI(δ) constrained to reject patterns with more than four nodes with the same label matches that are only one hop away. Thus, only GROWSTORE and GRAMI(1) are directly comparable since they both compute the same results. As shown in Figure 3.8a, GRAMI(1) is an order of magnitude faster than GROWSTORE (note the logarithmic scale). As expected GRAMI(4) computes more and larger patterns than GROWSTORE and GRAMI(1) (Figures 3.8b and 3.8c). An example of a frequent pattern discovered by GRAMI is illustrated on the right of Figure 3.8d and contains five nodes involving three different Computer Science areas. To compare, GROWSTORE computes the three nodes patterns at the left of Figure 3.8d that involve one and two areas. To compute these results, GRAMI(4) takes more time than GRAMI(1) but is still faster than GROWSTORE.

To further illustrate the benefits of GRAMI(δ) we have conducted another set of experi-
ments (Figure 3.9). The aim of the experiments is to illustrate the properties of the patterns that can be generated within a specific time budget. Figures 3.9a,b, consider the Citeseer dataset with the distance function $\Delta_h$ and compare between GROWSTORE, GRAMI(1) and GRAMI(4). Specifically, Figure 3.9a shows the minimum support threshold $\tau$ that can be achieved, when the above algorithms are allotted a time budget that ranges from one to five seconds (lower is better). For this budget range, Figure 3.9b illustrates the number of result patterns (higher is better). In both cases, GRAMI(1) and GRAMI(4) accomplish lower thresholds and result in more patterns than GROWSTORE.

**CGRAMI: User-defined constraints.** CGRAMI supports the addition of constraints on the returned results (Section 3.4). Using these constraints, the focus can be on more interesting pattern types like the ones that show interactions between nodes of a different type. To evaluate CGRAMI, we use the experimental setting of Figure 3.9a,b. The only difference is that we now use CGRAMI($\delta$) with a constraint that does not allow more than four nodes with the same label in a pattern. The corresponding results are illustrated in Figure 3.9c,d and are directly comparable to Figure 3.9a,b. In every case and within the same time budget allowed for both GRAMI and CGRAMI, CGRAMI results in a significantly lower minimum support threshold $\tau$ and significantly larger frequent patterns set. For instance, for the Citeseer dataset with a time budget of three seconds, CGRAMI(1) achieves a three times lower threshold and almost three times more patterns than GRAMI. Additionally, CGRAMI generates patterns having about three times more label interactions than GRAMI.

**AGRAMI: Approximate mining.** AGRAMI, which offers approximate subgraph and pattern mining (Section 3.4), can be tuned by the approximation parameter $\alpha$, $0 < \alpha \leq 1$ (value one means no approximation). Figure 3.10 illustrates the performance of GRAMI and AGRAMI for several values for the $\alpha$ parameter in the Patents and MiCo datasets. We evaluate two parameters, execution time and recall, i.e., the percentage of subgraphs returned by AGRAMI with respect to the actual complete set of frequent subgraphs. For the Patents dataset, the performance gain is significant, nearly an order of magnitude for
both $\alpha = 2 \cdot 10^{-5}$ and $\alpha = 3 \cdot 10^{-5}$. For $\alpha = 3 \cdot 10^{-5}$ the recall is always 100% (i.e., AGRA MI provides all subgraphs) except for $\tau = 63,600$ that is 95%. For $\alpha = 2 \cdot 10^{-5}$ the recall is always over 90%. For the MiCo dataset, the performance gain is significant, nearly an order of magnitude when $\alpha = 4 \cdot 10^{-4}$ and nearly two orders of magnitude when $\alpha = 2 \cdot 10^{-4}$. Interestingly, the recall is always 100%.

**Optimizations.** This experiment demonstrates the effect of the optimizations discussed in Subsection 3.3.3 on mining the different datasets. A summary is illustrated in Figure 3.11.

For the MiCo dataset, the most effective optimization is *Push-down pruning* (denoted by Pruning in Figure 3.11a) that achieves an improvement of up to two orders of magnitude. Following that, are the *Lazy search* and the *Decomposition pruning* optimizations, both are combined and denoted by Lazy in Figure 3.11a. The two optimizations accomplish an improvement of up to an order of magnitude. Last comes the *Automorphism* and *Unique labels* optimizations that achieve only 4% improvement, since most of the frequent subgraphs in the MiCo dataset neither have automorphisms nor unique labels. For presentation clarity in Figure 3.11a, we do not illustrate the results of the last two optimization methods. A similar trend also applies to Patents and Citeseer datasets (Figures 3.11b and 3.11c).

For the Aviation dataset (Figure 3.11d), a different optimization trend is noticed since this dataset is fundamentally different than MiCo Patents and Citeseer. In this case, the
most effective optimization is Unique labels (denoted by Unique in Figure 3.11d). As discussed earlier, the Aviation dataset is extremely sparse and has a very large number of distinct node labels, thus, the Unique label optimization is very effective. In contrast to the previous cases, all other optimizations do not offer any improvement and are not illustrated.

Comparison with subgraph isomorphism techniques. To address the frequent data mining problem, we may also employ subgraph isomorphism techniques [82]. For comparison, we have implemented GGQL; a modified version of GRAMI that replaces ISFREQUENT with a frequency evaluation function based on GRAPHQL [80]; one of the fastest state-of-the-art subgraph isomorphism techniques [82]. Clearly, as illustrated in Figure 3.12,
Figure 3.12: Performance comparison between GRAMI and GGQL; a modified version of GRAMI that replaces ISFREQUENT with a counting function based on GraphQL.

GRAMI outperforms GGQL by at least three times and up to more than an order of magnitude. This is easily justifiable since GRAMI uses several optimizations and visits only the necessary nodes in the input graph to solve the frequent subgraph mining problem.

### 3.6 Conclusion

Many important applications, such as bioinformatics, social networks, personalized advertisement, recommendation systems and security, depend on graph mining. This chapter introduces GRAMI; a versatile algorithm for discovering frequent patterns in a single large graph. The modeling of the frequency evaluation operation as a constraint satisfaction problem is the crux idea of GRAMI. We complement this idea with a set of optimizations that allows for the efficient performance of GRAMI. We also implement a version that supports structural and semantic constraints and an approximate version that scales to larger graphs. Our experimental results with real datasets demonstrate the effectiveness of GRAMI which is up to two orders of magnitude faster than existing approaches while discovering larger and more interesting frequent patterns.
Chapter 4

ScaleMine: Scalable Parallel Frequent Subgraph Mining

4.1 Introduction

The majority of existing FSM solutions are either serial [19, 6, 35]; or multi-threaded within a single machine [21, 22]. Since FSM is known to be an expensive graph operation, these systems cannot scale to large graphs. Recent systems implement parallel FSM for a single large graph on shared-nothing architectures. MRPF [28] and MRSUB [31] are based on Map-Reduce [125], ParGraph [60] is based on MPI, whereas Arabesque [30] and Pegi [29] are implemented on top of vertex-centric Pregel-like [126] frameworks. Table 4.1 shows the maximum reported graph size of each system along with the maximum number of workers. The table also includes our system, GRAMI, the state-of-the-art serial solution. Compared to GRAMI, the parallel systems do not support significantly larger graphs, although they utilize much more resources. For example, Arabesque with 20 workers reports 13x performance gain [30] against GRAMI. Similarly, Pegi with 20 workers reports only 10x performance gain [29]. Obviously, even after considering the overheads of Map-Reduce and Pregel, scalability of the aforementioned systems is severely limited.

To understand the scalability problem, we analyze the behavior of the FSM algorithm. FSM first searches the input graph for frequent edges, which are the simplest frequent subgraphs. For each such subgraph, all possible extensions are generated, and the algorithm calculates their frequencies. The process of generating larger candidate subgraphs is repeated recursively, until no further frequent subgraphs are identified. FSM is easily parallelizable: let a master node generate a pool of candidate subgraphs. Each available
Table 4.1: Existing parallel systems and the largest graph used in their evaluations (ScaleMine is the proposed system).

<table>
<thead>
<tr>
<th>System</th>
<th>Max #Nodes</th>
<th>Max #Edges</th>
<th>Max #Workers</th>
</tr>
</thead>
<tbody>
<tr>
<td>MRPF [28]</td>
<td>300</td>
<td>2,000</td>
<td>48</td>
</tr>
<tr>
<td>MRSUB [31]</td>
<td>81,306</td>
<td>1,768,149</td>
<td>80</td>
</tr>
<tr>
<td>ParGraph [60]</td>
<td>2,020,188</td>
<td>7,905,260</td>
<td>128</td>
</tr>
<tr>
<td>Arabesque [30]</td>
<td>2,745,761</td>
<td>13,965,409</td>
<td>640</td>
</tr>
<tr>
<td>Pegi [29]</td>
<td>N/A</td>
<td>100,000,000</td>
<td>50</td>
</tr>
<tr>
<td>GramI [35]</td>
<td>11,316,811</td>
<td>85,331,846</td>
<td>(serial) 1</td>
</tr>
<tr>
<td>ScaleMine</td>
<td>56,692,400</td>
<td>1,046,834,000</td>
<td>8,192</td>
</tr>
</tbody>
</table>

worker receives a candidate and checks whether it is frequent; the result is sent back to the master. We call this parallel version Baseline. All candidates are sent at least once to the workers, and the master receives the evaluation results. Although the number of candidates can be huge, the communication overhead is minimal compared to the processing overhead required to evaluate those candidates. Thus, communication cost is low and therefore Baseline is almost embarrassingly parallel.

Figure 4.1 shows strong scalability for Baseline compared to the ideal scalability (see Section 4.4 for the experimental setup). A strong scalability experiments measures how a solution benefits from increasing the available resources while fixing the problem size. An ideal solution should require 50% less time when the available number of workers is doubled, and this is represented by “Ideal Scalability” in Figure 4.1. Observe that, irrespective of the number of workers\(^1\), the total response time of Baseline remains almost constant (shown as flat line). Intuitively, FSM generates a search tree of candidate subgraphs. The shape of the tree is highly irregular and the computational cost of deciding whether a subgraph is frequent varies considerably among subgraphs. Due to the irregular space, there are periods when the task pool at the master is empty, so some workers may stay idle. Also, the variability of the computational cost generates stragglers. The result is a highly imbalanced system. In our example, only two workers were each utilized more than 35%, whereas the utilization of the majority was below 0.2%.

\(^1\)Each worker is assigned to a separate CPU core.
A possible solution is intra-task parallelism: frequency computation for each candidate subgraph is divided into subtasks that run in parallel. We call this version TaskDivision. For the above-mentioned experiment, TaskDivision achieves almost perfect load balance with roughly 100% utilization for every worker. Unfortunately, TaskDivision is not embarrassingly parallel: communication and synchronization cost is substantial and straight-forward pruning optimizations become extremely costly. Figure 4.1 demonstrates that TaskDivision can be more than an order of magnitude slower than Baseline.

In this chapter, we propose ScaleMine; a scalable parallel system for FSM in a single large graph. The main contribution of ScaleMine is the introduction of a two-phase approach consisting of an approximate phase followed by an exact one. First, ScaleMine executes a novel approximate FSM algorithm that uses sampling to: (i) identify a set of subgraphs that are frequent with high probability; (ii) collect various statistics about the input graph; and (iii) build a model to predict the execution time for each subgraph frequency calculation task. The approximate phase is fast and comprises a small fraction of the total computational cost.

In the subsequent exact phase, ScaleMine implements a hybrid of Baseline and TaskDivision. The master maintains a pool of tasks, each corresponding to frequency calculation

Figure 4.1: Strong scalability of Baseline and TaskDivision. Total response time in seconds. Twitter dataset; $\tau=160K$ (see Section 4.4 for details).
of a candidate subgraph. Available workers request tasks, calculate the frequency and return the result to the master. In contrast to existing approaches, if the task pool runs low, the master fills it with subgraphs identified in the approximate phase, meaning that workers do not stay idle. Furthermore, subgraphs identified in the approximate phase are, with high probability, frequent; therefore the algorithm prunes infrequent subgraphs early and does not waste time at wrong regions of the search space. For each task, the master uses the cost model built in the approximate phase, to simulate various scenarios of intra-task parallelism, and decides whether it pays off to split expensive tasks into subtasks, in order to improve the load balance while reducing response time. Finally, statistics collected during the previous phase, are used by the workers. Frequency calculation can be mapped to a constraint satisfaction problem, where the order of constraint checking impacts the execution cost. Workers use the statistics to generate low cost execution plans.

ScaleMine uses the MNI metric to compute the support of candidate subgraphs and it follows the approach proposed by GAMI as it is shown to efficiently handle larger graphs compared to other algorithms. Note that the output of ScaleMine is an exact solution. The approximate phase is used to improve load balance, provide information that guides the search faster towards the correct solution, and decide the tasks for which intra-task parallelism is beneficial. Table 4.1 compares ScaleMine to state-of-the-art systems: ScaleMine supports 10x larger graphs (i.e., 1B edges), scales to 12x more workers (i.e., 8,192 cores on a Cray XC40) and runs orders of magnitude faster. In summary, our contributions are:

- We develop ScaleMine, a scalable parallel FSM system for single large graphs.
- We propose a novel two-phase approach, consisting of an approximate phase that collects information and an exact phase that exploits the collected information to generate fast execution plans with good load balance.
- We conduct extensive experimental evaluation on a modest cluster and on a high-end Cray XC40 supercomputer, using large real datasets. Our results show that ScaleMine outperforms the state-of-the-art solutions by at least an order of magnitude in terms of
supported graph size, number of workers, and execution time.

The rest of this chapter is organized as follows. Section 4.2 introduces the approximation phase, then section 4.3 discusses the exact phase. Experimental evaluation is discussed in Section 4.4. Finally, section 4.5 concludes.

### 4.2 Approximate Phase

Balancing the workload is essential for the scalability of parallel solutions. Achieving good load balance is easier when the search space is known in advance; unfortunately, this is not the case for FSM. We tackle the load balancing problem by a novel two-phase approach. The first phase builds an approximation of the search space and collects statistics. The second phase, which returns the exact results, uses the approximation to balance the load among workers and optimize their execution plans.

An effective approximation of the FSM search space should be: (i) representative: the predicted search space should represent the exact one within acceptable accuracy; (ii) efficient: the approximation phase should have minimal overhead; (iii) informative: the approximation should be accompanied with statistics that can be used to optimize the performance of the exact phase. Although approximate FSM is not a new idea, none of the existing approximation techniques meets the aforementioned requirements. That is, they either return an insignificant fraction of the search space; do not generate the required statistics; or have a high computational cost.

ScaleMine introduces a novel approximation phase, based on sampling, that satisfies the aforementioned requirements. The approximate phase resembles the typical FSM algorithm. It begins by finding small frequent subgraphs, which are then extended into larger ones by adding edges. We employ an adaptive sampling approach (Section 4.2.1) to estimate fast and accurately whether a subgraph is frequent. During this process, ScaleMine collects useful statistics for each candidate subgraph (Section 4.2.2). These statistics are later utilized to optimize the exact FSM phase.
4.2.1 Sampling-based Subgraph Evaluation

For a candidate subgraph \( S \), a typical FSM algorithm populates the sets: \( MNI_{col}(v_1) \), \( MNI_{col}(v_2) \), \ldots, \( MNI_{col}(v_d) \) with valid nodes. Iterating over all nodes in each column is expensive since it involves subgraph isomorphism. Our approach randomly samples a small fraction of these nodes, and estimates the size of each column. Given an \( MNI_{col}(v_i) \), the process of validating each node resembles a binomial distribution. If the probability of success \( p_i \) is known, then the number of valid nodes in \( MNI_{col}(v_i) \) is \( \mu = N_i p_i \), where \( N_i \) is the number of nodes in the current column. Unfortunately, the value of \( p_i \) is unknown. \( p_i \) can be estimated by sampling a small number of nodes. Since the problem is to decide whether a candidate subgraph is frequent or not, we can relax the problem to estimating whether \( p_i \) is smaller or larger than \( p_i,\tau \), where \( p_i,\tau \) is called the \( \tau \)-probability of success and equals \( \tau / N_i \). Having \( p_i > p_i,\tau \) means that the number of valid nodes in \( MNI_{col}(v_i) \) is more than \( \tau \), and consequently \( MNI_{col}(v_i) \) is a valid column; otherwise, it is invalid.

ScaleMine employs the central limit theorem to estimate the probability that \( p_i \) is larger than \( p_i,\tau \). The theorem states that the distribution of the means of a large number of independent, identically distributed random variables is approximately normal, regardless of the underlying distribution [127]. For each \( MNI_{col}(v_i) \) belonging to a subgraph \( S \), \( k \) sets of \( n \) randomly selected nodes are sampled. The mean of each set is the number of valid nodes divided by \( n \). The means of the generated \( k \) sets constitute a normal distribution with mean \( \hat{\mu} = n \hat{p} \) and standard deviation \( \hat{\sigma} = \frac{\sigma}{\sqrt{n}} \), where \( \hat{p} \) is the probability of success estimated from the sampled nodes, and \( \sigma = \sqrt{n \hat{p}(1 - \hat{p})} \).

After generating the distribution, a vague, inconclusive, area is defined. Having a support threshold within this area means that the estimated support is not significantly different than the given threshold; therefore more sampling is needed to increase confidence. The vague area is bounded by:

\[
\text{low} = \hat{\mu} - (z\hat{\sigma}) \quad \text{and} \quad \text{high} = \hat{\mu} + (z\hat{\sigma})
\]
where $z$ is the area under the normal distribution curve for a specific confidence value. Having larger vague area requires more sampling and increases the accuracy of the decision; the trade-off is increased computational overhead.

We show in Figure 4.2 an example of a normal distribution generated by the sampling process, and mark three probability of success values $p_{\tau_1}$, $p_{\tau_2}$ and $p_{\tau_3}$, for different support thresholds $\tau_1$, $\tau_2$ and $\tau_3$, respectively. Note that, each $p_{\tau_i}$ is multiplied by $n$. Assume $\tau_2$ is the support threshold; the sampled nodes have larger mean than $p_{\tau_2}$, so the corresponding $MNI_{col}$ is predicted to be valid. On the other hand, for $\tau_3$, the corresponding $MNI_{col}$ is predicted to be invalid since $p_{\tau_3} > \mu$. An interesting case is when $\tau_1$ is used: $p_{\tau_1}$ is inside the vague area and the difference between $\mu$ and $p_{\tau_1}$ is not significant. Thus, we cannot make a confident decision, and more sampling is required. In general, for subgraphs with support values close to $\tau$, more samples are evaluated until $p_r$ moves outside of the vague area. In some cases, $p_r$ never gets out of the vague area, so we set a maximum number of samples to stop the process regardless of the obtained accuracy.

We summarize the proposed sampling technique in Algorithm 7. The list of domains are created for each node $v \in S$ (Line 1). For each domain, sampling is conducted, and the number of valid and invalid nodes are counted. This process iterates until the sample size is met (Line 15). Then the mean value $m$ is computed for each set of samples, and it is added to the distribution $T$. Support is estimated once the number of sampled nodes
Algorithm: **SAMPLINGBASEDEVAL**

**Input:** \( G \) the input graph, \( \tau \) support threshold, \( S \) Candidate Subgraph, \( maxS \) Maximum number of samples, \( minS \) Minimum number of samples, \( bSize \) sample size

**Output:** \( r \) the estimated support

1. \( D \leftarrow \text{CREATEDOMAINS}(G, S) \)
2. \( r \leftarrow 0 \)
3. **foreach** \( D_i \in D \) **do**
   4. \( nValid \leftarrow 0; \) \( totalValid \leftarrow 0; \) \( nInvalid \leftarrow 0 \)
   5. \( \text{counter} \leftarrow 0 \)
   6. \( P_\tau \leftarrow \tau / |D_i| \)
   7. **Reset distribution** \( T \)
   8. **while** true **do**
   9. \( u \leftarrow \text{GETRANDOMNODE}(D_i) \)
   10. \( b \leftarrow \text{ISVALID}(G, S, u, D_i) \)
   11. **if** \( b \) is true **then**
   12. \( nValid = nValid + 1 \)
   13. \( totalValid = totalValid + 1 \)
   14. **else** \( nInvalid = nInvalid + 1 \)
   15. **if** counter \((\text{mod} \ bSize)=0 \) **then**
   16. \( m \leftarrow \text{COMPUTE MEAN}(nValid, nInvalid) \)
   17. \( \text{Add} \ m \) to \( T \)
   18. **if** counter \( \geq \) minS **then**
   19. \( M \leftarrow \text{COMPUTE MEAN}(T) \)
   20. \( SD \leftarrow \text{COMPUTE SD}(T) \)
   21. **if** FINISHSAMPLING\((T, \tau, maxS)\) **then** break
   22. \( nValid \leftarrow 0 \)
   23. \( nInvalid \leftarrow 0 \)
   24. \( \text{counter} = \text{counter} + 1 \)
   25. \( \text{estimatedSize} \leftarrow (totalValid/\text{counter})*|D_i| \)
   26. **if** estimatedSize \( \leq \tau \) **then** \( r \leftarrow \text{estimatedSize} \)

**Algorithm 7:** Sampling-Based Subgraph Evaluation

meets the default minimum number of samples (Line 18). The mean and standard deviation are then computed for the distribution \( T \), which is assumed to follow a normal distribution according to the central limit theorem (Lines 19 and 20). FINISHSAMPLING (line 21) returns true if the estimated support is outside of the vague area, or when the maximum sample size (maxS) is reached.
4.2.2 Search Space Estimation

During the approximation phase, ScaleMine collects useful statistics for each candidate subgraph. We show in Section 4.3 how these statistics are utilized to achieve better performance during the exact phase. We show below the information collected during the approximation phase:

**Subgraph estimated support**: for each candidate subgraph, \( \hat{s}_G(S) \) is the value returned from Algorithm 7 which is an estimation of the exact value \( s_G(S) \).

**Subgraph evaluation time**: ScaleMine estimates the time required for the exact evaluation of a candidate subgraph as follows:

\[
\sum_{D_i \in D} \frac{\text{time}(D_i) \times |D_i|}{N_i}
\]

\( D \) is the set of all domains, \( \text{time}(D_i) \) is the time spent on evaluating the sampled nodes for domain \( D_i \), \( N_i \) is the number of sampled nodes and \( |D_i| \) is the domain size. We utilize this information to guide intra-task parallelism (see Section 4.3.2).

**Number of valid nodes per \( MNI_{col} \)**: besides having the estimated subgraph support, it is also important to know the estimated number of valid nodes per \( MNI_{col} \). This value is calculated in Line 25 of Algorithm 7. We use the estimated number of valid nodes per \( MNI_{col} \) for the early pruning in the exact phase (see Section 4.3.3). Note that we only store this information for approximated infrequent subgraphs because it only helps with evaluating infrequent subgraphs.

**Expected invalid column**: the estimated invalid column is the column that is predicted to have a number of valid nodes less than \( \tau \). The exact phase utilizes this information to optimize the execution plan. Note that we only store the invalid columns for only the approximated infrequent subgraphs.
4.3 Exact Phase

4.3.1 System Description

ScaleMine employs the master-worker paradigm on a multi-threaded shared-nothing environment. It uses the standard Message Passing Interface (MPI) for communication. Figure 5.2 shows the system architecture. The master receives the input graph and the user-defined support threshold $\tau$. The graph loader loads the input graph and dispatch it to the cluster nodes. Each node has a copy of the graph index, which is utilized by the workers (i.e., threads) running on that node; each core is assigned a single thread. Once the graph is loaded, ScaleMine starts its two-phase processing for finding the frequent subgraphs.

The first phase builds the approximate search space, which generates a pool of tasks, denoted by $P_{App}$. The pool stores both frequent and infrequent predicted subgraphs (i.e., approximations). The second phase handles the exact evaluation of FSM. Alongside $P_{App}$, ScaleMine uses a second pool of tasks, denoted by $P_{Ex}$, to store candidate subgraphs generated from exact evaluation. This phase starts by generating a set of tasks consisting of frequent vertices. Once they are evaluated, frequent subgraphs are added to the result set, expanded, and stored in the exact task pool $P_{Ex}$. The master prioritizes dispatching tasks from $P_{Ex}$ to available workers until it becomes empty. Statistics of each subgraph are also sent to the corresponding worker. If such statistics are not available for the subgraph, ScaleMine generates them on the fly by approximate evaluation of the subgraph. Dispatched tasks are prioritized by size; smaller subgraphs are processed first.

Due to the nature of FSM, the number of available tasks is small at the beginning and by the end of the evaluation process. Such behavior affects the scalability and the utilization of the available resources. To avoid having idle workers, ScaleMine dispatches tasks to idle workers from the pool of approximated subgraphs $P_{App}$ whenever $P_{Ex}$ is empty. These tasks are not random since they are generated by the approximation phase. As such, they are expected to be evaluated in future iterations. Instead of waiting for the exact evaluator
to produce them, they are evaluated ahead of time to benefit from the available resources. There is a probability that some of these tasks should not be evaluated at all, the approximation phase is accurate enough to minimize such cases.

Once a worker finishes its task, it sends the result back to the master and asks for more tasks. Then, the master updates the task pools ($P_{Ex}$ and $P_{App}$) and sends new tasks to available workers. Updating the task pool involves: (i) removing larger supergraphs from $P_{App}$ that contain a reported infrequent subgraph by the exact phase. (ii) Adding new subgraphs to $P_{Ex}$ by extending reported frequent subgraphs, and (iii) removing tasks from $P_{App}$ that match discovered subgraphs to avoid task duplication. ScaleMine explores the search space level by level, from smaller candidates to larger ones, until no more frequent subgraphs are found. ScaleMine incurs minimal communication overhead. Tasks along with their approximate statistical information are sent to workers, which report the computed support values back to the master. No synchronization data or embeddings list are communicated.

We discuss in the following subsections how ScaleMine exploits the knowledge collected during the approximate phase to provide a scalable FSM solution.
4.3.2 Subtasking

Reducing the number of idle workers by having enough tasks for all workers is not guaranteed to provide a balanced workload. Evaluating one subgraph can take significantly more time than another subgraph. Therefore, assigning an expensive task to a single worker would introduce a straggler worker which severely affects the load balance and hinders system scalability. To avoid this scenario, it is important to have coherent tasks; tasks that require almost the same processing time. ScaleMine utilizes the estimated evaluation time for each predicted subgraph to distinguish between expensive and light-weight tasks. Expensive tasks are divided into smaller subtasks which are evaluated by several workers while light-weight tasks are assigned to a single worker.

The goal of subtasking is to partition a task into $n$ subtasks; each subtask is a disjoint partition of the processing space. Partitioning can be either vertical or horizontal. Vertical partitioning assigns a different $MNI_{col}$ to different workers. Each worker will be responsible for evaluating its given column. This approach has three limitations: (i) usually the number of columns compared to the available workers is very small. Therefore, the maximum number of workers to be used is limited by the number of columns. (ii) Even if the number of columns is large enough, there is no need to evaluate all columns for candidate subgraphs that are indeed infrequent. As a result, a subset of the workers will end up doing useless work. (iii) Different columns have different execution overheads, which retracts to the first load imbalance problem.

In horizontal partitioning, graph nodes are partitioned among workers. Each worker, which has access to the whole input graph, is responsible for counting valid nodes only in its partition. A hash-based partitioning is a simple yet effective approach for distributing the workload. By opting for this partitioning strategy, enough subtasks are generated, and no extra overhead is required to process the needed columns. Thus, ScaleMine uses the horizontal task partitioning.

We now describe how ScaleMine decides the number of subtasks ($n$) that an expensive
task should be divided into. By having larger number of subtasks, more cores are utilized and load balance is enhanced. However, having more subtasks increases the processing overhead of each subtask since some pruning optimizations can not be utilized when the task is divided among multiple independent workers. Moreover, subtasking requires more synchronization overhead between the master and the involved workers. ScaleMine tries to find a near optimal value for $n$ which ensures both good load balance and minimal computation/communication overheads. ScaleMine follows a simulation-based approach that maximizes the load balance while minimizing the number of subtasks. This approach utilizes the statistics obtained from the approximation phase to simulate the runtime of different partitioning granularities. For each granularity, ScaleMine calculates the imbalance percentage as follows:

$$\lambda = \frac{L_{\text{max}}}{L} - 1$$

where $L_{\text{max}}$ is the maximum workload on any worker and $\bar{L}$ is the average workload over all workers. A smaller value of $\lambda$ translates to a more coherent distribution of the subtasks.

Task partitioning is based on a maximum subtask time $\theta$, if a task has a predicted time $T_p$ more than $\theta$, it is partitioned into $n = T_p/\theta$ subtasks. ScaleMine starts evaluating $\lambda$ without any task partitioning ($\theta=$maximum predicted task time), and keeps generating finer workloads by decreasing $\theta$, 10% after each iteration, until $\lambda$ becomes less than a given imbalance threshold.

### 4.3.3 Optimizations

ScaleMine utilizes some statistics collected during the approximation phase to optimize the performance of the exact phase. We highlight below each one of these optimizations. Notice that, all these optimizations are only applied for subgraphs predicted as infrequent.

**Early Pruning**: When evaluating an $MNI_{col}$, let $nV$ denote the number of the already found valid nodes and let $nR$ denote the number of remaining nodes. An important optimization
is to stop evaluating this \( MNI_{\text{cd}} \) when the stopping condition: \( nV + nR < \tau \) is met. In other words, the number of remaining nodes plus the valid ones is not enough to satisfy \( \tau \). It is easy to detect this case in a single-threaded solution [35] when the whole column is evaluated by a single task. However, this is quite challenging when the task is divided among multiple workers. For invalid subgraphs, ScaleMine employs a heuristic approach to decide when to stop evaluating a column. For a subtask \( i \), let \( nV_i \), \( nR_i \) and \( \beta_i \) be the number of already found valid nodes, the number of nodes to be evaluated and the percentage of work assigned to worker \( i \), respectively. The stopping condition can be re-written for each subtask \( i \) as: \( nV_i + nR_i < \tau \beta_i \). Interestingly, the predicted number of valid nodes, \( pV \), can be utilized to allow earlier break by modifying the stopping condition to: \( nR_i < (\tau - pV) \beta_i \).

That is, assuming a consistent distribution of valid nodes among subtasks, a break becomes earlier when the difference between \( \tau \) and \( pV \) becomes larger. In reality, subtasks tend to differ in the distribution of valid nodes. To accommodate for this, the following is added to the left-hand side of the stopping condition: \( nV - (pV \cdot \beta_i \cdot \alpha_i) \), where \( \alpha_i \) is the percentage of the so far progress of subtask \( i \). Once this condition is met, the remaining unprocessed nodes are treated as being valid nodes, giving an upper bound on the number of valid nodes. If this upper bound does not exceed \( \tau \), then for sure the exact value cannot exceed \( \tau \) too, and the column is reported as invalid and the whole subgraph as infrequent. For the case where the upper bound exceeds \( \tau \), the evaluation needs to be repeated after turning this optimization off. To minimize the chances of such case, we introduce \( m \); a constant value between zero and one (inclusive). Having a lower \( m \) delays the application of this condition. The final version of the stopping condition is:

\[
nV_i - (pV \cdot \beta_i \cdot \alpha_i) + nR_i < (\tau - pV) \cdot \beta_i \cdot m
\]

**Pruning expensive nodes:** Some graph nodes, especially in large dense graphs, are excessively expensive to evaluate. Similar to GRAM1, ScaleMine avoids evaluating these nodes
in favor of other light-weight nodes. Gram1 relies on a user-given threshold to identify expensive nodes. Instead, ScaleMine exploits the knowledge gained during the approximation phase to identify those nodes. During the approximate phase, for each \( MNI_{col} \) in each subgraph, ScaleMine maintains the average computation steps needed to evaluate the graph nodes. Then, during the exact phase, ScaleMine identifies expensive nodes as the nodes that need significantly more computation steps than their corresponding averages. Similar to the previous optimization, the costly nodes are assumed to be valid for support computation. Then, if the computed support is found to be larger than \( \tau \), full re-evaluation is required but with this optimization turned off.

**Starting with invalid columns:** For a subgraph that contains a predicted invalid column, ScaleMine starts the evaluation by inspecting the invalid column first, hoping that this column is really invalid and the whole subgraph being infrequent. As such, this optimization results in a lower cost execution plan by avoiding the overhead of evaluating other valid columns whenever an invalid column exists.

### 4.3.4 Correctness

ScaleMine is a two-phase solution; approximate and exact evaluation phases. ScaleMine starts the approximation phase to generate enough tasks and statistics that guide/help the exact evaluation phase. Notice that, the exact phase generates its own search space by starting from the frequent edges and extending subgraphs until no more frequent subgraphs are found. The final results of ScaleMine are produced by the exact phase. For a frequent subgraph which is not discovered by the approximate phase, ScaleMine will find it as it builds its own search space and exactly evaluates it. Sometimes, ScaleMine borrows some tasks from the pool of the approximation phase and gives it to idle workers. For a borrowed infrequent subgraph that was mistakenly considered as frequent by the approximation phase, ScaleMine exactly evaluates it. Consequently, ScaleMine is guaranteed to produce correct results in all cases.
4.4 Experimental Evaluation

In this section, we experimentally validate ScaleMine using large graphs. Specifically, we show the following: (i) ScaleMine significantly outperforms state-of-the-art FSM systems (Subsection 4.4.2). (ii) The effect of the proposed optimizations is significant (Subsection 4.4.3). (iii) The search space generated by the approximation phase achieves both high quality and minimal runtime overhead (Subsection 4.4.4). Finally, (iv) ScaleMine scales to thousands of computing cores on graphs with up to one billion edges (Subsection 4.4.5). In all experiments, we only report the total processing time for ScaleMine and its competitors which does not include the time needed for loading the graph into memory. As ScaleMine uses a randomization-based component (approximation phase), we run each experiment multiple times and report their average execution time.

4.4.1 Experimental setup

Datasets: We evaluated ScaleMine using four real graphs. Table 5.1 shows the statistics of each graph. MicoBig models the Microsoft co-authorship graph; nodes represent authors and labeled with their fields of interest. An edge represents a collaboration between two authors. To populate MicoBig, we crawled the computer science collaboration graph from academic.research.microsoft.com. Weibo [128] represents a microblogging network crawled from Sina Weibo.com, which allows users to follow each other. Each node represents a user and labeled with the city he lives in, while each edge represents a follower-followee relationship between two users. Twitter\(^2\) is the same data set used in Section 3.5. But, in this experiment we randomly assign node labels from a pool of 25 distinct labels using Gaussian distribution. Patents [124] is the same data set used in Section 3.5 but we use less number of categories to label graph nodes.

Implementation: ScaleMine is implemented in C++ and uses Message Passing Protocol

\(^2\)http://socialcomputing.asu.edu/datasets/Twitter
Table 4.2: Datasets and their characteristics

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Nodes</th>
<th>Edges</th>
<th>#node labels</th>
</tr>
</thead>
<tbody>
<tr>
<td>Patents</td>
<td>2,745,761</td>
<td>13,965,409</td>
<td>37</td>
</tr>
<tr>
<td>Twitter</td>
<td>11,316,811</td>
<td>85,331,846</td>
<td>25</td>
</tr>
<tr>
<td>Weibo</td>
<td>1,655,678</td>
<td>369,438,063</td>
<td>55</td>
</tr>
<tr>
<td>MicoBig</td>
<td>56,692,400</td>
<td>1,046,834,000</td>
<td>3,077</td>
</tr>
</tbody>
</table>

(MPI) for communication between the master and workers.

**Hardware Setup:** We used two hardware settings for our experiments. The first one is a local cluster of 16 machines each with 148GB RAM and two 2.1GHz AMD Opteron 6172 CPUs, each with 12 cores. The machines run a 64-bit 3.2.0-38 Linux Kernel and are interconnected by a 10Gbps Ethernet switch. We use this cluster for the comparison with existing FSM solutions. We also use a Cray XC40 supercomputer which has 6,174 dual sockets compute nodes based on 16 cores Intel processors running at 2.3GHz with 128GB of RAM. In this supercomputer, a hard limit of one day is set as a maximum processing time for any task, where jobs that exceed this limit are killed.

### 4.4.2 Comparison with Existing Systems

In this experiment, we compare ScaleMine to Gram1 and Arabesque [30], the state-of-the-art FSM systems. Since Gram1 is a single-threaded system, it under-utilizes the current multi-core architectures available in nowadays hardware. Arabesque is a generic distributed graph mining system that runs on a cluster of machines. This experiment is conducted on our local cluster of 16 machines.

Figure 4.4 shows how ScaleMine compares to the other systems using two datasets; Patents and Twitter. For medium sized graphs like Patents and Twitter, finding the frequent subgraphs is challenging which render the single machine approach; Gram1, unscalable. In figure 4.4(a), both ScaleMine as well as Arabesque significantly outperform Gram1 as they make better utilization of available resources. Moreover, ScaleMine outperforms Arabesque because it avoids storing and communicating intermediate results. Figure 4.4(b)
Figure 4.4: Performance of ScaleMine vs. existing FSM systems on a cluster of 16 machines (256 workers) using two datasets: (a) Patents and (b) Twitter

shows the results on the Twitter dataset. ScaleMine achieves up to two orders of magnitude better performance than GRAMI. For $\tau=160k$, GRAMI could not find the frequent subgraphs within two days. Also, Arabesque crashes for all support thresholds due to the storage and communication overheads associated with the excessive number of embeddings. Notice that for Patents, ScaleMine is only an order of magnitude faster than GRAMI although ScaleMine uses 256X more threads. For Patents, the distributed tasks are relatively small and the overhead of parallelizing these light tasks diminishes the benefits of parallelization. As a result, ScaleMine cannot benefit much from the available resources. On the other hand, for the Twitter tasks, which are more expensive, ScaleMine is more than 50X faster than GRAMI.

4.4.3 Optimizations

In this experiment, we measure the effect of each of the proposed optimizations using 512 computing cores on the supercomputer. We start from the baseline approach (Baseline), this approach applies exact evaluation, available tasks are dispatched to idle workers whenever possible. Then, we gradually apply each optimization. The first optimization is to build an approximate search space then utilize its pool of tasks to keep workers busy (Approx. Space). The second optimization is to divide expensive tasks, first by using the average predicted time (Task Division (Avg)), second by using our simulation-based task partitioning
Figure 4.5: Effect of ScaleMine’s optimizations using 512 cores on both Twitter ($\tau = 155k$) and Weibo ($\tau = 490k$, maximum size = five edges)

(Task Division (Sim)). Finally, we apply the early pruning optimization (Early pruning). A column marked with red "XX" indicates that the experiment did not finish within 24 hours.

Figure 4.5 shows the results on Twitter and Weibo datasets. Y-Axis shows the elapsed time in hours. Each bar represents the time spent after adding the corresponding optimization (the exact runtime is on top of each bar). For Twitter on $\tau=155k$, the baseline does not finish within one day due to the workload imbalance. We noticed in this experiment that a few cores were highly overloaded while the others were idle. When we introduce the search space approximation phase, the system finishes in 11 hours because it generates a pool of tasks to improve the utilization of workers. In other words, the system is able to complete the whole mining process significantly faster than the baseline. Utilizing the information collected during the approximation phase results in an order of magnitude improvement. With this optimization, the processing time for each candidate is estimated; hence expensive tasks are identified and divided among workers. With our workload simulation technique, we achieve a better performance by saving 700 seconds compared to using the average as a threshold. Finally, the early pruning strategy of ScaleMine improves system efficiency by finishing in 300 seconds less time.

A similar behavior is noticed in the Weibo dataset when using $\tau = 490k$ and limiting subgraph size to five edges. Notice that, utilizing the approximation phase and dividing the
tasks using the average time (Task Division (Avg)) does not help the system to finish within one day. On the other hand, utilizing the proposed simulation-based task division makes it practically feasible. Overall, the conducted experiments show the importance of all the optimizations proposed by ScaleMine.

### 4.4.4 Approximation Phase Performance

In this experiment, we measure the accuracy of the approximation phase by comparing the reported frequent patterns from the approximation phase to the exact frequent patterns reported at the end of the mining process. If a pattern is reported as frequent by the approximation phase and it is indeed frequent, we count this pattern as a true positive. If the approximation phase missed an actual frequent pattern, then we count it as a false negative. Finally, patterns found as frequent by the approximation phase but are not actually frequent are counted as false positives. We utilize the known F-measure metric for assessing the quality of our approximation phase.

Figure 5.5 shows the calculated F-measure for Patents and Twitter datasets. The X-axis represents different support values. For each dataset, the used support thresholds are the same thresholds used in Figure 4.4. For Twitter, ScaleMine maintains an F-measure $= 1$ for all the different support values. As for the patents dataset, ScaleMine achieves an
F-measure of more than 0.97. Achieving this high accuracy comes at a low computation cost compared to the total mining time. Figure 4.7 shows the time of the approximation phase alongside the time of the exact phase. In this experiment, we show the total time for each dataset using its lowest support threshold; i.e. we pick the most time-consuming mining process for each dataset. As shown in Figure 4.7, the approximation phase takes between 3% and 21% of the total execution time. Specifically, for larger graphs, with more expensive tasks, the percentage of approximation time becomes lower. This shows that our approximation phase can be used to support fast and accurate results as a standalone approximate FSM solution.

4.4.5 Scalability

We show in Figure 4.8 the scalability and speedup efficiency of ScaleMine using the four real graphs: Patents, Twitter, Weibo and MicoBig with $\tau = 15k, 155k, 460k$ and $8m$, respectively. For both Weibo and MicoBig, we set the maximum allowed frequent subgraph size to five edges. Figures 4.8(a) and 4.8(b) show the scalability and speedup efficiency for Patents and Twitter datasets when the number of workers (cores) ranges from 32 to 1024. ScaleMine achieves good speedup efficiency up to 512 cores for both datasets; 87% for Patents and 69% for the Twitter dataset. When the number of workers increases to 1024, the total time spent by ScaleMine decreased but it does not achieve good speedup effi-
Figure 4.8: Scalability and speedup efficiency of ScaleMine using different real datasets: Patents with $\tau = 15k$, Twitter with $\tau = 155k$, Weibo with $\tau = 460k$ and maximum subgraph size = five edges and MicoBig with $\tau = 8m$ and maximum subgraph size = five edges.

ciency; around 67% for Patents and 59% for Twitter. Such decrease in speedup efficiency is expected since subtasks become smaller and the parallelization overhead becomes relatively expensive. Figures 4.8(c) and 4.8(d) show the scalability and speedup efficiency for the largest two datasets; Weibo and MicoBig, starting from 512 to 8192 workers. We do not show the performance of ScaleMine using a lower number of cores as it takes significant time to finish. Similar to the last experiment, ScaleMine achieves good scalability and speedup efficiency for both datasets up to 4096 cores. After that, adding more workers does not significantly improve the performance which resulted in lower speedup efficiency.
4.5 Conclusion

In this chapter, we discussed ScaleMine, a scalable frequent subgraph mining system for a single large graph. ScaleMine is a dual-phase system that starts with generating a sample-based approximation of the search space. Then, it executes an exact evaluation phase that utilizes the approximation phase to achieve better load balance and efficient evaluation of the candidate subgraphs. Our results show that ScaleMine is at least an order of magnitude faster than state-of-the-art systems. Furthermore, ScaleMine scales to thousands of cores (i.e., 12x more than the competitors) and mine graphs with up to one billion edges (i.e., 10x more than the competitors).
Chapter 5

Pivoted Subgraph Isomorphism

5.1 Introduction

As discussed earlier, subgraph isomorphism is used for computing the support of candidate subgraphs. The processing overhead of subgraph isomorphism increases exponentially with the graph size. Moreover, the number of intermediate as well as final results of subgraph isomorphism is enormous even for moderate graph sizes. Recent subgraph isomorphism advancements [83, 84, 85] show significant improvement over prior solutions. But, they cannot deal with large graphs. The experimental evaluation, even for the state-of-the-art techniques, show results on small graphs using a maximum limit on the number of retrieved results. As such, using subgraph isomorphism for mining large graphs is prohibitively expensive.

Figure 5.1 shows $G$ as an example of a paper citation network. $G$ contains a list of authors (a), each author works for academia (A), industrial organizations (I), or both. Also, authors write papers (P) which cite other papers. Furthermore, figure 5.1 shows two example subgraphs; $Q_1$ and $Q_2$. Using subgraph isomorphism for calculating the support of each subgraph is unnecessarily expensive. For example, the $MNI$-based support of $Q_1$ is 1. To compute this value, a subgraph isomorphism algorithm needs to find all of the $^8P_3 = \frac{8!}{(8-3)!} = 336$ occurrences! Such overhead is unacceptable for larger graphs and queries. Based on our experiments (check section 5.5.2), for a relatively small graph with around 86K edges, we show that the average number of occurrences per query of size four edges is more than 26M.
In order to improve the performance of FSM, we propose to utilize a modified definition of the subgraph isomorphism problem. This problem definition is called Pivoted Subgraph Isomorphism (PSI) which is similar to the classical subgraph isomorphism but with a pivot query node that needs to be validated against the input graph. Given an input query graph with a specified pivot query node $q_i$, rather than finding all matches of a given query graph, the task of PSI is to find the set of graph nodes that match the pivot node and “satisfy” the given query graph. A graph node satisfies a query graph by being part of at least one embedding of this query in the input graph. Previous work has referred to the pivot node in the context of two different problems; once for mining the frequent neighborhood patterns [129] and the other for annotating functional residues in protein structures [130]. For the same experiment where subgraph isomorphism finds 26M occurrences for small queries, the number of nodes that qualify the pivoted query graph is only 140. This example highlights the possibility of significant improvement in FSM once PSI is used instead of subgraph isomorphism.

Other than FSM, PSI is also required in a variety of graph analysis applications. For example, it can be used for mining frequent neighborhood patterns [129], where the goal is to find the set of frequent patterns that originates from graph nodes with a specific label. Furthermore, figure 5.1 shows an example of interesting queries; $Q_1$ and $Q_2$; $Q_1$ searches...
for papers that are cited at least three times, and $Q_2$ searches for papers that resulted from the collaboration between at least two authors from academia and industry. The interesting node, which is $v_0$ for each query, is the pivot node, and is highlighted with a circle.

In order to support these applications, subgraph isomorphism picks a start query node and a list of candidate graph nodes. Then, it proceeds to find all occurrences of each candidate graph node. Indeed, a straightforward optimization to this solution would be to stop looking for matches once an occurrence is found for each candidate graph node. Nevertheless, such optimization cannot significantly improve these solutions since subgraph isomorphism is designed and optimized to find all occurrences of a given query. For example, for the majority of subgraph isomorphism techniques [81, 83], the query node with the highest selectivity is chosen as the start query node in the search plan. Based on this optimization, the number of intermediate results is significantly reduced. For the PSI problem, it is not possible to use this optimization since the start query node has to be the pivot node which is given in the query subgraph. Furthermore, the state-of-the-art solutions utilize complex data structures to efficiently compile the set of all matches [84, 85]. The utilization of such data structures is useless in the PSI case since only one solution is needed per every candidate graph node.

This section discusses the pivoted subgraph isomorphism problem and proposes an efficient solution for it. The PSI problem is transformed into a classification task, and a solution based on an optimistic-pessimistic approach is proposed. Given a list of candidate graph nodes, each node either satisfies the query graph or does not satisfy it. The first type of nodes is called “valid nodes” and the later is called ’invalid nodes’. For valid nodes, an optimistic function is designed to quickly find an occurrence of the query subgraph that contains that node. For invalid nodes, a pessimistic function is used. This function utilizes an aggressive pruning technique that quickly discards graph nodes that cannot match the query subgraph. Furthermore, a classifier is proposed to benefit from these two functions. This classifier is designed to predict which nodes match the query and which do not. For
a node that is predicted to be valid, the optimistic matching function is used. For the other case, the pessimistic function is used for invalid nodes. The classification step, the optimistic method, and the pessimistic method are all based on an index that maintains information about the neighbors around each graph node. This index is created during a pre-processing phase.

The rest of this section is organized as follows. Section 5.2 formalizes the problem. In Section 5.3, we give an overview of the proposed system architecture. Section 5.4 presents PSI and its optimizations. Finally, Section 5.5 presents the evaluation.

### 5.2 Preliminaries

Subgraph isomorphism is known to be expensive since it requires finding all occurrences of a subgraph query. PSI is a more relaxed definition, this relaxation is achieved by focusing on finding at most one occurrence per each graph node. Below, we define the pivoted graph and the PSI problem:

**Definition 8.** A pivoted graph is a tuple $\bar{\delta} = \{ G, v_p \}$, where

- $G$ is a labeled graph
- $v_p \in V_G$ is called the pivot node of $\bar{\delta}$

**Definition 9.** A pivoted subgraph isomorphism of $\bar{\delta}_S = \{ S, v_p \}$ to $\bar{\delta}_G = \{ G, u_p \}$ where $S = (V_S, E_S, L_S), G = (V_G, E_G, L_G), v_p \in V_S$ and $u_p \in V_G$, is an injective function $M : V_S \to V_G$ satisfying

- $L_S(v) = L(M(v))$ for all nodes $v \in V_S$,
- $(M(u), M(v)) \in E_G$ and $L_S(u, v) = L(M(u), M(v))$ for all edges $(u, v) \in E_S$
- $M(v_p) = u_p$

The above definition is similar to the typical subgraph isomorphism definition except for the last point where the mapping of $v_p$ has to be $u_p$. Pivoted subgraph isomorphism is
NP-Complete [129]. The pivot node \((v_p)\) is the node of interest, and it is usually assigned by the user issuing the query. For the case where PSI is used for FSM, each MNI column is populated with valid graph nodes by solving a pivoted subgraph isomorphism query \(\delta=\{G, v_p\}\), where \(G\) represents a candidate subgraph and \(v_p\) is the subgraph node corresponding to that MNI column.

**Definition 10.** A pivoted subgraph check \(\ell(\delta_G, \delta_S)\) is a function that returns true if there exists a pivoted subgraph isomorphism between \(\delta_G\) and \(\delta_S\), otherwise it returns false.

The goal of pivoted subgraph check is to decide whether there exists at least one occurrence of \(S\) in \(G\) where \(v_p\) is mapped to \(u_p\). For the case when there exists an occurrence, \(u_p\) is called a valid node, otherwise it is called an invalid node.

**Problem 3.** Given a graph \(G\) and a pivoted subgraph \(\delta_S\), the goal of pivoted subgraph search is to find every node \(u \in G\) that pass the pivoted subgraph check \((\ell(\delta_G, \delta_S))\) when \(u\) is considered a pivot node of \(\delta_G\).

Problem 3 defines a problem that can be used in applications such as FSM and supports interesting queries as shown in Figure 5.1.

A graph node \(u\) can be identified by its neighbor nodes. Those neighbors are then used to identify whether \(u\) qualifies certain conditions; such as conditions provided by a query. The work in [131] proposes an approximate graph matching technique that represents a node using a list of its neighbor nodes within \(K\) hops. This list is then used to prune candidate nodes based on an approximate score. Tale [132] is another system that finds approximate matches of query subgraphs in an input graph. It relies on an index, called NH-Index, that maintains information about the direct neighbors for each graph node. This index is used to prune graph nodes that cannot match the corresponding query nodes. Compared with the previous technique and other indexing techniques that rely on indexing paths, trees or subgraphs, the size of NH-Index scales linearly with the graph size. In order to achieve good efficiency and focus on more important results, Tale order query nodes according to
their importance. For Tale, importance of a node is defined as its degree. Though, Tale can be extended to support more complex importance definitions. Later, Arijit et. al [133] proposed a graph index based on label propagation from neighbor nodes. Based on that label propagation, the graph structure around each node is approximately captured. This structure is reflected in the weights assigned to each label. In this work, we follow the later approach since it is able to capture better information about the neighborhood than the former techniques. Basically, a node $u$ is indexed with the list of weighted labels, those labels are propagated to $u$ from its neighbors. We call this list the neighborhood signature, and it is defined as follows:

**Definition 11.** Let a graph $G = (V, E, L)$ and $u \in V$. The neighborhood signature of $u$ within a distance $D$ is the list of pairs $NS_u^D = \{(l_1, w_1), (l_2, w_2), \ldots, (l_n, w_n)\}$, where $l_i \in L$ is a label, $w_i \in R$ is the weight of that label and $D$ is the maximum propagation depth.

Let $W_{NS_u^D}(l_i)$ refer to $w_i$; the weight of $l_i$ in $NS_u^D$. If there does not exist a pair with label $l_i$, $W_{NS_u^D}(l_i) = 0$. Label weights are propagated from nearby nodes, for each label its weight is calculated as follows:

$$W_{NS_u^D}(l_i) = \sum_{d=1}^{D} 2^{-d} \times C_u(l_i, d)$$  \hspace{1cm} (5.1)

Where $C_u(l_i, d)$ is the number of nodes with label $l_i$ and their shortest distance to node $u$ is $d$. To benefit from the neighborhood signature, we use the same maximum propagation depth for both query graphs and the input graph. Thus, we will omit the use of superscript $D$ where it is not needed.

Figure 5.1 shows an example of neighborhood signature. There are four different labels; \{p, a, A, I\}. Assuming that the graph is undirected and the used maximum propagation depth = 2, the signature of node $u_0$ is calculated as follows: (i) First, the label of $u_0$ is weighted as 1. Consequently, the list of weights from distance 0 = \{(p, 1), (a, 0), (A, 0), (I, 0)\}. (ii) Second, considering the first level of neighbor nodes, $u_0$ is connected to only two nodes
each with label \( p \). Therefore, the weight of \( p \) in this case is \( 2 \times 0.5 \). Consequently, the propagated weights from this level = \{(p, 1), (a, 0), (A, 0), (I, 0)\}. (iii) Using the second level of nodes with two hops away from \( u_0 \); i.e. \( u_1, u_4, u_5, u_6, u_7, u_8 \) and \( u_9 \). In this level there are four nodes with label “\( p \)” and three nodes with label “\( a \)”. As a result, the weight of “\( p \)” = \( 4 \times 0.25 = 1.0 \) and the weight of “\( a \)” = \( 3 \times 0.25 = 0.75 \). Consequently, the propagated weights from this distance = \{(p, 3), (a, 0.75), (A, 0), (I, 0)\}. Finally, \( NS_{u_0}^2 \) is the sum of the propagated weights across all levels, hence, \( NS_{u_0}^2 = \{(p, 3), (a, 0.75), (A, 0), (I, 0)\} \).

**Definition 12.** A signature \( NS_i \) satisfies another signature \( NS_j \) if for every \((l_i, w_i) \in NS_j\), there exists \((l_j, w_j) \in NS_i\) where \( l_i = l_j \) and \( w_i \leq w_j \).

In figure 5.1, \( u_3 \) has a neighborhood signature \( NS_{u_3} = \{(p, 3), (a, 0.75), (A, 0), (I, 0)\} \). For the query graph \( Q_1 \), \( NS_{v_0} = \{(p, 2.5)\} \). \( NS_{u_0} \) satisfies \( NS_{v_0} \) since the weight of label “\( p \)” in \( v_0 \) is less than that of \( u_3 \). As for \( u_1 \), the weight of label “\( p \)” is 2, which is less than the weight of “\( p \)” in \( v_0 \), which means that \( u_1 \) cannot satisfy \( v_0 \).

**Proposition 5.** if two nodes have the same neighbors, their neighborhood signatures are the same.

**Proposition 6.** Given the graph \( G \), a pivoted subgraph \( \partial = \{Q, v_p\} \) and a node \( u \in V_G \), if \( u \) is a valid node given \( \partial \), then \( NS_u \) must satisfy \( NS_v \).

**Proof.** The used approach is proof by contradiction. Suppose a graph node \( u \) is valid for the pivoted graph \( \partial = \{G, v\} \) and \( SN_u \) do not satisfy \( SN_v \). For \( SN_u \) to not satisfy \( SN_v \) requires the existence of at least one pair \((l, w_v) \in SN_v \) with no corresponding pair in \( SN_u \), or there exists a pair \((l, w_u) \in SN_u \) where \( w_u < w_v \). Consequently, the number of neighbor nodes of \( u \) with label “\( l \)” would be less than the neighbors of \( v \) with label “\( l \)”.

**Definition 13.** The satisfiability score \( SS(u, v) \) of the two nodes \( u \) and \( v \) is a measure of the likelihood of node \( u \) being valid for node \( v \), and is calculated as:

\[
SS = \frac{avg_{l \in L(v)}(NS_u(l)/NS_v(l))}{SS(u, v)}.
\]
In figure 5.1, the satisfiability score of nodes $u_5$ and $v_0$ of $Q_2$ is $SS(u_5, v_0) = 2.25$, whereas the satisfiability score of nodes $u_4$ and $v_0$ of $Q2 SS(u_4, v_0) = 1.5$. Satisfiability of $u_5$ is more than $u_3$ score, reflecting the fact that $u_5$ has higher connectivity to nodes that match with neighbors of $v_0$ in $Q_2$.

### 5.3 Pivoted Subgraph Isomorphism

#### 5.3.1 System Architecture

Figure 5.2 shows the architecture of the proposed solution for PSI. Given a pivoted subgraph query, PSI populates the list of input graph nodes that are candidate matches of the pivoted node. These candidate nodes are nodes with the same label as the pivoted node. Then, a small subset of these candidates is used to train a machine learning model. For each training node, PSI evaluates the pivoted subgraph isomorphism algorithm to find whether that node is valid or invalid. Then, PSI creates the training data composed of the feature vector of each training node as well as its class; valid or invalid. The neighborhood signature of each node is used as the feature vector; each label in the signature represents a feature, and the corresponding weight of that label is the feature weight.

For each of the remaining candidate graph nodes, PSI utilizes its feature vector and consults the machine learning model to predict whether this node is expected to be valid or invalid. If the model predicts that the candidate is a valid match, PSI uses an algorithm optimized for valid nodes. If the candidate is predicted to be invalid, PSI uses another algorithm which is designed for invalid nodes. Once the chosen algorithm finishes evaluating a candidate node, whether it is valid or invalid, the result is cached and used for next similar candidate nodes. This caching component allows for improving the classifier adaptively, alleviating the mistakes that were previously taken.

Two algorithms are designed for the pivoted subgraph isomorphism test. The first, which is called pessimistic (see Section 5.4.1), is more suitable for invalid nodes since it
is designed to prune the candidate graph nodes aggressively. The second, which is called optimistic (see Section 5.4.2), tries to reach a solution as quickly as possible by minimizing the number of traversed graph nodes. Hence, it is more suited for valid nodes. Both algorithms return the correct class of a node even if they are applied to an unfavored node (e.g., pessimistic applied to a valid node).

5.3.2 Data Pre-processing

The goal of the pre-processing step is to construct the neighborhood signature for each graph node. The whole graph is loaded into memory using the adjacency list representation. This list is implemented as a hashmap, where node IDs are the keys and the list of direct neighbors are the values.

We discussed in Section 5.2 how to calculate the neighborhood signature of each node. Starting from each graph node, we explore the labels of neighbors one hop at a time till the maximum propagation depth is met. The main limitation of this approach is that it is computationally expensive. Its computational complexity is $O(|N| \cdot |d|^D)$ where $|N|$ is
the number of nodes in the graph, \(|d|\) is the average degree, and \(D\) is the used maximum propagation depth.

A more efficient approach is based on matrix computations. First, PSI creates a matrix \(NS : N \times L\), where \(N\) is the number of graph nodes and \(L\) is the number of distinct node labels. \(NS^0\) is initialized as follows:

\[
NS^0(n, l) = \begin{cases} 
1 & l \in L(n) \\
0 & \text{otherwise} 
\end{cases} \quad (5.2)
\]

Then, for the subsequent \(D\) iterations, the neighborhood signature of each node is updated as follows:

\[
NS^i(n) = NS^{i-1}(n) + \frac{1}{2} Adj(n) * NS^{i-1} \quad (5.3)
\]

where \(Adj(n)\) is the adjacency matrix row that corresponds to node \(n\). The proposed matrix-based method for calculating neighborhood signatures has a computational complexity of \(O(|N|.|d|.|D|)\).

5.3.3 Training

The goal of the training step is to build a classification model that predicts whether a node is valid or invalid for a given pivoted graph. The first step is to create a list of candidate nodes. These are the nodes that qualify simple tests like label and degree matching. A small percentage of the candidate nodes are chosen randomly, and used for building the predictive model. For each training node, its feature vector is composed of the pairs in the neighborhood signature, where each label-weight pair represents a feature and its weight. The pessimistic pivoted subgraph isomorphism check is used to assign a class for each training node. The pessimistic algorithm is preferred since, on average, it performs better than the optimistic one. The next step is to build a classification model using the feature vector of
each training node as well as the outcome of the pivoted subgraph isomorphism check. This model is built using the “Random Forest” classification algorithm [134], which is a simple yet effective classifier. However, other efficient classification algorithms can be used. The selection of a particular classification technique is orthogonal to this work. Since the goal is to improve the overall efficiency of pivoted subgraph isomorphism, the efficiency of the selected algorithm is a major requirement for the selection of a classification algorithm.

5.3.4 Model-Based Pivoted Subgraph Isomorphism

The following discussion describes how to combine both the pessimistic and optimistic methods to improve the overall efficiency of the PSI solution. Algorithm 8 shows the details of this step. Each graph node has a neighborhood signature which is used to predict the class of that node. Each candidate graph node is classified as either being valid or invalid (Line 6). Then, based on the classification result, either the optimistic algorithm (Line 8) or the pessimistic algorithm (Line 10) is used to get the correct decision. This decision is then fed back to a cache component attached to the model (Line 11). Upon validating next candidate nodes, PivotSubIso checks the cache, if a similar node has been processed before, a decision is looked up from the cache without consulting the classifier. The similarity between two nodes is decided based on their neighborhood signatures. Two neighborhood signatures are similar if they have similar weights of their corresponding labels. Caching improves the overall efficiency by avoiding the time spent on consulting the prediction model. Furthermore, the effectiveness is also improved since the the correct decision of a similar node is used, regardless of the correctness of the prediction.

5.4 Two PSI Methods

In this section, the details of the two proposed PSI algorithms are discussed. First, the pessimistic algorithm is discussed, which is optimized for invalid nodes. This is followed by the optimistic algorithm which is optimized for valid nodes.
Algorithm: PSIOptimized

Input: $G$ the input graph, $S$ the query pivoted subgraph
Output: $R$ the list of valid graph nodes

1. $C \leftarrow \text{GetCandidateNodes}(G, S)$
2. $(T:L) \leftarrow \text{SplitTraining}(C)$
3. $M \leftarrow \text{GenerateModel}(G, S, T)$
4. Initialize $Cache$
5. foreach $l_i \in L$ do
6. \hspace{1em} $prediction \leftarrow \text{Predict}(M, Cache, l_i)$
7. \hspace{2em} if $prediction$ is valid then
8. \hspace{3em} $r \leftarrow \text{OptimisticPSI}(G, S, l_i)$
9. \hspace{2em} else
10. \hspace{3em} $r \leftarrow \text{PessimisticPSI}(G, S, l_i)$
11. \hspace{1em} Add $(l_i, r)$ to $Cache$
12. \hspace{1em} if $r$ is valid then
13. \hspace{2em} $R \leftarrow R + r$

Algorithm 8: Optimized Pivoted Subgraph Isomorphism

5.4.1 Pessimistic Pivoted Subgraph Isomorphism

A typical subgraph matching algorithm uses a recursive function to evaluate and find mappings of query nodes to input graph nodes. The number of these recursive steps increases significantly as we go deeper in the query graph evaluation. In order to improve the efficiency of existing subgraph isomorphism approaches, pruning of candidate graph nodes is applied using node information, such as node label and degree. Since these techniques rely on local node properties, they do not lead to significant pruning. The pessimistic approach extends the pruning power by employing the neighborhood signature, which captures wider properties from neighbor nodes. The proposed pruning is based on proposition 6. This proposition identifies candidate nodes as those that have neighborhood signatures that satisfy the corresponding query node. This proposition allows the pruning of a significant number of graph nodes that do not qualify the query node neighborhood signature. Notice that it may qualify some false negative nodes as well (i.e., nodes that do not match the query graph but satisfy the corresponding neighborhood signature).

Furthermore, a modified search order optimization is used to enhance the efficiency.
This optimization is based on the neighborhood signature. Similar to the current approaches, this optimization is based on prioritizing query nodes with higher selectivity. For each query node $v$, graph nodes that satisfy its neighborhood signature are counted. Since evaluating all graph nodes is quite expensive, a small sample of the graph nodes is utilized to estimate the counts. Finally, the order of query nodes is set according to the estimated count in an ascending order. There are some cases where the selected query execution plan, which is not always optimal, might lead to an excessive number of intermediate steps. For this case, a better option would be to modify the selected plan on the fly and start evaluation from scratch. If a query spends much time and exceeds a certain threshold of intermediate steps, it is stopped. Then, the used plan is modified by selecting a query node which is causing a large number of iterations. Then, this node is advanced in the plan order. The intuition behind this is that it is better to avoid backtracking through a long list of query nodes. Rather, it would be better to have a smaller number of preceding query nodes so that recursion becomes faster.

### 5.4.2 Optimistic Pivoted Subgraph Isomorphism

The goal of optimistic subgraph isomorphism is to find an occurrence of the given query graph as quickly as possible. Given the assumption that there exists an occurrence, it worth spending some effort trying to reach that occurrence early rather than traversing other graph nodes that are to be pruned. Since subgraph isomorphism needs to find all occurrences, prioritizing graph nodes has no advantage. On the other hand, in the optimistic PSI case, ordering is important to reach a result quickly. The intuition is to traverse nodes with higher chances to quickly lead to a result. The optimistic algorithm utilizes the satisfiability score as a measure of such likelihood, i.e., nodes with higher satisfiability are traversed before other nodes. Higher scores mean that a graph node is more expected to satisfy the query node since it has more neighbors that may match query nodes.

Algorithm 9 follows the typical subgraph isomorphism algorithm except that whenever
Algorithm: PSIOptimistic

Input: $G$ the input graph, $S$ the query subgraph, $v$ the current query node, $u$ the current graph node

Output: $R$ true if a result is found, false otherwise

1. if Full mapping then
2. Return true
3. $v_{next} \leftarrow \text{GetNextQueryNode}(S)$
4. $C \leftarrow \text{GetNextCandidateNodes}(G, S, v_{next})$
5. $C_{ordered} \leftarrow \text{SortByScore}(C)$
6. foreach $c \in C_{ordered}$ do
7. \quad $valid \leftarrow \text{OptimisticPSI}(G, S, v_{next}, c)$
8. \quad if valid then
9. \quad \quad Return true
10. Return false

Algorithm 9: Optimistic Pivoted Subgraph Isomorphism

a result is found it returns true (Line 2). Moreover, Line 5 shows that candidate nodes are sorted by the scoring function. This method is relatively expensive, especially when the input graph node has many neighbors. Computing the score and ordering neighboring graph nodes may introduce unacceptable overhead. A super optimistic step is proposed to avoid such overhead. In this step, only a small portion of neighbor nodes are scored, ordered and evaluated, in the hope that the checked nodes are capable of satisfying the query graph. For the case when the super optimistic step cannot find a match, the full optimistic method is used to guarantee the correctness of the results.

5.5 Evaluation

5.5.1 Experimental setup

In this section, we experimentally validate the effectiveness of the proposed PSI techniques. First, we compare against the state-of-the-art subgraph isomorphism techniques. Then, we show the effect of the two optimized versions of PSI. Finally, we show the effect of using PSI in ScaleMine for enhancing support evaluation.

Datasets: We evaluated PSI using five real graphs that are widely used in existing work...
Table 5.1: Datasets and their characteristics

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Nodes</th>
<th>Edges</th>
<th>#node labels</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yeast</td>
<td>3,112</td>
<td>12,519</td>
<td>71</td>
</tr>
<tr>
<td>Cora</td>
<td>2,708</td>
<td>5,429</td>
<td>7</td>
</tr>
<tr>
<td>Human</td>
<td>4,674</td>
<td>86,282</td>
<td>44</td>
</tr>
<tr>
<td>Twitter</td>
<td>11,316,811</td>
<td>85,331,846</td>
<td>25</td>
</tr>
<tr>
<td>Weibo</td>
<td>1,655,678</td>
<td>369,438,063</td>
<td>55</td>
</tr>
</tbody>
</table>

Table 5.1 shows the statistics of each graph. Weibo [128] and Twitter\(^1\) are described in Section 4.4.1. Human and Yeast are graphs that model protein interaction networks, vertices represent proteins and edges represent the interactions between them. Cora\(^2\) is a citation graph consisting of 2,708 publications (nodes) and 5,429 citations between them (edges). Each node has a single label representing an area of Machine learning.

**Query Graphs:** For each dataset, we generate random query graphs of different sizes (from four to ten nodes). Each query graph is a connected subgraph extracted from the data graph. We conduct random walks with restart on the data graph to extract random query graphs for each size. For Human, Cora and Yeast, we generate 100 queries for each query size. While for Twitter and Weibo we use only ten random queries per query size.

**Hardware Setup:** We used two hardware settings for our experiments. The first one is a local cluster of 16 machines each with 148GB RAM and two 2.1GHz AMD Opteron 6172 CPUs, each with 12 cores. The machines run a 64-bit 3.2.0-38 Linux Kernel and are interconnected by a 10Gbps Ethernet switch. We also use Shaheen II, a Cray XC40 supercomputer which has 6,174 dual sockets compute nodes based on 16 cores Intel processors running at 2.3GHz with 128GB of RAM. In all experiments, we use a maximum processing time of 24 hours for any task after which the process is killed.

**Competitors:** We compare PSI against two state-of-the-art subgraph isomorphism solutions: (i) CFL-Match [85]; the best reported subgraph isomorphism solution to time. (ii) TurboIso [83]; a recent subgraph isomorphism solution that utilizes the query structure

\(^1\)http://socialcomputing.asu.edu/datasets/Twitter

\(^2\)http://linqs.umiacs.umd.edu/projects//projects/lbc/
to group similar parts of the query and process them at once in addition to an optimizer that looks for an efficient query execution plan. TurboIso is further optimized by allowing it to stop once it finds a match for each candidate node. Please note that we could not apply the same optimization to CFL-Match as its source code is not available and we could obtain only the executables from the authors.

### 5.5.2 Comparison with Existing Systems

In this section, we compare the performance of PSI against state-of-the-art subgraph isomorphism solutions namely, CFLMatch and TurboIso.

**Number of Matches:** In this experiment, we show how the number of query matches grows with the query and the input graph sizes. Table 5.2 shows the total number of matches for different query sizes for Yeast, Cora and Human datasets. In this experiment, we compare the number of matches that the subgraph isomorphism solutions report against the number of results of pivoted subgraph isomorphism. As Table 5.2 shows, the number of possible matches for subgraph isomorphism solutions grows exponentially as the query size increases even for small and sparse datasets like Yeast. As the dataset gets denser, subgraph isomorphism finds up to nine orders of magnitude more results than the pivoted version. For Human dataset, which is larger and denser than Cora and Yeast, this excessive number of possible matches makes it impossible to solve queries with query sizes more than five nodes. As we show next, the number of candidates that subgraph isomorphism solutions finds affects the performance and scalability of these solutions severely.

**Query Performance:** In this experiment, we compare the average query response time of PSI against the state-of-the-art subgraph isomorphism solutions. Subgraph isomorphism
techniques are the only available solutions for answering pivoted subgraph isomorphism queries. For fair comparison, we optimized TurboIso to stop once it finds an occurrence. Thus, it does not spend extra time on retrieving unnecessary occurrences. Figure 5.3 shows the performance of PSI for three real datasets with query sizes ranging from four to ten nodes. Yeast is a small dataset with many node labels which makes subgraph isomorphism a relatively easy task (see Figure 5.3(a)). As the query size increases, the time required for query evaluation increases. The optimizations proposed by CFLMatch works better compared to PSI and TurboIso for small query sizes (4 to 7). As the query size grows (8 to 10), all the systems achieve comparable performance. Figure 5.3(b) shows the performance of PSI for Cora. CFLMatch’s optimizations work well for small query sizes (4,5) and achieve the best performance. However, as the query size grows, it generates huge amounts of query matches. Therefore, it performs worse than the pivoted version of TurboIso and
PSI, especially for large queries. On the other hand, PSI significantly outperforms both TurboIso and CFLMatch for query sizes 7,8,9. For the query with size 10, both TurboIso and CFLMatch failed to finish the query evaluation within 24 hours and we had to abort them. Finally, Figure 5.3(c) shows the performance of PSI for the Human dataset. As this dataset is larger and denser compared to Cora and Yeast, both TurboIso and CFLMatch could not evaluate most of the query sizes. On the other hand, PSI evaluated all the queries with up to two orders of magnitude better performance.

### 5.5.3 Optimizations

In the first experiment, we show the effects of our optimizations using our largest two datasets; Weibo and Twitter. Figures 5.4(a) and 5.4(b) shows the performance of the model-based solution “PivotSubIso” compared to “Optimistic” which employs only the optimistic method and “Pessimistic” which uses the pessimistic approach for all nodes. Using the optimistic approach for all types of nodes performs severely worse than the other alternatives. The reason is that “Optimistic” assumes all candidate nodes are valid. Therefore, it tries to score and order the neighbor nodes such that it visits first the nodes that will lead to a match. While this overhead is minimal in general, applying this step for all graph nodes and their neighbors imposes significant overhead. On the other hand, the pessimistic approach performs slightly better than the optimistic for the Twitter datasets and an order of
magnitude better for the Weibo dataset. This is because the pessimistic approach avoids the overhead of scoring and ranking of all graph nodes. Also, it prunes the search space significantly. However, the pessimistic assumes the given node is invalid and therefore it visits graph nodes in a random order till it finds a match. Consequently, it misses the opportunity to find a quick solution for valid nodes. Our model-based solution tries to predict whether the candidate node is valid or invalid. Then, based on the prediction, it invokes either the optimistic or the pessimistic approach. By doing so, it avoids the unnecessary overhead of exploring large portions of the search space and scoring/ranking a huge number of graph nodes. Figures 5.4(a) and 5.4(b) show that the model-based approach is up to three orders of magnitude faster.

**Prediction Accuracy and Overhead**

In this experiment, we measure the prediction accuracy of PSI by comparing the model prediction (valid or invalid) of each pivoted node against the correct value. Figure 5.5 shows the calculated accuracy for the datasets used in our evaluation. The X-axis represents different query sizes. For the smaller datasets; i.e. Yeast, Cora, and Human, PSI achieves more than 90% accuracy. As the dataset gets larger, i.e. Twitter and Weibo, the accuracy
Figure 5.6: Model training and prediction time vs. the total time using different real datasets decreases slightly. However, for all query sizes, PSI achieves accuracy higher than 80%.

Achieving this high accuracy comes at a low computation cost compared to the total execution time. Figure 5.6 shows the time of the model training and prediction phases compared to the total pivoted subgraph isomorphism time for the different datasets. For Yeast (see Figure 5.6(a)), the training time is comparable to the total isomorphism time especially for small query sizes. This is mainly due to the sparsity and the small size of Yeast dataset. However, the difference between the training phase and the total time becomes larger as the query size increases (3X for query size 10). As the input graphs get larger, evaluation time becomes very costly. Therefore, the training time becomes very insignificant compared to the total time; see Figures 5.6(b) to 5.6(d).
Table 5.3: Comparison between ScaleMine and ScaleMine+

<table>
<thead>
<tr>
<th>Dataset</th>
<th>$\tau$</th>
<th>#Cores</th>
<th>ScaleMine</th>
<th>ScaleMine+</th>
</tr>
</thead>
<tbody>
<tr>
<td>Twitter</td>
<td>155K</td>
<td>32</td>
<td>25015</td>
<td>2677</td>
</tr>
<tr>
<td>Weibo</td>
<td>460K</td>
<td>32</td>
<td>-</td>
<td>25666</td>
</tr>
<tr>
<td>Weibo</td>
<td>460K</td>
<td>256</td>
<td>13600</td>
<td>2497</td>
</tr>
</tbody>
</table>

5.5.4 Applying PSI to FSM

The final experiment is to show the effect of incorporating our techniques on a subgraph isomorphism intensive application such as FSM. We compare ScaleMine [36] with ScaleMine+, which is a modified version of ScaleMine that employs PSI for the support evaluation step. We used Shaheen II for this experiment. Table 5.3 shows the results on two large graphs; Twitter and Weibo. For Twitter, we used 32 cores, support threshold = 155K and limited the maximum size of frequent subgraph to five edges. As for Weibo, we used support threshold = 460K and maximum size of five edges. We used two settings for Weibo; the first uses only 32 cores and the second uses 256 cores. It is obvious that our techniques significantly improves ScaleMine. For Twitter, there is an order of magnitude difference. As for Weibo on 32 cores, ScaleMine cannot finish within 24 hours, whereas ScaleMine+ finishes within 7.5 hours. When we increase the number of the used cores, both systems finish, and the difference is also significant (6X).

5.6 Conclusion

In this chapter, we showed how the improvement in the subgraph evaluation step can lead to a significant improvement in FSM. We discussed PSI, a different formulation for subgraph matching that is more suitable for FSM. Moreover, we introduced a novel technique to efficiently process PSI queries. The comparison with the state-of-the-art subgraph isomorphism techniques reveals that the proposed solution is at least an order of magnitude faster for large graphs and queries. Furthermore, we show that when PSI is used, FSM solutions can be boosted by at least six times.
Chapter 6

Incremental Frequent Subgraph Mining on Dynamic Graphs

6.1 Introduction

Emerging graph-based applications manage continuously evolving graphs. Examples include social networks, where friendships (i.e., graph edges) are established or dissolved over time; web graphs, where pages and links are constantly updated; or chemical-to-protein interaction networks, where knowledge in biomedical databases is frequently updated. A straightforward approach for continuous frequent subgraph mining in evolving graphs is to run an FSM algorithm from scratch after every graph update; we call this Naive. Typically, over a number of iterations, candidate subgraphs are evaluated and the frequent ones are extended. This involves numerous NP-hard graph isomorphism computations, rendering Naive infeasible in practice. Due to the high computational complexity, existing approaches on evolving graphs either target the simpler case of a stream of small graphs [33], or produce approximate results [34].

There are many applications that can benefit from an incremental FSM solution. Many graph based security applications utilize FSM techniques. Typically these applications are expected to be working in real-time. For example, nuclear smuggling is a world wide dangerous threat. Mining nuclear smuggling data is crucial for preventing such threat [135]. Based on FSM, a set of characteristic patterns of nuclear smuggling events are mined. Future activities that follow such patterns typically require further investigation. Since the smuggling data is updated rapidly, an incremental mining approach is important for the

---

1One iteration of the mining task on a graph with a few millions edges can take hours to finish on a typical machine (Section 3.5).
efficient utilization of such data. Moreover, graph-based anomaly detection techniques are used to prevent large-scale security threats [136]. In this setting, an anomalous subgraph is either part of or missing from a non-anomalous subgraph. Non-anomalous subgraphs are repetitive subgraphs that are discovered by utilizing subgraph mining techniques. Detecting anomalies and suspicious activities in real-time is crucial for securing systems against modern and sophisticated threats. Consequently, having a batch-based solution could fail to detect potential threats. In another domain, the configuration management database (CMDB), which can be represented as a graph [137], is important to describe the IT infrastructure entities and their interrelationships in an organization. A CMDB is considered an important information resource about the largely undocumented IT practices of a large organization, and thus mining the CDMB graph for frequent subgraphs can reveal the infrastructure patterns. These patterns are then used to set or modify IT policies. Such database usually changes rapidly and on regular basis. The maintained frequent subgraphs should reflect the up-to-date changes to the database. Failing to cope with such rapid increase, or simply waiting for a large batch of changes will negatively affect the decision outcomes. Another example is in the graph management domain, where FSM is commonly used for building indexes [15]. Creating such indexes requires a lot of time, especially for large graphs. In order to index dynamic graphs, there should be an efficient solution to incrementally update the index, instead of building it from scratch. An outdated index can drastically impact the advantages of using indices, as such, real-time index update is crucial.

We are interested in exact solutions for the continuous FSM problem on a single large evolving graph, and we focus on incremental approaches. Our problem resembles frequent itemset mining over a stream of transactions. Setting aside various approximations [68, 66], there are numerous exact incremental methods [72, 71, 73, 70, 74]. Many of them, such as the well-known MOMENT [69] system, are based on variations of the idea of a “fringe” of itemsets. Fringe is the set of itemsets on the border between frequent and infrequent ones. After the arrival of a new transaction in the stream, only the fringe needs to be
updated, reducing the cost significantly. The idea of fringe is also applicable on graphs. We implemented MomentFSM, an adaptation of MOMENT to graphs. Unfortunately, our experiments reveal that MomentFSM is too expensive in practice both in terms of memory and computational cost, because it needs to store all embeddings for each subgraph in the fringe and needs to check all subgraphs in the fringe.

In this chapter we propose IncGM+, a fast incremental technique for the continuous FSM problem on a single evolving graph. IncGM+ is orders of magnitude faster than its competitors, scales to much larger graphs and consumes limited memory. IncGM+ is based on the fringe concept, but introduces a number of novel contributions that collectively result in superior performance: (i) Instead of storing all embeddings for each subgraph in the fringe, we store only enough embeddings to prove that the subgraph is frequent or infrequent. (ii) We develop a novel index to efficiently maintain the stored embeddings. Our index is updated dynamically to reflect the current status of the system, while imposing negligible memory overhead. (iii) We propose a set of heuristics that significantly improve performance by reordering the execution of the required subgraph isomorphism operations. Our heuristics are based on information collected while processing past graph updates, and result in avoiding redundant checks. (iv) Finally, to cope with large number of updates, we support batch processing.

The rest of this chapter is organized as follows. Section 6.2 formalizes the problem. Section 6.3 describes the details of our incremental approach, whereas section 6.4 presents the experimental evaluation. Finally, section 6.5 concludes.

**6.2 Preliminaries**

There has been a recent focus on evolving graphs due to the nature of emerging applications. We first define evolving graphs:

**Definition 14.** An evolving graph $G_D = (V_D, E_D, L_D)$ consists of a set of nodes $V_D$, a set of edges $E_D \subseteq V_D \times V_D$ and a function $L_D$ that assigns node labels. Over time, $V_D$ may
change by node additions or deletions, \( E_D \) may change by edge additions or deletions, and \( L_D \) may change the label assignment of existing nodes.

An important task in graphs is to find matches of one graph in another graph, which is subgraph isomorphism. Similarly, the same operation can be applied on the dynamic graphs. Each match resulting from the isomorphism of a subgraph \( S \) to a graph \( G \) is called an embedding of \( S \) in \( G \). The goal of FSM in a static graph is to find the set of frequent subgraphs. Utilizing the MNI metric, the result set of the FSM task is defined as follows:

**Definition 15.** Given a static graph \( G \) and support threshold \( \tau \), the FSM result set \( R \) is defined as: \( R = \{Sub_1, \ldots, Sub_n\} \), where each \( Sub_i \in R \) has \( s_G(\text{Sub}_i) \) greater than or equal to \( \tau \).

Where \( s_G(\text{Sub}_i) \) is the MNI-based support value as defined in Definition 3. Similar to Figure 2.2, Figure 6.1 illustrates the search space for a typical FSM task. Figure 6.1 shows that the search space is divided into two main sets; the set \( R \) of frequent subgraphs (solid circles) and the set of infrequent subgraphs (striped and empty circles). Each of these sets is also composed of two sets. The maximal frequent subgraphs (MFS) is a subset of the set \( R \). MFS is a compressed set that efficiently represents \( R \), it is defined as follows:

**Definition 16.** MFS = \( \{S_1, \ldots, S_m\} \) is the set of all maximal frequent subgraphs such that for every \( S_i \in \text{MFS} \), \( S_i \) is frequent and there is no other subgraph \( S_j \in R \), where \( S_i \) is a subgraph of \( S_j \).

MFS is a compressed representation of the FSM result set \( (R) \); any frequent subgraph can be constructed from elements of MFS. As shown in Figure 6.1, the number of elements in MFS is much smaller than those in \( R \). Thus, focusing on MFS rather than \( R \) allows for performance improvement. Another interesting set is the set of minimal infrequent subgraphs (MIFS):
Definition 17. \( \text{MIFS} = \{S_1, \ldots, S_m\} \) is the set of all minimal infrequent subgraphs such that for every \( S_i \in \text{MIFS} \), \( S_i \) is infrequent and there is no other subgraph \( S_j \notin R \), where \( S_j \) is a subgraph of \( S_i \).

The set of infrequent subgraphs is huge; \( \text{MIFS} \) is a feasible representation for the set of all infrequent subgraphs. Other infrequent subgraphs can be constructed by extending elements from \( \text{MIFS} \). In the evolving graph setting, the goal of FSM is to continuously report the result set while the graph is updated. In this setting, FSM is defined as follows:

Problem 4. Given an evolving graph \( G_D \) and a minimum support threshold \( \tau \), the problem of frequent subgraph mining in evolving graph \( G_D \) is to continuously report the result set \( R_t = \{\text{Sub}_1, \ldots, \text{Sub}_n\} \), where each \( \text{Sub}_i \in R_t \) has \( s_{G_D}(\text{Sub}_i) \geq \tau \) after graph updates at time \( t \).

The time granularity \( t \) depends on the application under consideration. For sensitive applications that require on-line processing, \( t \) should be very small. Whereas for other less sensitive application, \( t \) can be set to higher values. Dynamic graph updates can be considered as a stream of edge and node updates. Updates are either node/edge additions, deletions or label modifications. In the following, we focus only on edge additions and...
deletions, since all other types of updates can be supported by graph updates. Figure 6.2 illustrates an example of an evolving graph at three points in time: $t_1$, $t_2$ and $t_3$. Suppose $\tau = 2$ and $G$ is the evolving input graph. At time $t_1$ the number of matches of $S_1$ is two, whereas $S_2$ has only one match; hence, $S_1$ is frequent and $S_2$ is infrequent. Advancing to time $t_2$, edge $u_6 - u_7$ is deleted and the number of embeddings of $S_1$ becomes one, while the number of embeddings for $S_2$ does not change. As a result, both $S_1$ and $S_2$ are infrequent. At time $t_3$, edge $u_2 - u_8$ is added, increasing the number of matches of $S_2$ to two; thus, $S_2$ becomes frequent.

### 6.3 Incremental Graph Mining

In this section, we propose $IncGM+$, an incremental FSM solution for evolving graphs. $IncGM+$ employs three novel techniques to improve the efficiency. First, it prunes the search space by focusing on a set of carefully selected subgraphs (fringe subgraphs). After each graph update, only these subgraphs are evaluated. Other elements of the search space
are evaluated only if needed. Second, IncGM+ maintains a minimal number of embeddings for each of the selected subgraphs. These embeddings are used to enhance or to avoid evaluating the fringe subgraphs. Finally, IncGM+ utilizes information collected during past iterations to improve the efficiency of next iterations. We use IncGM to refer to the incremental solution that only uses search space pruning.

6.3.1 Search Space Pruning

IncGM utilizes the well known “fringe” concept for incremental search space evaluation. It employs the fringe subgraphs; subgraphs that either belong to MIFS or MFS. An example of the fringe subgraphs is shown in Figure 6.1. By using this fringe, search space is significantly pruned and focus is given to those parts of the search space that are more sensitive to changes. The fringe subgraphs are evaluated after each graph update. Evaluating a subgraph (i.e., finding whether it is frequent or not) requires significant overhead to find its embeddings. To accelerate IncGM, it is important to avoid unnecessary evaluations. For example, it is useless to evaluate fringe subgraphs that are not expected to be affected by recent updates. In the following, we show two propositions that can be exploited to avoid such unnecessary overhead.

**Proposition 7.** Adding an edge to the input graph results in increasing the support of one or more subgraphs. Thus, after an edge addition at time \( t \), the only difference (if exists) between the result set \( R_{t-1} \) and \( R_t \) is the addition of one or more subgraphs to \( R_t \).

**Proposition 8.** Removing an edge from the input graph results in decreasing the support of one or more subgraphs. Thus, after an edge deletion at time \( t \), the only change that might happen to the result set \( R_t \) is to remove one or more subgraphs from \( R_{t-1} \).

In other words, only elements of MIFS need to be evaluated after edge additions, and only elements of MFS need evaluation after edge deletions. Given a graph update, Algorithm 10 shows how IncGM utilizes the fringe subgraphs to prune the search space. The
Algorithm: FringeBasedMining

Input: $G$ the input graph, $\tau$ support threshold, $U$ graph update

1. if $U$.class was not seen before then Add $U$ to MIFS
2. FRINGE ← MINE($G_D$, $\tau$)
3. if $U$ is edge addition then
   4. foreach $S \in$ MIFS do
      5. if SUBGRAPH($U$, $S$) then EVALUATE($G$, $\tau$, $S$)
      6. if $S$ changes status then UPDATEFRINGE(FRINGE, $S$)
   7. else
      8. foreach $S \in$ MFS do
         9. if SUBGRAPH($U$, $S$) then EVALUATE($G$, $\tau$, $S$)
         10. if $S$ changes status then UPDATEFRINGE(FRINGE, $S$)

Algorithm 10: Fringe-Based Incremental Frequent Subgraph Mining

fringe is either populated from an initial FSM step, or updated by the previous iteration of the algorithm. IncGM evaluates a fringe subgraph $S$ only if it is infrequent and the update is edge addition (line 4), or $S$ is frequent and the update is edge deletion (line 8). Computing the new support is done by calling EVALUATE. This function decides whether $S$ is frequent or not by searching for enough embeddings of $S$ in $G$ to satisfy $\tau$. EVALUATE needs to search for embeddings from scratch since IncGM does not maintain previously found embeddings. Thus, EVALUATE incurs significant overhead. When a subgraph changes its status, the fringe is updated by calling UPDATEFRINGE (Lines 6 and 10). For a subgraph $S_{\text{FREQ}}$ that is recently found to be frequent, UPDATEFRINGE updates the fringe by: (1) Adding $S_{\text{FREQ}}$ to MFS, (2) Removing $S_{\text{FREQ}}$ from MIFS, and (3) extending $S_{\text{FREQ}}$ by joining it with other frequent subgraphs of the same size [6]. The extended subgraphs are added to MIFS and recursively evaluated. For a new infrequent subgraphs $S_{\text{INFREQ}}$, UPDATEFRINGE updates the fringe by: (1) Adding $S_{\text{INFREQ}}$ to MIFS, (2) Removing $S_{\text{INFREQ}}$ from MFS, and (3) Adding decompositions of $S_{\text{INFREQ}}$ to MFS. Each decomposition is created by removing one edge from $S_{\text{INFREQ}}$. Finally, the added decompositions are recursively evaluated. Further pruning is done in lines 5 and 9; $S$ is evaluated only if $U$ is contained in $S$. This pruning is possible since there is no way to affect the support of a subgraph $S$ by an edge update that is not contained in $S$. Line 1 contains an important step for the correctness of
the algorithm: Any new edge update, which was not seen before, is added to \textit{MIFS}. Without this step, \textit{IncGM} will not consider this edge or any of its supergraphs for evaluation, even when this edge becomes frequent.

\textbf{Correctness:} Assume there exists a frequent subgraph $S$ that is not recognized as frequent by Algorithm 10. According to the anti-monotone property, each edge of $S$ must be frequent. Algorithm 10 recognizes all frequent edges by: past iterations (when it is already frequent), current iteration (when it was previously infrequent) or line 1 (when it was not seen before). Moreover, \textsc{updateFringe} guarantees that for a frequent edge, all of its frequent extensions are discovered. Hence, Algorithm 10 will identify $S$ as frequent, which contradicts with the first assumption.

By utilizing the discussed propositions and optimizations, the search space is significantly pruned. However, algorithm 10 suffers from considerable overhead caused by \textsc{evaluate}. Performance can be improved in two ways. First, by optimizing \textsc{evaluate}. Second, by limiting the number of times \textsc{evaluate} is called. These improvements are discussed next.

\subsection{6.3.2 Embeddings-based optimization}

\textsc{evaluate} incurs the most computation overhead, most of this overhead is devoted to finding embeddings from scratch. Reducing this overhead can be achieved by maintaining a list of previously found embeddings, and thus, finding embeddings from scratch is avoided. Storing all embeddings is prohibitively expensive, since the number of embeddings grows exponentially with the graph size. A better solution is to store a minimal number of embeddings, which is small enough to fit in the available memory. The proposed approach, which is called \textit{IncGM+}, limits memory consumption by adopting the following guidelines:

- For each subgraph $S \in \textit{MFS}$, \textit{IncGM+} stores minimal number of embeddings that makes $s_G(S) \geq \tau$, other embeddings are not maintained. Each embedding corresponds to at least one cell in the $\textit{MNI}$ tab, and in many cases a single embedding corresponds to more
Figure 6.3: (a) Input graph $G$ (b) A Subgraph $S$ (c) the MNI table of $S$ embeddings in $G$ when $\tau = 3$

than one cell. Consequently, for a subgraph $S$, the upper bound for the number of embeddings needed by $IncGM+$ to satisfy $\tau$ is $\tau \cdot |S|$, where $|S|$ is the number of nodes in subgraph $S$. In many cases, the number of maintained embeddings can be much lower. For example, in Figure 6.3, assume $\tau = 3$. For $S$ to be frequent, only three embeddings are required to satisfy $\tau$, regardless of the actual number of existing embeddings.

- For each subgraph $S \in MIFS$, $IncGM+$ stores all of its embeddings. By definition, each subgraph in $MIFS$ has support less than $\tau$. In other words, each infrequent subgraph has at least one $MNI_{col}$ that has a number of valid nodes less than $\tau$. Thus, the number of embeddings for an infrequent subgraph $S$ is bounded by $T \cdot |S|$, where $T < \tau$.

By utilizing the above guidelines, the number of stored embeddings for both $MFS$ and $MIFS$ is bounded by $\tau$. Though, storing them as a simple list is inefficient. For efficient maintenance of the stored embeddings, we propose the Fast Embeddings Lookup Store
Algorithm: Embeddings-Based FSM

Input: $G$ the input graph, $\tau$ support threshold, $U$ Dynamic updates

1. if $U$ was not seen before then Add $U$ to $MIFS$
2. FRINGE $\leftarrow$ MINE$(G_D, \tau)$
3. if $U$ is edge addition then
   4. foreach $S \in MIFS$ do
      5. EMBEDS $\leftarrow$ SEARCHLIMITED$(S, U)$
      6. if EMBEDS is null then UPDATE_SUPPORT$(G, \tau, S)$
         else
            7. FELS_UPDATE(EMBEDS)
            8. $S.freq \leftarrow$ MNI$(S)$
            9. if $S$ changes status then UPDATEFRINGE(FRINGE, $S$)
   10. else
      11. $S_{All} \leftarrow$ MIFS $\cup$ MFS
      12. REMOVE EMBEDS($S_{All}, U$)
      13. foreach $S \in MFS$ do
         14. if MNI$(S) < \tau$ then
            15. UPDATE_SUPPORT$(G, \tau, S)$
            16. if $S$ changes status then UPDATEFRINGE(FRINGE, $S$)

Algorithm 11: Optimized Incremental Frequent Subgraph Mining Based on Storing Minimal Number of Embeddings

(FELS). FELS allows efficient addition and removal of embeddings. Moreover, MNI-based support computation is significantly accelerated by FELS. More details about FELS is presented in Section 6.3.3.

Algorithm 11 shows how IncGM+ exploits the materialized embeddings. When an edge is added (Line 3), IncGM+ only needs to search for new embeddings instead of searching for old embeddings from scratch. At line 5, SEARCHLIMITED finds the new embeddings by applying subgraph isomorphism containing the added edge. Those newly found embeddings are added to the FELS object associated with $S$ (Line 8). Then, the current subgraph is checked for being frequent (Line 10). This check is efficiently conducted by utilizing the FELS object. Finally, the fringe is updated accordingly (Line 10). Note that, without maintaining the list of embeddings, it is required to call EVALUATE, which needs to search for embeddings from scratch. SEARCHLIMITED employs the following optimization: In some cases, the local area around the added edge is dense and contains a large number of
embeddings. Only in such scenario, searching the local area for all embeddings poses extra overhead compared with \texttt{EVALUATE}, which follows the approach proposed by Grami and is designed to efficiently fill the \textit{MNI} table. For that reason, a limit on the number of embeddings is set. While searching the local area for new embeddings, if the number of found embeddings exceeds that limit, the algorithm halts and falls back to the normal evaluation method (\texttt{EVALUATE}). This is the only case that \texttt{EVALUATE} is needed for edge additions.

By maintaining the list of embeddings, Calling \texttt{EVALUATE} after edge deletion is almost avoided. When an edge is deleted, some of the maintained embeddings will vanish and need to be removed from the list of embeddings associated with a subgraph $S$ (Line 13). But in many cases, the deleted edge does not affect any of the stored embeddings, especially when the input graph is large, and the stored embeddings represent a small portion of the graph. In such cases, the evaluation of $S$ is not affected. If an edge deletion results in the removal of stored embeddings, then \textit{MNI} is computed using the remaining ones. If, based on the currently maintained embeddings, the computed \textit{MNI} value satisfies $\tau$, then there is no need to do further processing (Line 15). Otherwise, \texttt{EVALUATE} is used to find more embeddings (Line 16).

Algorithm 11 treats edge additions and edge deletions differently. For edge additions, only elements in \textit{MIFS} are processed. While, for edge deletions, elements in both \textit{MFS} and \textit{MIFS} are processed. The following discussion highlights the reasons for this difference. Edge additions are more expensive since new embeddings are to be found. While for edge deletions, obsolete embeddings are removed from the embeddings lists, these removals are efficiently done by our novel data structure (\textit{FELS}). Due to its efficiency, edge removal is not postponed and is immediately applied to the two sets: \textit{MIFS} and \textit{MFS} (Line 13). While for edge addition, in order to minimize the processing overhead and memory consumption, embeddings are only added to subgraphs belonging to \textit{MIFS}. As a result, not all existing embeddings of subgraphs in \textit{MFS} are maintained. Thus, it is possible for a subgraph to be frequent even if its maintained embeddings cannot satisfy $\tau$ (line 15). For such case,
calling `EVALUATE` is required in order to find other embeddings that were not discovered before (Line 16).

**Correctness:** Decisions regarding infrequent fringe subgraphs are based on the complete list of existing embeddings. Thus, these decisions are guaranteed to be correct. As for frequent fringe subgraphs, they only maintain a minimal number of embeddings to satisfy $\tau$. When a graph update is edge addition, then there is no effect on the frequent edges. When the update is edge deletion and it does not affect the set of embeddings, the decision will not be affected as it is based on maintained embeddings. If the deletion affects the maintained embeddings, then full re-evaluation is used to guarantee the correctness.

### 6.3.3 Fast Embeddings Lookup Store (FELS)

`FELS` is an efficient store for maintaining a list of embeddings. It allows fast access and update. Moreover, it is used to compute the support value using the stored embeddings.
Components: Each FELS object corresponds to a subgraph S and has three components:
1- A Hash table of embeddings of S, 2- Inverted index from nodes to embeddings, and 3- An MNI table. Each embedding is hashed by its unique key, this key is created by concatenating the embedding node IDs ordered according to their corresponding S nodes IDs. The inverted index is used to lookup embeddings given their nodes. FELS utilizes the MNI table to compute the MNI-based support value. Each cell in the MNI table corresponds to a node, the number of embeddings containing this node is attached to each cell.

Figure 6.4 illustrates the (FELS) object of subgraph S from Figure 6.3. six embeddings are maintained in this object, namely: \{e_1, e_2, e_3, e_4, e_5, e_6\}, each one having a unique key. For example, embedding e_2 has key: “u_{21}.u_{19}.u_{20}”. The inverted index in Figure 6.4.a contains 16 distinct graph nodes, each node indexes the embeddings it is contained in. For example, e_2 is indexed by u_{21}, u_{19} and u_{20}. Some nodes may index more than one embedding such as u_{12} which indexes two embeddings e_5 and e_6. Figure 6.4.b shows the MNI table, each column corresponds to a specific node \in S and is populated with distinct matching nodes \in G. There is a counter value attached to each cell representing the number of embeddings indexed by the node corresponding to this cell. For example, node u_{12} has a value two as it indexes two embeddings: e_5 and e_6.

Operations on Embeddings: FELS supports efficient addition and removal of embeddings. For adding an embedding E, it is inserted in the hash table using its key. The hash table ensures that only one instance of E exists in FELS. Then, the inverted index is updated by adding a pointer to E from each u \in E. Finally, each node u \in E is inserted in the MNI table. If a node does not exist in its corresponding MNI_{col}, then an entry for this node is created, and its counter is set to 1. Otherwise, the counter associated with its entry is incremented. As for removing an embedding E, the key of E is used to efficiently remove it from the hash table. Then, all pointers to E are removed from the inverted index. Those nodes that do not point to any other embeddings are then removed from the inverted index. Finally, the MNI table is updated by decrementing the counter associated with each
A node is removed from the $MNI$ table if its counter becomes 0.

**MNI Computation:** *FELS* utilizes the existing embeddings to compute the $MNI$-based support values. This is done by checking the length of each $MNI_{col}$ and reporting the minimum length as the $MNI$ value. For example, in Figure 6.4, given $\tau = 5$, $S$ is frequent because its support value based on the $MNI$ table is five. Suppose that edge $u_{11} - u_{12}$ is deleted from the input graph. Then embedding $e_5$ becomes obsolete and is removed from the inverted index and the $MNI$ table. The new set of embeddings becomes: \{ $e_1, e_2, e_3, e_4, e_6$ \}. By consulting the $MNI$ table, all of its columns become of length 5. Thus, the support value is still 5. This happens because $u_{12}$ and $u_{10}$ entries in the $MNI$ table both had an attached value of two (two embeddings indexed by each one). Since the embedding $(u_{11}, u_{12}, u_{10})$ is removed, the counter attached to each node is decremented. Thus, $u_{11}$ is removed, while $u_{12}$ and $u_{10}$ both remain in the $MNI$ table.

### 6.3.4 Reordering

The execution order can affect the performance significantly. The problem is how to decide which order is better to use. *IncGM* exploits information collected during past iterations to follow better ordering, it employs the following two ordering heuristics:

1- **Nodes reordering:** Given an input graph $G$ and a subgraph $S$, an invalid node is a node that belongs to $G$ and cannot be part of an embedding of $S$ in $G$. Checking the validity of these nodes is usually the reason behind most of the processing overhead. To enhance the performance, the list of invalid nodes is maintained during previous iterations. Then, while evaluating the support of $S$ in subsequent iterations, invalid nodes are postponed for the hope that other nodes can satisfy $\tau$. As such, a significant amount of computation associated with invalid nodes is avoided.

2- **$MNI_{col}$ reordering:** A subgraph is infrequent if it has at least one invalid column. It is usually normal for infrequent subgraphs to stay infrequent and to have the same invalid column in future iterations. After evaluating the support of infrequent subgraphs, *IncGM*
maintains the invalid column for each one. Then, for future evaluations, \( \text{IncGM} \) starts by checking the invalid columns. As such, the redundant overhead of checking \( MNI_{col} \) other than the invalid ones is avoided.

### 6.3.5 Batching

For practical applications with heavy workloads, batching can be used to speedup the processing. Grouping updates and processing them at once allows expensive support computations to be aggregated for improved efficiency. The proposed batching approach consists of two parts; updates grouping and subgraphs pruning.

**Updates grouping.** Updates grouping utilizes the following three steps: First, repeated updates are removed. For instance, when adding an edge \( u_i \rightarrow u_j \) more than once, only one addition is considered. Second, edges that cancel each other are ignored. For example, when an edge \( u_i \rightarrow u_j \) is added then deleted, there is no need to process any of these two updates. The third optimization is grouping optimization which is non-trivial compared to the first two steps. The goal of grouping optimization is to assure that any subgraph is processed at most once, even if more than one edge update affects that subgraph. The first step is to group edge updates of the same class together. Then for each edge class, the set of affected subgraphs are identified. The final step is to union all the sets of affected subgraphs into the \( \text{ToBeChecked} \) list, which contains the set of subgraphs that need to be processed. Significant processing saving can be achieved by using this optimization as will be seen in the evaluation section.

**Subgraphs Pruning.** The relationship among subgraphs belonging to \( \text{ToBeChecked} \) can be of great value. Each subgraph \( \in \text{ToBeChecked} \) can be either a child or a parent of one or more subgraphs \( \in \text{ToBeChecked} \). A subgraph \( S_1 \) is a child of another subgraph \( S_2 \), if \( S_1 \) is infrequent and it is a supergraph of \( S_2 \). Also, a subgraph \( S_2 \) is a parent of \( S_1 \) if \( S_2 \) is a frequent subgraph and it is a subgraph of \( S_1 \). The following propositions highlight interesting properties of these relationships.
Proposition 9. Given $S_1, S_2 \in \text{ToBeChecked}$. If a subgraph $S_1$ is a child of a subgraph $S_2$, and $S_2$ is checked and found to be infrequent. Then, $S_1$ can be safely removed from $\text{ToBeChecked}$.

Proposition 10. Given $S_1, S_2 \in \text{ToBeChecked}$. If a subgraph $S_1$ is a parent of a subgraph $S_2$, and $S_2$ is checked and proved to be frequent. Then, $S_1$ can be safely removed from $\text{ToBeChecked}$.

In other words, no need to evaluate a frequent subgraph if one of its children is found to be frequent. Also, no need to evaluate an infrequent subgraph if one of its parents is found to be infrequent. Many candidate subgraphs can be pruned by exploiting propositions 9 and 10. The question is which subgraphs to start evaluating in order to maximize the benefits of this pruning. It is better to start with subgraphs that are about to change their status rather than processing subgraphs that will not be affected. For example, for an infrequent subgraph $S_1$ which is a child of a frequent subgraph $S_2$, if it is known that $S_1$ will become frequent after applying the current batch of updates, then it is better to start with $S_1$, and hence, processing of $S_2$ is avoided. But, such information is not known in advance. Thus, we propose a heuristic-based solution. To predict a good ordering, we design a simple algorithm and a scoring function. The scoring function gives higher scores to subgraphs that are expected to change their status. The algorithm works as follow: First, all edge deletions are processed on all subgraphs $\in \text{ToBeChecked}$. Second, the $\text{ToBeChecked}$ list is shortened by removing the subgraphs that are still frequent after the first step. Finally, $\text{ToBeChecked}$ is sorted in an ascending order according to the following scoring function:

$$\text{Score}(S) = \begin{cases} 
(\alpha_S + \#\text{Edges} \times \beta_S) - \tau & \text{if } S \in \text{MFS} \\
\tau - (\alpha_S + \#\text{Edges} \times \beta_S) & \text{if } S \in \text{MIFS}
\end{cases}$$

Where $\alpha_S$ is the recent support of $S$, $\#\text{Edges}$ is the number of edge additions in the current batch. $\beta_S$ is the expected increase in the support of $S$ per edge addition. $\beta_S$ is
estimated using the previous iterations; $\beta_S = \text{median}(L_S)$, where $L_S$ is the list of support increments per each previous edge addition. The final step of the algorithm is to evaluate the candidate subgraphs according to the ordered list. While evaluating the candidates, parents of frequent subgraphs are removed from ToBeChecked as well as children of infrequent subgraphs. New subgraphs that arise as a result of extending existing subgraphs are appended to the end of ToBeChecked.

### 6.4 Experimental Evaluation

In this section, we experimentally compare the proposed incremental approaches; IncGM and IncGM+ against competitors relying on existing techniques. IncGM represents our incremental approach based on fringe pruning (Subsection 6.3.1 4.1). IncGM+ is the extension of IncGM that maintains a minimal number of embeddings and utilizes the ordering optimizations (Subsections 6.3.2, 6.3.3, 6.3.4).

**Competitors:** Since there is no prior work on the problem of incremental FSM, we implemented two baseline competitors: Naive and MomentFSM. Naive executes an FSM algorithm from scratch after each graph update. For our experiments, Naive employs GRAMI which is the state-of-the-art FSM technique. MomentFSM borrows some ideas from Moment [69]; a well known solution for incremental frequent itemset mining. MomentFSM maintains a fringe of subgraphs that lay on the boundary between frequent and infrequent subgraphs. It stores all embeddings for each fringe subgraph. After each graph update, fringe subgraphs are re-evaluated and their embeddings are updated accordingly.

**System specs:** All experiments are conducted using a machine with 2.67GHz Intel Xeon processor and 192GB of RAM. The machine runs Linux Ubuntu 12. Our systems and the mentioned competitors are implemented in Java. Notice that, we used a machine with large memory to be able to run MomentFSM while both IncGM and IncGM+ can run on a machine with much lower memory.
Datasets: We use four real directed graphs in our experiments. Namely, Citeseer, Yahoo, Patents and Twitter. Table 6.1 summarizes the characteristics of these graphs, and Section 3.5 discusses the details of each dataset except Yahoo. Yahoo is a graph which is part of the Yahoo! Webscope project (G5 v1.0) that represents a network of Yahoo! Messenger user communications over 28 days. Each node represents a user and is labeled with his address zip code. Each edge represents a communication between two users. A timestamp is attached to each edge stating the first communication time and date.

Workloads: For each dataset, three workloads are generated: edge additions, edge deletions and an equal mix of additions and deletions.

Twitter: Twitter does not have timestamps; therefore, edge additions and deletions are randomly chosen. We use 2500 edge updates for each workload.

Patents: Edges are ordered according to their grant date. For edge additions, the experiment proceeds with the original graph without the latest 50K edges. Then, these edges are added ordered by their timestamps. For edge deletions, the whole graph is used. Then, 50K edges are deleted following the chronological order. For the mixed workload, the original graph is used without the latest 25K edges. Then the newest 25K edges are added while the oldest 25K edges are deleted.

Yahoo: As this dataset has real timestamps, Workloads follows the chronological order as shown for the Patents dataset. The size of each workload is 50K updates.

CiteSeer: The workload follows a similar approach of the Twitter dataset except for the deletions workload. Since Citeseer is a small graph, the deletions workload starts with the

---

Table 6.1: Datasets and their characteristics

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Density</th>
<th>Nodes</th>
<th>Edges</th>
<th>Distinct node labels</th>
</tr>
</thead>
<tbody>
<tr>
<td>Twitter</td>
<td>Dense</td>
<td>11,316,811</td>
<td>85,331,846</td>
<td>25</td>
</tr>
<tr>
<td>Patents</td>
<td>Dense</td>
<td>2,745,761</td>
<td>13,965,409</td>
<td>418</td>
</tr>
<tr>
<td>Yahoo</td>
<td>Dense</td>
<td>99304</td>
<td>903113</td>
<td>5576</td>
</tr>
<tr>
<td>CiteSeer</td>
<td>Medium</td>
<td>3,312</td>
<td>4,732</td>
<td>6</td>
</tr>
</tbody>
</table>

---

\(^2\text{http://webscope.sandbox.yahoo.com}\)
original graph extended with 2500 random edges, then they are removed in a random order. Each workload contains 2500 updates.

In the rest of this section, we compare the different approaches in terms of efficiency and memory consumption. Finally, the proposed batching technique is evaluated.

### 6.4.1 Efficiency

In this experiment, the efficiency of the different systems is measured using different workloads. For each workload, the elapsed time of graph loading and processing the whole workload is measured. Figure 6.5 shows the efficiency results. The y-axis indicates the
elapsed time (in log scale), while the x-axis represents the support threshold. For Twitter, Patents and Yahoo, it is not feasible to process the whole workload using Naive. Thus, we estimate the elapsed time by measuring the time for processing a small subset of the workload then extrapolating the total time to process the whole workload. For edge additions, IncGM+ shows at least two orders of magnitude improvement compared to Naive. This improvement is the result of using the fringe subgraphs to prune the search space. As for the deletions workload, the improvement is more than three orders of magnitude. Compared to the additions workload, the deletions workload affects a smaller subset of the fringe. Consequently, deletion updates are processed faster. MomentFSM shows good performance for smaller graphs and higher $\tau$ values (i.e, Figure 6.5.B.1 for $\tau = 160$). For larger graphs and lower $\tau$, it cannot even complete the task. For example, MomentFSM consumes available memory and cannot finish any task for Twitter. This happens because MomentFSM needs to store all embeddings for the fringe subgraphs. For larger graphs and lower $\tau$ values, the number of embeddings becomes enormous, storing these embeddings requires storage exceeding the available memory.

Figure 6.5 shows the benefits of the embeddings-based optimization implemented in IncGM+. The improvement is particularly notable for both Citeseer and Patents. For higher $\tau$ values, the average size of frequent subgraphs found in the Twitter and Yahoo datasets is rather small. Maintaining these small embeddings almost equals the overhead of searching for embeddings. Hence, the embeddings-based optimization gives minimal improvement. For lower $\tau$ values, the size of frequent subgraphs gets larger, and the cost of searching for embeddings becomes significant. Thus, utilizing already found embeddings starts paying off. For example, at Figure 6.5.C.3, there is a considerable improvement when $\tau = 110k$.

### 6.4.2 Memory overhead

The following experiments measure the memory overhead. Figure 6.6 shows the number of subgraphs stored in each set: MFS and MIS for the different datasets and different $\tau$
values. It is clear that the size of each set is not extremely large, and they do not rapidly get bigger as $\tau$ decreases. Moreover, the size of $MFS$ is always smaller than the size of $MIFS$.

Another interesting observation is that the size of $MIFS$ is affected by the number of distinct labels in the input graph. $MIFS$ gets larger as the number of distinct labels increases. For example, Citeseer has the smallest number of distinct labels, and it has the smallest $MIFS$. Also, Yahoo has the largest number of distinct label and, accordingly, the largest $MIFS$.

Figure 6.7 shows the memory consumption of each system. $Naive$ does not store any intermediate results, so it consumes the least amount of memory. On the other hand, $MomentFSM$ maintains all embeddings corresponding to each fringe subgraph. Consequently, it consumes a lot of memory (i.e., Figure 6.7.2) and even crashes due to exceeding the memory limit (i.e., Figure 6.7.3). For this, $MomentFSM$ cannot be considered a feasible solution. Figure 6.7 also highlights the memory consumption of our proposed techniques. In comparison with $Naive$, $IncGM$ does not excessively consume much memory to maintain the fringe. The maximum increase in memory consumption appears for the smallest graph (around 4X the memory of $Naive$). While for the larger graphs, the increase is relatively smaller (Figures. 6.7.2-6.7.4). Although $IncGM+$ maintains a list of embeddings, its extra memory usage is insignificant compared to $IncGM$. Our approach, which carefully selects a minimal number of embeddings, consumes minimal extra memory space.

There is a correlation between the extra memory overhead and the efficiency results of Figure 6.5. As more memory is consumed, the performance difference between $IncGM$ and $IncGM+$ becomes more significant. In Figure 6.5.A.2, for $\tau \geq 19k$, there is no improvement
Figure 6.7: Memory overhead.

Figure 6.8: Twitter batching. Time represents the time required to process the given workload using different batch sizes compared to edge by edge updates.

In IncGM+ results. This is because no embeddings are utilized when \( \tau \) is greater than 18k which is confirmed by the memory consumption (Figure 6.7.2). When \( \tau = 18k \), differences between IncGM and IncGM+ in both memory consumption and efficiency is clear.

6.4.3 Batching

In the following experiments, we try to answer the following three questions: What is the effect of batching on performance? How incremental batching is compared to Naive batching? and what are the benefits of our batching optimizations?.

Figure 6.8 tries to answer the first question. For this experiment, we use Twitter, the largest graph in our datasets. Two workloads are used in the experiments, Figure 6.8.a
Figure 6.9: Effect of different batch size. Time represents the time required for processing all update batches.

shows the results for edge additions and Figure 6.8.b is for deletions. For each workload, three settings are used, the first is to process edge by edge (Inc1), the second is to batch 100 updates and process them together (Inc100), the third is to batch 500 updates (Inc500). Each bar represents the workload processing time using the corresponding batch size. Batching improves the overall performance as the batch size increases. For example, Inc500 outperforms Inc1 by more than an order of magnitude for the additions workload. As for deletions, there is a significant improvement, but not as much as in additions.

Figure 6.9 answers the second question. It shows the speedup of incremental batching compared to Naive batching. The improvement differs as batch size is changed while $\tau$ is fixed. Both experiments are using the additions workload. Figure 6.9.a shows the results for Twitter when $\tau = 150k$. Speedup keeps decreasing as batch size increases until they almost converge. Figure 6.9.b shows the results for Patents when $\tau = 18k$. Starting with a batch size of 100, the incremental approach is two orders of magnitude faster. After a certain threshold (i.e., batch size=10k), Naive becomes closer to our incremental approach. The main reasoning behind this observation is that as the batch size gets larger, more parts of the search space are touched and, therefore, less pruning can be applied. Similar observations are also reported in other incremental approaches [138, 139].

The last experiment answers the third question regarding the effect of the proposed batching optimization techniques; the grouping and the subgraph pruning optimizations.
Table 6.2: Comparing the performance of the different batching optimization techniques.

<table>
<thead>
<tr>
<th>Dataset/τ</th>
<th>Citeseer/120</th>
<th>Patents/18k</th>
</tr>
</thead>
<tbody>
<tr>
<td>NoBatching</td>
<td>121</td>
<td>3125</td>
</tr>
<tr>
<td>Opt1</td>
<td>86</td>
<td>488</td>
</tr>
<tr>
<td>Opt2</td>
<td>18</td>
<td>396</td>
</tr>
</tbody>
</table>

Table 6.2 shows the results where (NoBatching) refers to the edge-by-edge processing, (Opt1) refers to the grouping optimization and (Opt2) refers to both the grouping and subgraph pruning optimizations. As noticed from the results, significant improvement is achieved by applying Opt1 and Opt2. This improvement results from the significant pruning of candidate subgraphs that need to be evaluated.

6.5 Conclusions

In this chapter, we study the problem of frequent subgraph mining on evolving graphs. We highlight the importance of the problem and show that current solutions are inapplicable. A novel solution is proposed, based on utilizing information collected during previous iterations. Such information is exploited to enhance the performance of next iterations. Furthermore, a novel index is proposed to improve the efficiency of frequency evaluation. Finally, we discuss how batching can be utilized to improve the performance. Through extensive experiments on large real datasets, we show that our solution outperforms state-of-the-art static FSM algorithms by up to three orders of magnitude.
Chapter 7

Concluding Remarks

Frequent Subgraph Mining is a popular graph operation because of its importance in graph analytics and mining. Because of its complexity, prior state-of-the-art techniques cannot discover frequent subgraphs in large input graphs using low frequency thresholds. Consequently, the usefulness of FSM is limited. The next section concludes with a summary of our contributions that improve FSM and allow it to process real large graphs. Then, the subsequent section gives an outlook on possible future research directions.

7.1 Summary of Contributions

In this thesis, four components are introduced to improve the efficiency of frequent subgraph mining. The first component is GRAMI, a single-threaded FSM solution. GRAMI minimizes the overhead of evaluating candidate subgraphs by transforming it to a constraint satisfaction problem. Based on this transformation, GRAMI evaluates each candidate subgraph by conducting a minimal number of subgraph isomorphism operations. Furthermore, GRAMI utilizes a set of optimizations that significantly improves the overall efficiency of the FSM task. The second component of this thesis discusses ScaleMine, a parallel FSM system that utilizes thousands of processors. ScaleMine employs a novel two-phase technique that balances the workload among the compute nodes. The first phase builds an approximate search space, which is then utilized by the second phase to minimize the overall overhead as well as balance the workload among the compute nodes. The third component focuses on accelerating the task of subgraph evaluation, which is the
bottleneck of FSM. In this component, pivoted subgraph isomorphism is used instead of the traditional subgraph isomorphism definition. Consequently, finding relevant subgraph occurrences becomes more efficient by adopting the new definition. Furthermore, a novel algorithm is introduced to optimize the efficiency of this task. The last component, which we call \textit{IncGM+}, supports FSM on dynamic graphs. It employs a novel algorithm for mining dynamic graphs. Moreover, \textit{IncGM+} relies on maintaining a minimal set of subgraphs to decide which subgraphs becomes frequent after the most recent graph updates. \textit{IncGM+} is also boosted by supporting a batch of updates.

Our extensive evaluation shows that the proposed components significantly outperform existing techniques. Compared to single-threaded solutions, \textsc{Grami} scales to an order of magnitude larger graphs and achieves up to two orders of magnitude better performance. \textsc{ScaleMine} outperforms existing parallel solutions by at least an order of magnitude. Additionally, it scales to an order of magnitude larger real graphs and is able to utilize 12X more compute cores. Our subgraph evaluation solution is shown to significantly outperform existing state-of-the-art subgraph isomorphism techniques. Furthermore, it boosts \textsc{ScaleMine} by up to an order of magnitude when it replaces the default subgraph evaluation operation. Finally, \textit{IncGM+} is up to three orders of magnitude faster and consumes at least an order of magnitude less memory compared to other competitors that rely on available techniques.

Based on this work, we expect future research, in multidisciplinary domains, to benefit from the proposed techniques to support FSM applications on modern large graphs.

### 7.2 Future Research Directions

The proposed solutions assume that each machine has an in-memory copy of the input graph. However, this assumption limits these solutions to graphs that fit within the memory capacity of a single machine. An interesting extension is to investigate efficient techniques to partition the input graph and assign each partition to a different machine. Typically, graph partitioning increases the communication overhead due to the exchange of data
and some intermediate results among machines. Employing adaptive partitioning techniques [140] is a possible solution to reduce this overhead. By overcoming this limitation, we can scale to at least an order of magnitude larger graphs (i.e., 10B edges) and more workers (i.e., 100,000 cores).

Another research direction is to study scalable solutions for the transactional FSM setting. Existing solutions for this setting share the same problems as those for the single graph setting. The proposed techniques can be applied to the transactional setting after applying minor modifications. Nonetheless, there exist many opportunities for further improvements which are specific to the transactional setting. For instance, the list of input graphs can be ordered, so that graphs with higher chances to include a candidate subgraph are checked first. Coupled with this, evaluation is stopped immediately once the frequency threshold is met. Consequently, checking input graphs that do not contribute to the frequency is avoided, and efficiency is boosted. Furthermore, an interesting property of the transactional setting is that the input data can be easily partitioned. This property allows for mining massive graphs (e.g., tens of billions of edges) without worrying about partitioning. Processing such massive graphs introduces other challenges, such as the huge number of intermediate as well as final results. Such challenges will affect the efficiency of the used algorithms and require novel techniques to deal with them.
REFERENCES


161


