MINING APPROXIMATE FREQUENT PATTERNS FROM GRAPH DATABASES

By

Pranay Anchuri

A Dissertation Submitted to the Graduate Faculty of Rensselaer Polytechnic Institute in Partial Fulfillment of the Requirements for the Degree of DOCTOR OF PHILOSOPHY

Major Subject: COMPUTER SCIENCE

Examinning Committee:

Mohammed Zaki, Dissertation Adviser

Malik Magdon Ismail, Member

Mark Goldberg, Member

T. Ravichandran, Member

Rensselaer Polytechnic Institute
Troy, New York

August 2015
(For Graduation August 2015)
© Copyright 2015
by
Pranay Anchuri
All Rights Reserved
# CONTENTS

LIST OF TABLES ............................................................ vii
LIST OF FIGURES .......................................................... viii
ACKNOWLEDGMENTS ............................................................ x
ABSTRACT ................................................................. xii

1. Introduction ................................................................. 1
   1.1 Background .............................................................. 1
   1.2 Motivation ............................................................... 1
      1.2.1 Configuration management databases ......................... 2
      1.2.2 Protein protein interactions .................................. 2
      1.2.3 Protein 3D structures .......................................... 4
   1.3 Problem setting ......................................................... 4
      1.3.1 Notation .............................................................. 5
   1.4 Problem variations ...................................................... 6
      1.4.1 Types of database ................................................ 7
      1.4.2 Types of patterns ................................................ 7
   1.5 General methodology ................................................... 9
   1.6 Thesis contributions .................................................. 10

2. Related Work ............................................................... 12
   2.1 General pattern mining algorithms ................................. 12
   2.2 Graph mining ........................................................... 13
      2.2.1 Mining from graph databases ................................ 13
      2.2.2 Mining from single large graph ................................ 14
      2.2.3 Approximate graph mining .................................... 16
      2.2.4 Mining probabilistic datasets ................................. 17
         2.2.4.1 Mining probabilistic relational data ................... 17
         2.2.4.2 Graph mining in uncertain graphs ....................... 18
         2.2.4.3 Frequent pattern mining in uncertain graphs ........ 18
         2.2.4.4 Sampling approaches ..................................... 19
   2.3 Graph querying ......................................................... 19
5.2 Preliminaries ........................................... 66
  5.2.1 Coverage computation is polynomial iff $P = NP$ .......... 70
  5.2.2 Complexity assuming an oracle for subgraph isomorphism .. 70
    5.2.2.1 Reduction to coverage problem ..................... 71
5.3 Properties of the coverage function .......................... 72
  5.3.1 Patterns with higher support ............................ 72
  5.3.2 Diverse patterns ..................................... 73
  5.3.3 Non antimonotic property .............................. 74
5.4 Greedy algorithms for maximizing the coverage ............ 74
  5.4.1 Estimating coverage from the pattern embeddings .......... 74
    5.4.1.1 Upper and lower bounds for union of events .......... 76
5.5 Computing the coverage using edge representatives ....... 77
  5.5.1 Edge view of graphs ................................ 78
  5.5.2 Edge representatives .................................. 78
  5.5.3 Estimating coverage from edge representatives .......... 79
    5.5.3.1 Notation ..................................... 79
    5.5.3.2 Approximate coverage ............................ 79
  5.5.4 Intuition behind the $c(x, y)$ approximation ............ 80
5.6 Enumerating high coverage patterns ........................ 81
  5.6.1 Single pattern .................................... 82
  5.6.2 Enumerating $k$ Patterns ............................. 82
  5.6.3 Updated edge probabilities ............................ 83
5.7 Preliminary results ...................................... 83
  5.7.1 Single maximal pattern ............................... 84
  5.7.2 $k$ maximal patterns ................................. 84

6. Summarizing Frequent Patterns ................................. 88
  6.1 Introduction .......................................... 88
  6.2 Pattern similarity ...................................... 89
  6.3 Clustering ............................................ 92
  6.4 Extracting representatives from clusters ................. 93
  6.5 Pattern similarity ..................................... 93
    6.5.1 Comparing similarity metric ........................ 94
7. Future Work ..................................................... 97
  7.1 Applications ............................................. 97
    7.1.1 Anti-money laundering .............................. 97
    7.1.2 Analyzing infrastructure networks ............... 99
  7.2 Meta-heuristic methods for graph isomorphism ....... 99
  7.3 Systems .................................................. 100
  7.4 Evaluation ............................................... 100
  7.5 Mining approximate patterns with edge mismatches ... 101
    7.5.1 Preliminaries ....................................... 101
    7.5.2 Mining label mismatch isomorphisms ............. 101
    7.5.3 Algorithm to mine label and structure-based ap-
         proximate patterns .................................. 103
       7.5.3.1 Processing at a node ......................... 105
       7.5.3.2 Search space exploration .................... 106
       7.5.3.3 Detecting $\phi_{\alpha,0}$ automorphic nodes in
                lattice ..................................... 107
       7.5.3.4 Enumerating connected subgraphs: ............ 108
    7.5.4 Derived vertex for edge deletion case ............ 109
       7.5.4.1 Fixing a pair vs pattern .................... 111
REFERENCES ................................................... 113
LIST OF TABLES

3.1 IP Address related attributes and their values. .......................... 24
3.2 Attribute Entropies. .................................................... 24
3.3 CMDB dataset properties ................................................. 35
3.4 Largest frequent graphs extracted from CMDB graphs ................. 35
3.5 Times for removing duplicate maximal patterns ....................... 38
4.1 k-hop label of vertices 2 and 20 ..................................... 46
4.2 k-hop labels of vertices 3 and 50. ................................. 47
4.3 Dataset and maximal pattern statistics .............................. 56
4.4 Time for random walks in CMDB dataset .......................... 57
4.5 SCOP: Effect of $\alpha$ ..................................................... 59
4.6 SCOP: Runtime of Different Algorithms ............................ 59
4.7 SCOP: Number of derived label checks ............................ 60
4.8 SCOP: Number of extensions tried ................................ 60
4.9 SCOP: Time for Different Steps vs. $\text{minsup}$ .................... 60
4.10 Relative effectiveness of the labels on SCOP dataset ............. 61
4.11 Time for random walks in PPI Dataset ............................ 62
4.12 PPI: Number of derived label checks ............................. 62
4.13 PPI: Number of extensions tried ................................. 63
5.1 Properties of sampled probabilistic graphs .......................... 84
6.1 Variations of simflood algorithm ................................... 94
6.2 CMDB-Miner vs simflood when $I = 2$ .............................. 96
6.3 CMDB-Miner vs simflood when $I = 4$ .............................. 96
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Snippet from a CMDB graph</td>
<td>2</td>
</tr>
<tr>
<td>1.2</td>
<td>PPI network in yeast</td>
<td>3</td>
</tr>
<tr>
<td>1.3</td>
<td>3D structure of myoglobin</td>
<td>5</td>
</tr>
<tr>
<td>1.4</td>
<td>Example database graphs and pattern</td>
<td>6</td>
</tr>
<tr>
<td>1.5</td>
<td>Example of approximate isomorphism</td>
<td>8</td>
</tr>
<tr>
<td>3.1</td>
<td>Example pattern and its isomorphisms</td>
<td>26</td>
</tr>
<tr>
<td>3.2</td>
<td>Flow network for support computation</td>
<td>28</td>
</tr>
<tr>
<td>3.3</td>
<td>Example of edge disjoint embeddings</td>
<td>32</td>
</tr>
<tr>
<td>3.4</td>
<td>Flow network without longest path optimization</td>
<td>32</td>
</tr>
<tr>
<td>3.5</td>
<td>Flow network with longest path optimization</td>
<td>33</td>
</tr>
<tr>
<td>3.6</td>
<td>Attribute Pruning</td>
<td>36</td>
</tr>
<tr>
<td>3.7</td>
<td>Time for sampling maximal patterns in CMDB graphs</td>
<td>37</td>
</tr>
<tr>
<td>3.8</td>
<td>Sampling Time for Company A using Longest Path Optimization</td>
<td>37</td>
</tr>
<tr>
<td>3.9</td>
<td>Times for random walks in artificial datasets</td>
<td>41</td>
</tr>
<tr>
<td>4.1</td>
<td>Notation for the LMiner algorithm</td>
<td>43</td>
</tr>
<tr>
<td>4.2</td>
<td>Pattern (a), database graph (b), and cost matrix (c).</td>
<td>47</td>
</tr>
<tr>
<td>4.3</td>
<td>Flow network for $h_2(2)$ and $h_2(20)$</td>
<td>53</td>
</tr>
<tr>
<td>4.4</td>
<td>CMDB: Approximate Pattern</td>
<td>57</td>
</tr>
<tr>
<td>4.5</td>
<td>Complete Enumeration Expensive</td>
<td>58</td>
</tr>
<tr>
<td>4.6</td>
<td>SCOP: Approximate Pattern and its Structure</td>
<td>61</td>
</tr>
<tr>
<td>4.7</td>
<td>Approximate PPI Pattern and GO Enrichment</td>
<td>63</td>
</tr>
<tr>
<td>5.1</td>
<td>shows an uncertain graph with two edges 5.1a and its four possible worlds</td>
<td>69</td>
</tr>
<tr>
<td>5.2</td>
<td>shows the construction to prove NP-Hardness of the coverage</td>
<td>71</td>
</tr>
<tr>
<td>Section</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>---------</td>
<td>-------------</td>
<td></td>
</tr>
<tr>
<td>5.3</td>
<td>Uncertain graph to show that the objective function prefers diverse patterns.</td>
<td></td>
</tr>
<tr>
<td>5.4</td>
<td>Example showing the edge view of a graph</td>
<td></td>
</tr>
<tr>
<td>5.5</td>
<td>Shows an example of $c(x, y)$ approximation.</td>
<td></td>
</tr>
<tr>
<td>5.6</td>
<td>Coverage for a single random walk</td>
<td></td>
</tr>
<tr>
<td>5.7</td>
<td>Coverage for the optimal set</td>
<td></td>
</tr>
<tr>
<td>5.8</td>
<td>Maximal patterns from greedy algorithm</td>
<td></td>
</tr>
<tr>
<td>6.1</td>
<td>Sample Maximal Patterns</td>
<td></td>
</tr>
<tr>
<td>6.2</td>
<td>Example of augmented for computing similarity</td>
<td></td>
</tr>
<tr>
<td>7.1</td>
<td>Figure showing transactions between three financial institutions</td>
<td></td>
</tr>
<tr>
<td>7.2</td>
<td>Sample database</td>
<td></td>
</tr>
<tr>
<td>7.3</td>
<td>Search space of edge deletions</td>
<td></td>
</tr>
<tr>
<td>7.4</td>
<td>Search space for bounded edge mismatches</td>
<td></td>
</tr>
</tbody>
</table>

Page numbers: 73, 78, 81, 85, 86, 87, 90, 91, 98, 102, 104, 110
ACKNOWLEDGMENTS

It is a great pleasure to thank all the people who made this thesis possible.

First and foremost, I like to thank my adviser, Professor Mohammed Zaki, who has supported since me the end of my undergraduate days. He has taught me, both consciously and unconsciously, how good data mining research is done. I learned a great deal of things from him during our weekly meetings. The turning point for my research was one of the initial meetings with him that extended for nearly 3.5 hours. His passion for research has been my primary motivation since then. I’m highly indebted to him for encouraging me to go on multiple research internships and for writing me reference letters.

I would also like to thank, Prof. Magdon-Ismail, Prof. Goldberg and Prof. Ravichandran for serving as my committee members. A special thanks to Prof. Magdon-Ismail for his valuable suggestions during the time when we were working on the Community Detection problem. I learned a lot from the Computer Science theory courses in our department. In this regard, I would like to thank Prof. Anshelevich, Prof. Drineas, Prof. Goldberg and Prof. Magdon-Ismail.

I was very fortunate to go on multiple research internships during these 5 years. All these internships have played a vital role, either directly or indirectly, in this thesis. I am highly indebted to all my mentors and managers. It was a pleasure to work with Dr. Bonchi and Dr. Gullo at Yahoo! Labs, Barcelona. The chapter on probabilistic graph mining wouldn’t have been possible without their support. I spent a wonderful summer in Mountain View working at LinkedIn. I would like to thank Sam and Roshan for their support when I was working on the Hotspot Detection problem. I also had the pleasure of working twice at IBM Research. A big thanks to my mentors Dr. Berlingerio and Dr. Almasi. My internships wouldn’t have been possible without the wonderful support from Ms. Dawn at the Career center at RPI.

My transition to US was very smooth, thanks to my friends at RPI. I had a great time hanging out with John, Fred, Louis and Yi from the CS department and
I’ll miss the weekly bowling nights, monthly sushi nights and birthday parties! Over
the years I shared house with some of the most fun loving people. It was a great
pleasure to share lab space with Neel, Robert, Hilmi, Geng and Haiqiong. Thanks
for putting up with my messy desk space. Ajay, Bala and I shared some of the
funniest moments at our college avenue apartment. Thanks to Rahul for engaging
in arm chair discussions about politics and for helping me out with paper work at
RPI while I was away from school. Our department co-ordinators Terry and Chris
did a fabulous job in making sure that the department stays connected with various
events all round the year.

My internships have all been wonderful experiences. I made very good friends
during my internships. Mike, the Italian trio (Riccardo, Dani and Davide), Pawel
and Yinlam deserve a huge shout-out for all the good times in Dublin. Thanks also
to Nitish, Farshad and Natali for their awesome company while I was working at
Yahoo.

Living in Doha (during my stay at QCRI) was a very unique experience. I
would like to thank the members of our decent gang. I share a special bond with
Kenton, Luam, Gabriel, Minsu, Suin, Kunwoo and Zhongyu and I hope we all get
to meet in Doha for the 2022 world cup. Thanks to Alexandra for organizing the
local trips and also for her wonderful hospitality when I visited her in Lausanne.
Thanks to Kiran for helping me get started in Doha.

Stuyding at IIIT-Hyderabad was a great learning experience for me. The
research based curriculum at IIIT motivated me to pursue graduate studies. I
would like to thank my adviser Prof. Pudi for helping me to get started in Data
Mining research. I made some very good friends during my undergraduate degree.
In particular, I would like to thank Kalyan, Vijay, Siva, GSK and all the babus
(Meher, Gadde and Mahesh) for their support over the years.

Finally, but by no means least, thanks go to Amma, Nanna, Annayya and
Padmashree for their unbelievable support. They are the most important people in
my world and I dedicate this thesis to them.
ABSTRACT

Graph analytics is the process of discovering patterns and insights from data that can be modeled as graphs. Algorithms for graph analytics fall into two broad categories: Mining and Management. Graph mining algorithms are often used in graph management and vice versa. In recent times, these algorithms have become an indispensable tool for analyzing networks in domains such as i) Computational biology, ii) Infrastructure and mobile sectors, iii) Cybersecurity.

Many real-world graphs have complex labels on the nodes and edges. Mining only exact patterns yields limited insights, since it may be hard to find exact matches. However, in many domains it is relatively easy to compute some cost (or distance) between different labels. Using this information, it becomes possible to mine a much richer set of approximate subgraph patterns.

In this thesis, we present methods for mining approximate frequent patterns from a single large graph. The approximate pattern mining algorithms that we consider in this thesis cover two main categories: Firstly, instead of computing the exact support of a pattern we approximate the support by an upper-bound. Secondly, we tolerate bounded label and structural mismatches when finding matches of a pattern in the database. A common theme across both these paradigms is that the problems are NP-Hard to even approximate up to a constant factor of the optimal value. We formally prove the exponential complexity of all these problems and also present approximation algorithms for solving the problems at scale.

Approximate Support: The problem of mining frequent subgraph patterns from a database of graphs is a well studied problem in the literature. However, the methods proposed for the multiple input graphs scenario cannot be directly extended for mining patterns from a single large graph. This is due to the fact that the support function, is defined in terms of all the embeddings of a pattern, which can be exponential. We propose a network-flow based approach that gives a polynomial time approximation for the support function. We also propose a three step procedure to summarize the output of graph mining algorithms.
Approximate Patterns: We propose three models for approximate matching of patterns in a given input graph. First, we allow for bounded label mismatches of the pattern. To find these approximate matches, we proposed a neighborhood-label based algorithm that can efficiently prune infeasible matches of the pattern. Second, we allow both bounded label and structural mismatches of the pattern. To efficiently find such approximate matches, we repeatedly use our label-only algorithm on a specially constructed subgraph of the pattern. Finally, we discuss why the existing models cannot be adapted to mine interesting patterns from uncertain graphs where edges are associated with a probability of existence. Therefore, we propose coverage based pattern mining that is a novel way to think about pattern mining in uncertain graphs. Our algorithm essentially enumerates a set of patterns that covers distinct regions of the input graph with high probability.

We apply our models and methods on several real-datasets such as: 1. Configuration management databases representing the infrastructure entities and their inter-relationships in large IT companies. 2. Protein-Protein interaction network in organisms such as yeast and graphs representing 3D structure of proteins.
CHAPTER 1
Introduction

1.1 Background

Frequent pattern mining (FPM) is the task of finding repetitive substructures in the data. It is one of the fundamental and widely studied data mining problems. It forms the foundation of many other data mining tasks including classification, cluster analysis, rule mining etc. One of the commonly referred association between beer and diapers in the transaction data of a convenience store results from the application of FPM.

Traditionally, FPM comes in four different flavors: 1. Itemset Mining - finding frequent subset of items from lists of items, 2. Sequence Mining - finding frequent subsequence patterns from lists of sequences, 3. Tree Mining - finding frequent subtree patterns from trees, and 4. Graph Mining - finding subgraphs from graph(s). Several variations of these problems have been proposed in the literature.

Abstract Problem Statement: In general, FPM problem can be stated as: Given a set of \( n \) data points \( D = d_1, d_2, \ldots, d_n \) and a parameter \( \theta \), \( 0 < \theta \leq 1 \), find all objects \( p \) that are contained in at least \( \theta \) fraction of \( n \) data points. Often, the data points and the objects belong to the same abstract class. An object \( p \) is called a frequent pattern if its support, defined by the fraction of points that contain \( p \), is greater than a user given threshold \( \theta \), called minimum support.

For example, in the case of graph mining, data points represent a database of graphs and frequent patterns are labeled graphs that maintain a subgraph isomorphism relationship with at least \( \theta \) data points.

1.2 Motivation

We motivate the need for frequent subgraph mining using three real world examples.
1.2.1 Configuration management databases

A configuration management database (CMDB) is used to manage and query the IT infrastructure of an organization. It stores information about the so-called configuration items (CIs) – servers, software, running processes, storage systems, printers, routers, etc. As such it can be considered to be a single large multi-attributed graph, where the nodes represent the various CIs and the edges represent the connections between the CIs (e.g., the processes on a particular server, along with starting and ending times). Fig. 1.1 shows a snippet from a real-world CMDB graph, displaying only type labels. A CMDB provides a wealth of information about the largely undocumented IT practices of a large organization, and thus mining the CDMB graph for frequent subgraph patterns can reveal the de facto infrastructure patterns. Once mined, these patterns can be used to either set the default IT policies, or refine them if found unsatisfactory. Thus, the discovery of infrastructure patterns is an important real-world application of subgraph mining in IT domain.

![Figure 1.1: Snippet from a CMDB graph](image)

1.2.2 Protein-protein interactions

Proteins in a biological cell interact via an extensive network. These interactions affect all the processes in the cell and play a fundamental role in many
diseases. Therefore, understanding the the protein-protein interactions (PPI) helps in improving our understanding of the diseases.

Modern molecular biology has developed high throughput techniques to detect the interactions with high sensitivity and specificity. For example, the Fig. 1.2 shows the interactions between proteins in yeast [1] represented as a network. Each circle represents a protein and the interacting proteins are connected by a line. Frequent subgraph pattern in the PPI network corresponds to a set of proteins involved in a recurring biological process.

Figure 1.2: Interaction between proteins in yeast (Saccharomyces cerevisiae)
1.2.3 Protein 3D structures

Cells in all known living organisms contain DNA which is converted to RNA to build the proteins. DNA is a long sequence of connected molecules of one of the four types: A, C, T, and G. Every subsequence of 3 molecules encodes for an amino acid and the sequence of amino acids denotes the primary structure of the protein. The chain of amino acids folds along itself to form a complex 3D structure. The folds lead to the secondary structure of the proteins. More complex folds are represented by the ternary structure. The 3D structure of a protein can be represented as a network of interconnected amino acids. Each amino acid is connected to other spatially close amino acids.

Protein Data Bank (PDB) [2] is a publicly available repository containing the structural information of large biomolecules such as proteins. Proteins structures often exhibit high structural similarity. The Structural Classification of Proteins (SCOP) [3] database provides access to the structural similarities between all the proteins available in the PDB. In the SCOP database proteins are grouped into a four level hierarchy. Finding frequent substructures in the 3D structure of the proteins in different/same hierarchies will enable us to form associations between the structure and the functionality of the protein. Therefore, methods in this domain will further our understanding of the structure of proteins.

1.3 Problem setting

The problems mentioned in section 1.2 belong to different domains but the common feature among them is that they model relationships between different objects. The CMDB graph models relationship between IT entities. The PPI network models the interaction between the proteins and SCOP models the spatial relationship between amino acids in the 3D protein structure.

A graph is an abstract concept that captures relationships between a set of objects. A simple graph is defined by specifying the objects and the relationships between the objects. Additionally, each entity and relation can have attributes. For example, in a CMDB graph name of an IT entity is an attribute for the object representing the entity. We begin by introducing some mathematical notation to
1.3.1 Notation

Let \( \Sigma \) denote a given set of labels. A \textit{labeled simple graph} is a triple \( G = (V, E, L) \), where \( V \) is the set of vertices or nodes, \( E \subseteq V \times V \) is the set of (unordered) edges, and \( L \) is the labeling function for both nodes and edges, so that \( L(v) \) is the label of a node \( v \), and \( L(e) = L(a, b) \) is the label of an edge \( e = (a, b) \). The origin and destination of an edge \( e = (a, b) \) are \( a \) and \( b \) respectively. The \textit{order} of the graph is the number of nodes \(|V|\), and the \textit{size} of the graph is the number of edges \(|E|\).

We say that \( G' = (V', E', L') \) is a \textit{subgraph} of \( G = (V, E, L) \), denoted \( G' \subseteq G \), if there exists a 1-1 mapping \( \pi : V' \rightarrow V \), such that \( (v_i, v_j) \in E' \) implies \( (\pi(v_i), \pi(v_j)) \in E \). Further, \( \pi \) must preserve vertex and edge labels, i.e., \( L(v_i)' = L(\pi(v_i)) \) for all \( v_i \in V' \), and \( L(v_i, v_j)' = L(\pi(v_i), \pi(v_j)) \) for all edges \( (v_i, v_j) \in E' \). The mapping \( \pi \) is called a \textit{subgraph isomorphism} from \( G' \) to \( G \). If \( G' \subseteq G \) we also say that \( G \) \textit{contains} \( G' \). If \( G' \subseteq G \) and \( G \subseteq G' \), we say that \( G \) and \( G' \) are \textit{isomorphic}. In this work, we restrict ourselves to \textit{connected, simple graphs}.

\textbf{Database:} The input database, denoted by \( D \), is a set of \( n \) connected graphs.
$G_1, G_2, \ldots, G_n$. 

**Pattern:** A pattern, denoted by $P$, is also a connected labelled graph.

**Support:** There are several ways to define the support of a pattern. Most commonly used definition is the fraction of graphs in the $D$ that have $P$ as a subgraph. Therefore, support of $P$, $Sup(P)$ is

$$Sup(P) = \frac{|\{G_i | P \subseteq G_i\}|}{n} \quad (1.1)$$

The support can also be defined in terms of the absolute number of database graphs which has pattern $P$ as a subgraph.

**Frequent pattern and Minimum support:** A pattern $P$ is called frequent if its support, $Sup(P)$, is at least the value of minimum support, denoted by $\theta$, which is part of the user input.

Figure 1.4 shows a database $D$ of 3 graphs and a sample pattern $P$. Pattern $P$ is a subgraph of $G_1$, $G_2$. The support, $Sup(P) = \frac{2}{3}$. The pattern $P$ is called frequent if the user defined value of minimum support, $\theta$, is less than $\frac{2}{3}$.

![Figure 1.4: Example database graphs and pattern](image)

**Frequent Subgraph Mining:** Given a database $D$ of graphs and a minimum support $\theta$. Find all frequent patterns.

### 1.4 Problem variations

We describe the variations of the subgraph mining problem that we deal in this thesis. The alternatives are at two different levels: a) Types of database. b) Types of frequent patterns.
1.4.1 Types of database

The input graph database can be a set of labeled graphs or a single large graph. The definition of the support has to be changed when mining from a single graph because the support as defined in equation 1.1 results in a $0 - 1$ support for all the candidate patterns. By fixing $\theta = 1$, every possible subgraph of the input graph is frequent. This definition of support cannot differentiate truly recurring subgraphs from the rest of frequent patterns with support equal to 1. An alternative for single graph case is counting the number of isomorphisms from the pattern into the database. As we will see later, counting the number of isomorphisms can be of exponential complexity.

1.4.2 Types of patterns

Relaxing the conditions for the existence of subgraph isomorphism will help us discover more general and interesting patterns that cannot be found otherwise. Two relaxations that we consider are label and structural mismatches.

*Label mismatch relaxation:* Assume we can associate a cost for matching any two given vertex labels. These costs can be represented by a cost matrix, $C$. The cost between labels $i$ and $j$ is given by the $ij^{th}$ entry in the matrix, $C[i][j]$. The label relaxation allows bounded label mismatches between the vertices in the pattern and their mappings in the database. There is no restriction on the number of vertices for which the label is preserved under the isomorphism.

*Label, Structure mismatch relaxation:* This relaxation allows for some of the pattern edges to be missing in the database under the isomorphism mapping. The only restriction is that the mapping of vertices are connected in the database. This helps to prune out trivial and uninteresting patterns from the output of the algorithm.

Obviously, more relaxed the conditions are more general are the frequent patterns that the mining algorithm discovers. The relaxed versions of the isomorphisms are called approximate isomorphisms which we define below:

*Approximate subgraph isomorphism:* Given a database graph $G$, a cost matrix $C$, a pattern $P$, and label and structure mismatch thresholds denoted by $\alpha$ and
respectively, we define a function $\phi_{\alpha,\beta} : V_P \rightarrow V_G$ as an approximate subgraph isomorphism of $P$ in $G$ if and only if:

- $\phi_{\alpha,\beta}$ is an injective function.
- The label mismatch cost of $\phi_{\alpha,\beta}$ defined by $C(\phi_{\alpha,\beta}) = \sum_{u \in V_P} C[L(u)][L(\phi_{\alpha,\beta}(u))]$ is at most $\alpha$ i.e., the total cost of matching the labels of pattern vertices and its mapping in the graph $G$ doesn’t exceed the threshold $\alpha$.
- The edge mismatch cost of $\phi_{\alpha,\beta}$ defined by $E(\phi_{\alpha,\beta}) = | \{(u, v) | (u, v) \in E_P, (\phi_{\alpha,\beta}(u), \phi_{\alpha,\beta}(v)) \notin E_G \} |$ is at most $\beta$. In other words, the number of edges $(u, v)$ for which their mappings in $G$ are not connected cannot exceed the edge mismatch threshold $\beta$.

The isomorphism defined $\pi$ as defined in section 1.3.1 is equivalent to $\phi_{0,0}$ and the label relaxed isomorphism is $\phi_{\alpha,0}$ for some $\alpha \geq 0$. Figure 1.5 shows a database (1.5a), a pattern (1.5b), label costs (1.5c) and two approximate isomorphisms of the pattern in the database. In the isomorphism $\phi_1$, the structure mismatch cost, $\beta$, is 1 because the mappings of the edge (1, 3) are not connected by an edge in the
database. In $\phi_2$, the label mismatch cost $\alpha = 4$, comes from the mismatch between the labels of vertices 2, 4 and that of their mappings 10, 30 in the database.

1.5 General methodology

Most of the frequent mining algorithms have two basic steps, namely a) Candidate Generation, and b) Support Computation.

Candidate Generation: The set of all possible frequent subgraphs of the database graphs forms a partial order with $P = \emptyset$ at the top. The candidate generation step enumerates potential frequent patterns by traversing the partial order from the top. The most commonly used exploration strategies are: 1. Depth first, 2. Breadth first, and 3. Random extensions. Size of the partial order is exponential but the search process can be made efficient by utilizing the downward closure property of the support function. In other words, the moment any subgraph is found infrequent the entire subtree under it in the search lattice can be removed from further consideration. As we will see, it is not trivial to come up with such a support function in all variations of the problem.

Support Computation: To check if a generated candidate pattern is frequent, its support in the database has to be computed. The candidate pattern is frequent if its support is at least $\theta$. This step is the computational bottleneck in most of the pattern mining algorithms.

Algorithm 1 shows various steps in our generic subgraph mining algorithm. By varying the values of $\alpha$ and $\beta$, frequent patterns with different properties as described in section 1.4.2 will be discovered. In general, we start from the empty graph and extend the current candidate pattern by a random edge. After each extension, we ensure that the pattern is frequent with the given mismatch thresholds. The majority of the thesis is focused on line 8. The steps annotated with [Optional] tag
are used when we require the output to be refined without duplicates.

**Algorithm 1**: General algorithm for mining frequent subgraph patterns

**Data**: Database $D$, Cost matrix $c$, Label mismatch threshold $\alpha$, Structural mismatch threshold $\beta$ and minimum support $\theta$.

**Result**: frequent interesting patterns in $D$

1. $\mathcal{F} \leftarrow \emptyset$;
2. while **Termination condition is not met** do
3. $P \leftarrow \emptyset$;
4. while $P$ is frequent do
5. for Extensions $e$ of $P$ do
6. $P' \leftarrow P \cup e$;
7. [Optional]: Continue only if $P'$ is not yet explored.;
8. $s \leftarrow \text{Support of } P' \text{ in } D \text{ under the } \alpha \text{ and } \beta \text{ conditions};$
9. if $s \geq \theta$ then
10. $P \leftarrow P'$;
11. break;
12. $\mathcal{F} \leftarrow \mathcal{F} \cup P$
13. [Optional]: $\mathcal{F} \leftarrow \mathcal{F}$ with duplicates removed;
14. [Optional]: $\mathcal{F} \leftarrow \text{Representative patterns from } \mathcal{F};$
15. return $\mathcal{F}$;

**1.6 Thesis contributions**

Our main contribution in this thesis is efficient algorithms for mining subgraph patterns with different properties. The primary focus is on the support computation step which is computationally the most intensive step.

*Mining from single large graph*: In chapter 3, we describe an algorithm to mine exact frequent patterns from a single large graph. We show that the algorithm discovers interesting patterns from CMDB data.

*Approximate mining with label mismatches*: The algorithm for mining patterns with bounded label mismatches is described in chapter 4. We present results on
PPI and SCOP data apart from CMDB dataset.

Mining patterns from uncertain graphs: In addition to handling approximate structural and label mismatches, we also present algorithms for mining patterns from uncertain graph databases. Edges in an uncertain graph are associated with a probability of existence. In chapter 5, we propose a new way of looking at pattern mining in uncertain graphs and show that it can extract highly recurring, diverse patterns.

Summarizing frequent patterns: Frequent pattern mining algorithms usually produces many frequent patterns. For a domain expert to understand these patterns it is important that the output of the mining algorithms is summarized. In chapter 6, we present an algorithm to summarize frequent patterns discovered in the CMDB graphs. The algorithm can be easily applied to the output of any other subgraph mining algorithm.

Approximate mining with structural and label mismatches: We present an algorithm that allows for bounded structural and label mismatches. In chapter 7, we describe a strategy for mining these approximate isomorphisms.
CHAPTER 2
Related Work

2.1 General pattern mining algorithms

Mining frequent itemsets and associations between the items is first problem tackled in the field of frequent pattern mining. Agarwal and Srikant proposed Apriori algorithm [5] to mine association rules from transaction data. In general, any apriori like method generates candidate patterns of length \( k + 1 \) by combining two patterns of length \( k \) having a common \( k - 1 \) length subpattern. The search space of the frequent patterns can be modeled as a layered lattice. Therefore, apriori based methods are usually classified as breadth first enumeration methods.

Eclat algorithm [6] proposed by Zaki et al., is a depth first algorithm for mining association rules. It uses a vertical representation of the database and requires a single pass over the database for enumerating all the association rules. The FP-growth algorithm by Han et al. [7] is another efficient method for mining complete set of frequent patterns. It first passes twice over the database to construct a compressed data structure called FP-Tree. In the second step it extracts the frequent itemsets directly by traversing the FP-tree. For a detailed explanation of the algorithms refer to the survey by Goethals [8].

For mining sequential patterns, Agarwal and Srikant [9] proposed apriori based methods that scale linearly with the number of records in the database. Zaki proposed SPADE algorithm [10]. The sequences and the transaction list is stored in a vertical id-list format. The SPADE algorithm employs a divide and conquer strategy to mine the search lattice. The PrefixSpan algorithm by Pei et al. [11] avoids generating potentially exponential number of candidates like other apriori based sequence mining algorithms. It also reduces the number of database scans required to generate the complete set of frequent sequences.
2.2 Graph mining

Graph mining comes in several different flavors. In this section, we present related work on several related graph mining problems. Algorithms for these problems differ in the types of data they can handle and also the type of patterns they can extract from the input data.

2.2.1 Mining from graph databases

Frequent pattern mining algorithm in graph data is theoretically harder compared to pattern mining in other structured data sources like sets, sequences and trees because checking for subgraph isomorphism is an NP-Complete problem [12]. It means that there is no polynomial time algorithm for subgraph isomorphism unless P=NP. On the other hand the complexity class of the graph isomorphism problem is not yet known. Most of the graph mining algorithms in the literature compromise with approximate algorithms for practicality of the algorithm. Our algorithms follow the same approach: they are highly scalable but potentially returns false positives.

One of the first substructure detection algorithms in graph data was proposed by Cook and Holder as part of the SUBDUE system [13]. SUBDUE finds repetitive patterns from the graph data based on the concept of Minimum Description Length introduced by Rissanen [14]. The system discovers substructures that compress the original data by maximum. As a side effect it can build a concept hierarchy in the input data based on the substructures. The method has been successfully applied to chemical compounds data, CAD circuits etc.

FSG algorithm developed by Kuramochi and Karypis [15] is the first apriori based algorithm for frequent subgraph mining. In FSG, candidate patterns with \( k + 1 \) edges are generated by combining two frequent patterns with \( k \) edges having \( k - 1 \) edges in common. To reduce the number of duplicate candidate generated, FSG computes the cores of frequent patterns and joins them efficiently. Moreover, it uses the downward closure property of the support function to further reduce the number of candidates generated.

The gSpan algorithm by Yan and Han [16] is one of the fastest algorithms for
frequent subgraph discovery. Unlike FSG that enumerates the frequent patterns in a breadth first order, gSpan explores the search space in a depth first manner. It achieved significant speedups compared to other algorithms at the time because it combines the candidate generation step and the frequency computation steps. It avoids duplicate candidate generation by using a canonical code (DFS code) and right most path extensions. The DFS code of a graph is constructed from a depth first traversal starting at some vertex. A complete order is defined on the DFS codes in such a way that two graphs are isomorphic if and only if their minimum DFS codes are identical.

GASTON [17] developed by Nijjsen and Kok works particularly well for molecular graph databases. It first enumerates paths and tree like patterns before graphs with cycles thereby avoiding the costly graph isomorphism step on structures without cycles. Another difference compared to the gSpan algorithm is that it stores the embedding list of the frequent patterns. By using the embedding lists it only generates candidates that appear in the database.

Other algorithms for mining from graph databases include [18], [19], and [20]. Refer to the survey [21] for a more complete list of relevant graph mining algorithms.

### 2.2.2 Mining from single large graph

The methods described in section 2.2.1 assume that the database contains a set of graphs, and cannot be directly applied when the database is just a single large graph. This is because they define pattern support to be the number of graphs in the database that contain the pattern. As long as a single embedding is found, the support can be incremented by one, and as such these methods do not have to deal with the problem of enumerating all the embeddings, or computing the maximum number of edge (or node) disjoint embeddings. Also, pattern support, as defined for a database of many graphs, is anti-monotonic, i.e., a super-graph cannot have support more than any of its subgraphs. This property allows for fast pruning of candidate patterns during pattern search, since we can prune a pattern (and all of its extensions), when its support falls below a user specified minimum support threshold, $\minsup$. However, the number of embeddings is clearly not anti-
monotonic. For example, let $\text{minsup} = 3$, and let the database graph comprise a node labeled $A$, connected to two nodes labeled $B$, and further, let each of the $B$ nodes be connected to three nodes labeled $C$. In this database graph, the edge $A-B$ has two embeddings (below $\text{minsup}$), but the pattern $A-B-C$ has six embeddings (above $\text{minsup}$). The lack of anti-monotonicity is clearly a problem for support computation.

Several recent approaches have been proposed to tackle the challenges in mining a single graph. Kuramochi and Karypis [22] proposed a support counting measure that is anti-monotonic. They proposed three different formulations for mining a single graph. The first is based on an exact maximum independent set (MIS) of the overlap graph, which gives the exact set of edge disjoint embeddings. The other two approaches are based on approximate MIS, which provide a subset and superset of the edge disjoint embeddings. However, they require enumerating all the embeddings and then discarding the ones that overlap. Since the total number of possible embeddings is exponential, it makes these methods incapable of finding bigger patterns. Further, instead of the MIS, we propose a network-flow based approach. Fiedler and Borgelt [23] gives a formal proof that maximum independent set based support counting is anti-monotonic. We also prove that our flow-based upper-bound on the number of edge disjoint embeddings leads to an anti-monotonic pruning criteria. Bringmann and Nijssen [24] proposed an image-based support of a pattern, defined as the minimum number of mappings from a vertex in the pattern to a vertex in the graph. Our flow-based approach yields a tighter upper bound compared to the image-based support. We used this notion of support in our approximate mining methods [25], [26] and [27]. Chen et al. [28] proposed a $g\text{Approx}$ to mine frequent approximate patterns from a single large network. They defined the pattern space and proposed strategies to explore it. The support of the pattern is same as the image-based support and requires all the approximate embeddings of the pattern. Li et al. [29] proposes a method to compute edge disjoint support to find frequent dense subgraphs in a single graph. This method is not suitable patterns that are not very dense. Besemann and Denton [30] tackles graphs in which nodes have multiple attributes. The edge disjoint support is computed by constructing a
bipartite graph with the original node set \((V)\) and a node for every attribute \((U)\). Edge disjointness is imposed on \(V\), allowing for overlap in the bipartite edges that connect a vertex in \(V\) to one of its attributes in \(U\). Recent theoretical work has focused on proving necessary and sufficient conditions for anti-monotonicity of edge overlap based graph support measures [31], and in other generalizations such as homomorphisms and isomorphisms, for labeled and unlabeled, directed and undirected graphs [32].

2.2.3 Approximate graph mining

There has been little work in approximate subgraph mining. In [28], they proposed \(g\text{Approx}\) to mine approximate frequent subgraphs. The degree of approximation between a pattern and its isomorphism includes label mismatches and missing edges. The search space is explored in a depth first order and the support of a pattern is computed by enumerating all its isomorphisms. This approach is not feasible for large graphs with label multiplicities as there are potentially an exponential number of isomorphisms [26]. In [33], they proposed APGM to approximate frequent subgraphs from a database of graphs. The method is similar to the \(g\text{Approx}\) method, with the main difference being that the entire 1-hop neighborhood of the current embeddings is explored to enumerate all extensions of the frequent pattern and their corresponding embeddings, whereas \(g\text{Approx}\) enumerates the embeddings for a single extension in each step. However, they store the complete set of approximate embeddings of the current frequent pattern, which can be a problem. In [34], the authors proposed strategies to speed up the existing approximate mining algorithms, by limiting the number of candidates and also the number of duplicate checks performed. They assume that the underlying algorithm takes care of the label and/or edge mismatches. In [35], they proposed a randomized algorithm to mine approximate patterns from a database graphs. This method only handles edge mismatches and not label costs.

The concept of using derived labels based on the structure and the attributes is frequently used in detecting graph isomorphism [36] and computing graph kernels [37, 38]. Our methods to prune representative sets are somewhat similar to the
wiesfieler lehman kernel to test isomorphism between two graphs $G_1$ and $G_2$ [39], where after every iteration the labels are sorted and renamed with a different string in such a way that the pairwise relationship is maintained. $G_1$ is not isomorphic to $G_2$ if the label set of the graphs differ. The specifics of how and what information we update is different, and also we use the information for subgraph isomorphism instead of graph isomorphism.

2.2.4 Mining probabilistic datasets

Recently, there has been lot of work on uncertain data. Specifically, there is lot of interest in extending traditional graph problems to uncertain graph data. We now discuss some of these general graph problems and frequent subgraph mining in the context of uncertain graphs.

2.2.4.1 Mining probabilistic relational data

Much works has been done to deal with uncertainties in database systems. In [40], the authors have proposed a database system that supports queries with uncertain predicates. Each tuple in the probabilistic database is assigned a probability. A probabilistic database induces a probability distribution over all possible database instances considering each tuple as an independent event. Their method retrieves a ranked list of tuples that match an input query under possible world semantics. The Trio system developed at Stanford [41] manages not only the data but also its lineage and accuracy. Trio can be used in several application domains including data cleaning, information extraction systems and approximate query processing. Soliman et al. [42] developed a method for processing top-k queries in uncertain databases. The top-k problem is modeled as that of a state space search and efficient algorithms have been proposed to explore this space. Most of the uncertain data processing solutions assume independence between the events. Lakshmanan et al. [43] proposed a unified framework that takes into account the known dependencies between the events. The ProbView system developed by the authors allows the user to pick a different strategy on a session and operator basis.
2.2.4.2 Graph mining in uncertain graphs

There exist efficient algorithms for some of the graph problems ported to the uncertain case. Zou et al. [44] defined the problem of maximum cliques in uncertain graphs. They proposed a branch and bound approach to compute $k$ cliques of maximum probability. Liu et al. [45] proposed an algorithm to detect reliable clusters from an uncertain graph. The method uses ideas from information theory for representing the intermediate clusters. Parchas et al. [46] proposed a method to extract representative instances of an uncertain graph. The method tries to preserve structural properties of the input graph in the extracted representatives. Khan et al. [47] proposed a method for computing the reachable set of nodes from a given node in an uncertain graph. They developed an novel indexing technique, $RQT_{Tree}$, to efficiently compute the sets of reachable nodes.

2.2.4.3 Frequent pattern mining in uncertain graphs

Zhou et al. published a series of works to compute frequent subgraph patterns from graph databases. These works mainly differ in the definition of a frequent subgraph in an uncertain graph(s). In [48], a random variable is defined to indicate the support of a candidate pattern in a specific instantiation of the uncertain graph. The support is then defined by the expected value of this random variable over all possible instantiations of the given uncertain graph. A candidate pattern is considered as frequent if the expected support is greater or equal to a given $\theta$ threshold. They showed that the problem is $\#P$-Complete and proposed an algorithm to approximate the support of a pattern. The approximation works by first computing all embeddings of the candidate pattern and then the support is estimated by formulating the problem as DNF counting where each embedding corresponds to a boolean formula.

In [49], support of a pattern is defined as the probability that it is frequent in an instantiation of the given uncertain graph database. They proposed a dynamic programming based solution to estimate the support. In [50], the authors have proposed a method to speedup the computation of support in uncertain graph databases. The method uses an index to reduce the number of comparisons made.
during the search for frequent subgraph patterns.

### 2.2.4.4 Sampling approaches

Frequent pattern mining algorithms usually return a large number of patterns and interpreting them is a big challenge. This is especially true if the output is presented to a human user for further analysis. Sampling approaches like those proposed in [26,35,51] mine a representative set of maximal patterns from a database of graphs or a single graph. In our work, we perform a random walk in the search space to enumerate a maximal pattern.

### 2.3 Graph querying

Graph querying is another problem that is related to subgraph mining. The goal is to find matches of a given query graph in a single graph or database of graphs. In [52], they proposed an indexing method to extract the approximate occurrences of a given graph query in large graph databases. In [53], they proposed a polynomial time algorithm for detecting isomorphism between spectrally distinguishable graphs. An isomorphism, if it exists, is obtained by matching the steady state vectors of Markov chains in both the graphs. Giugno et al. developed GraphGrep [54] method to find all occurrences of a query graph in a database of graphs. They used a fingerprinting based technique to prune invalid matches of a given query graph. Zhang et al. proposed SAPPER [55] to retrieve approximate matches of a query graph in large graphs. Their method allows for missing edges in the database graph. Wang et al. [56] proposed a graph kernel method, G-Hash, for similarity search in large graph databases. Instead of explicitly embedding the graphs into a vector space, G-Hash tries to enrich the label of a vertex with information about its neighborhood. There have been methods to find approximate matches between graphs of similar sizes. Such methods are of high interest in the Bioinformatics domain. Koyuturk et al. [57] proposed a method to compare PPI networks in two different species. Pinter et al. [58] developed MetaPathwayHunter, a metabolic pathway querying tool. The authors proposed a novel graph matching algorithm that finds approximate matches to a query pathway in a collection of pathways. These meth-
ods differ from our methods in the sense that we are trying to find many matches of a relatively small pattern in a larger graph. The problem with the indexing approaches is that they are efficient in retrieving a single match for the query graph but fail at retrieving all matches, and thus are not suited to mine frequent patterns. Furthermore, they assume that the query is given, and thus they do not perform pattern enumeration as required in graph mining.
CHAPTER 3
Mining from Single Large Graph

We propose a new approach for mining/sampling maximal subgraph patterns from a single database graph. In particular, we propose a new network-flow based definition of graph support which is an upper-bound on the number of edge-disjoint embeddings, and which allows us to prune patterns the moment they become infrequent. We also propose an optimization to get a tighter upper-bound on the number of edge-disjoint embeddings. We further propose a fast filter-based approach for eliminating isomorphic (i.e., duplicate) patterns.

We describe CMDB-Miner, a system for mining infrastructure patterns from CMDB graphs. It has three components: 1. Preprocessing the graph, 2. Sampling maximal patterns, and 3. Summarizing maximal patterns.

3.1 Preliminaries

Let $G = (V, E, L)$ be a single large database graph, and let $P = (V', E', L')$ be a candidate pattern, whose support we want to compute. Let $\pi$ be a subgraph isomorphism from $P$ to $G$. The sequence $\pi(v_1), \pi(v_2), \ldots, \pi(v_n)$ over all $v_i \in V'$ is called an embedding of $G'$ in $G$. For an edge $e_i = (a_i, b_i) \in E'$, define $\pi(e_i) = \pi(a_i, b_i) = (\pi(a_i), \pi(b_i)) \in E$. The sequence $\pi(E') = \pi(e_1), \pi(e_2), \ldots, \pi(e_m)$ over all edges $e_i \in E'$ is called an edge mapping of $P$ in $G$. For example, given the database graph $G$ in Fig. 3.1a, and the candidate pattern $P$ in Fig. 3.1b, the subgraph isomorphism $\pi_3$ from $P$ to $G$ specified by the mapping $p_0 \to g_0$, $p_1 \to g_9$, $p_2 \to g_7$, $p_3 \to g_{11}$, corresponds to the embedding $0, 9, 7, 11$, and the edge map-

* Portions of this chapter previously appeared as
ping \((0, 9), (9, 7), (7, 11), (11, 9)\). Since \(\pi\) uniquely specifies the embedding and edge mapping, we use these terms interchangeably.

There are several ways to compute the number of occurrences, called the support, of \(P\) in \(G\). The most straightforward definition is to define the support of \(P\) as the number of possible embeddings of \(P\) in \(G\), denoted \(\text{sup}_a(P)\). Figure 3.1c shows all the possible embeddings of \(P\) in \(G\). There are ten embeddings of \(P\) in \(G\) for this example, thus \(\text{sup}_a(P) = 10\). Unfortunately, there can be exponentially many embeddings of a pattern in the database graph. For example, if \(G = P = K_n\), where \(K_n\) is the complete graph on \(n\) nodes, with all node and edge labels being the same, then there are \(n!\) distinct embeddings of \(P\) in \(G\). Unfortunately, due to the label multiplicities in the CMDB graphs, this is a real problem in this application. To avoid the combinatorial blowup, support can also be defined as the maximum number of node or edge disjoint embeddings of \(P\) in \(G\), denoted \(\text{sup}_n(P)\) and \(\text{sup}_e(P)\), respectively. Let \(\Pi\) be the set of all possible embeddings of \(P\) in \(G\). We say that two embeddings \(\pi, \pi' \in \Pi\) are node disjoint, if \(\pi(v_i) \neq \pi'(v_j)\) for all nodes \(v_i, v_j \in V'\). We say that \(\pi\) and \(\pi'\) are edge disjoint if \((\pi(v_i), \pi(v_j)) \neq (\pi'(v_a), \pi'(v_b))\) for all edges \((v_i, v_j), (v_a, v_b) \in E'\). Figures 3.1d and 3.1e show examples of a maximum set of edge and node disjoint embeddings, respectively. These sets are not unique; for example, the embedding set \(\{\pi_0, \pi_3, \pi_9\}\) is also edge-disjoint. However, the edge-disjoint support of \(P\) is \(\text{sup}_e(P) = 3\), and the node disjoint support is \(\text{sup}_n(P) = 2\). Finding the maximum number of edge (or node) disjoint embeddings is equivalent to finding the maximum independent set (MIS [59]) in an embeddings graph, where each embedding is a node, and there exists an edge between two embeddings if they share an edge (or node). Unfortunately, the MIS problem is known to be NP-hard, and thus both the edge and node disjoint embeddings are expensive to compute. One of the novel aspects of CMDB-Miner is that we approximate the edge-disjoint support via a network-flow based approach. We prefer edge-disjointness, since node-disjointness is more constrained (every node-disjoint embedding is also an edge-disjoint embedding, but not vice-versa).
3.2 CMDB-Miner: Overview

CMDB-Miner has three main steps. Given the particular characteristics of CMDB graphs, first we pre-process them to extract the relevant attributes for each configuration item, and summarize the graph. Second, we perform random walks in the pattern space to extract a sample of the maximal frequent patterns. In the third step (discussed in chapter 6), we cluster the maximal patterns (since many of them may be similar) and extract a set of representative patterns from each cluster. The latter constitute the infrastructure patterns presented to the IT practitioners to help manage and set the IT configuration policies throughout the organization. The details of the first two steps are given below.

3.3 Graph Pre-processing

CMDB graphs have many different types of composite items, and each CI may have many possible attributes (with various values). Furthermore, there are many degree one nodes, called leaf nodes in CMDB graph. Before mining these graphs, we preprocess them in two ways to aid in interpretation and mining. First, we prune attributes based on their entropy, and second, we summarize the multiplicities among the leaf nodes.

**Entropy-based Attribute Pruning:** Based on the distribution of values for each attribute, we observed that across the various instances of the same CI type, some of the attributes either have a single value, or they have all distinct values. Let $p_v = \frac{m_v}{m}$ be the probability of observing value $v$ for an attribute $a$ of a given CI type, where $m$ is the total number of occurrences of attribute $a$, and $m_v$ is the number of times $a$ has value $v$. The entropy of $a$ is defined as $E(a) = -\sum_v p_v \log_2 p_v$. We prune the uninformative attributes, namely those that have very low or very high entropy, by discarding the tails of the entropy distribution (e.g., discarding attributes within the bottom 5% and top 5% of entropy). This results in a significant reduction in the number of attributes. Table 3.1 shows instances of an IP Address CI, with three attributes. The DNS Server has the same values, whereas the IP Address has all distinct values across the instances. The entropy of these attributes are shown in Table 3.2. DNS Server, and IP Address will both be pruned, since the former has
very low and the latter very high entropy.

**Summarizing Leaf Nodes:** A peculiarity of CMDB graphs is that a vast majority of nodes are *leaf nodes*, defined as those with degree one. Further, an *internal node*, defined as a node with degree more than one, can be connected to many of the same types of leaf nodes, a characteristic we call *node multiplicity*. Some of the CI types like *process, ip_address, etc.*, have a wide range of multiplicities. CMDB-Miner employs leaf level summarization, which reduces the size of the CMDB graphs significantly, and aids interpretation of the mined infrastructure patterns.

For each leaf node \( u \) with CI type \( t \), we define its same label siblings as: \( \text{Sib}(u, t) = \{ x | L(x) = t, x \text{ is a leaf node, } x \text{ and } u \text{ have common neighbor} \} \). For every CI type \( t \), we define its multiplicities as: \( \text{Mult}(t) = \{ m | \exists u, u \text{ is a leaf, } L(u) = t, |\text{Sib}(u, t)| = m \} \). In other words, the multiplicities of CI type \( t \), is the multiset comprising the number of its occurrences at the leaf level with common internal neighbors. We discretize the multiplicities \( \text{Mult}(t) \) using equi-width binning. For each internal node connected to leaf nodes, we then attach a new label of the form \( x \rightarrow [l, u]y \), which is interpreted as internal node \( x \) having between \( l \) and \( u \) occurrences of CI type \( y \) as a leaf. Fig. 1.1 shows an example of such a label, namely, *issftpservice* \( \rightarrow [0, 2] \text{ issftpservice} \), meaning that *issftpservice* is connected to up to two other leaf nodes with CI type *issftpservice*.  

Table 3.1: IP Address related attributes and their values.

<table>
<thead>
<tr>
<th>DNS Server</th>
<th>NetMask</th>
<th>IP Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>192.168.1.1</td>
<td>255.255.255.0</td>
<td>192.168.1.2</td>
</tr>
<tr>
<td>192.168.1.1</td>
<td>255.255.0.0</td>
<td>192.168.1.3</td>
</tr>
<tr>
<td>192.168.1.1</td>
<td>255.255.0.0</td>
<td>192.168.1.4</td>
</tr>
<tr>
<td>192.168.1.1</td>
<td>255.255.255.0</td>
<td>192.168.1.5</td>
</tr>
<tr>
<td>192.168.1.1</td>
<td>255.255.0.0</td>
<td>192.168.1.6</td>
</tr>
</tbody>
</table>

Table 3.2: Attribute Entropies.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Entropy</th>
</tr>
</thead>
<tbody>
<tr>
<td>DNS Server</td>
<td>( \log_2(1) = 0.0 )</td>
</tr>
<tr>
<td>NetMask</td>
<td>( \frac{2}{5} \log_2 \left( \frac{2}{5} \right) + \frac{3}{5} \log_2 \left( \frac{3}{5} \right) = 0.971 )</td>
</tr>
<tr>
<td>IP Address</td>
<td>( \log_2 \left( \frac{1}{5} \right) = 2.322 )</td>
</tr>
</tbody>
</table>
3.4 Sampling maximal patterns

The goal of this step is to extract a sample of maximal frequent subgraphs from a large, sparse CMDB graph. We do this via random walks in the pattern space, starting from the empty graph, and extending the current candidate pattern by a random edge. After each extension we ensure that the pattern is frequent, according to our new network-flow based approach (described below). Thus random pattern extension and support computation form the two sub-steps for each candidate.

3.4.1 Random walks in pattern space

CMDB-Miner takes as input a parameter $k$, specifying the number of walks to perform. Each random walk begins with the empty graph, and extends the patterns via random edge extensions. If a random extension yields a frequent pattern, based on the flow-based support described below, it is accepted. Otherwise, the extension is rejected, and we try another random extension. If none of the possible extensions yield a frequent pattern, we are guaranteed that the current pattern is maximal, and we add it to the set of maximal patterns $M$. It is important to note that, unlike other graph mining approaches that check for isomorphism during pattern growth (to eliminate duplicates), CMDB-Miner does not check for isomorphism until all the $k$ walks finish. This way we pay the price for isomorphism only for patterns that are maximal, and not at each extension. This strategy confers significant efficiency.

Within a given walk, we assume that the edges (and nodes) in $P$ are numbered in the order in which they are added to generate $P$, starting from an empty graph. Edge ordering automatically leads to node ordering as well. For example, Fig. 3.1a shows a database graph $G$, and Fig. 3.1b shows a candidate pattern graph $P$. $e_1 = (p_0, p_1)$ is the first, $e_2 = (p_1, p_2)$ is the second, and $e_3 = (p_2, p_3)$ is the third edge to be added to $P$. All of these are examples of forward edges, i.e., an edge that introduces at least one new node to $P$. The nodes are ordered from $p_0$ to $p_3$. Due to node ordering, a forward edge is implicitly directed from a lower to a higher numbered node. The last edge to be added to complete $P$ is $e_4 = (p_3, p_1)$, and is an example of a backward edge, defined to be an edge between existing nodes. A backward edge is implicitly directed from a higher to lower numbered node. This
direction information is used in our flow-based support detailed next.

![Network Flow Diagram]

**Figure 3.1:** (a) A database graph. (b) A pattern graph. (c) All embeddings of \( P \). (d) and (e): edge and node disjoint embeddings of \( P \).

**Network-Flow Based Pattern Support:** Recall that a flow network \( G = (V, E) \) is a directed graph with two distinguished vertices – source \( s \) and sink \( t \). Every ordered edge \( (u, v) \in E \) has a capacity \( c(u, v) \geq 0 \). A flow in this network is a function \( f : E \to \mathbb{R} \) that satisfies the following properties: i) capacity constraint:
\[ f(u, v) \leq c(u, v), \text{ and ii) flow conservation: } \sum_{u \in V} f(u, v) - \sum_{v \in V} f(v, u) = 0, \text{ for all } v \in V \setminus \{s, t\}. \] The value of a flow is defined as \( |f| = \sum_{v \in V} f(s, v) \), and maximum flow is a flow with the maximum value. It is known that if all the edge capacities \( c(u, v) \) are integers, then there exists a maximum flow with only integer flows on the edges. A path from node \( u \) to \( v \) in a flow network \( G = (V, E) \) is a sequence of distinct vertices \( (v_1, v_2, \ldots, v_k) \) such that \( u = v_1, (v_i, v_{i+1}) \in E \) for all \( 1 \leq i \leq k-1 \), and \( v_k = v \). The length of this path is \( k - 1 \). A path from \( s \) to \( t \) is also called a \( s-t \) path.

We now describe the construction of a flow network in which the maximum flow corresponds to an upper bound on the edge disjoint support of the pattern. The main idea is that any embedding of a pattern can be viewed as a path from \( s \) to \( t \) in the flow network, and edge disjointness can be imposed by using unit capacities on the edges.

Consider a pattern \( P = (V', E', L') \), and a database graph \( G = (V, E, L) \). Let \( E' = \{e_1, e_2, \ldots, e_m\} \) be an ordering of the edges in \( P \) (e.g., the order in which pattern \( P \) was obtained). Recall that each edge is oriented, i.e., it is a forward or backward edge. Let \( \Pi_i \) denote the set of all embeddings in \( G \) for a single edge \( e_i \in E' \). For example, Fig. 3.2a shows the embeddings for each oriented edge in \( P \). The flow network \( F = (V_F, E_F) \) is constructed from the set of embeddings by setting \( V_F \) to be the set of distinct nodes over all the edge embeddings \( \Pi_i \), and by adding the directed edge \((a_j, b_j)\), with capacity \( c(a_j, b_j) = 1 \), for each embedding \( a_j, b_j \in \Pi_i \), with \( 1 \leq j \leq |\Pi_i| \) and \( 1 \leq i \leq m \). Further, we add an edge \((s, u)\) for each distinct \( u \) such that \((u, v) \in \Pi_1\), with capacity \( c(s, u) = n_u \), where \( n_u \) is the number of time node \( u \) appears in \( \Pi_1 \). Finally, we add an edge \((v, t)\) for each distinct \( v \) such that \((u, v) \in \Pi_m\), with capacity \( c(v, t) = n_v \), where \( n_v \) is the number of times \( v \) appears in \( \Pi_m \). Fig. 3.2b shows the flow network obtained from the embeddings of each edge in \( P \). For instance, since \( 0, 1 \in \Pi_1 \), we add the edge \((0, 1)\) in \( F \) with capacity \( 1 \). Likewise, since \( 2, 1 \in \Pi_4 \) we add the edge \((2, 1)\) in \( F \) with capacity \( 1 \). The same is done for all embeddings in \( \Pi_j \), for \( 1 \leq j \leq 4 \). There are three distinct start nodes in \( \Pi_1 \), namely \( \{0, 5, 8\} \), thus we add three edges from the source: \((s, 0)\) with capacity \( 2 \), \((s, 5)\) with capacity \( 1 \), and \((s, 8)\) with capacity \( 1 \). Finally, there
are two distinct end nodes in $\Pi_4$, namely $\{1,9\}$, thus we add two edges to the sink: $(1,t)$ with capacity 3, and $(9,t)$ with capacity 1.

<table>
<thead>
<tr>
<th></th>
<th>$A-B$</th>
<th>$B-C$</th>
<th>$C-D$</th>
<th>$D-B$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$e_1(p_0,p_1)$</td>
<td>$e_2(p_1,p_2)$</td>
<td>$e_3(p_2,p_3)$</td>
<td>$e_4(p_3,p_1)$</td>
<td>$\Pi_1$</td>
</tr>
<tr>
<td>0,1</td>
<td>1,3</td>
<td>3,2</td>
<td>2,1</td>
<td></td>
</tr>
<tr>
<td>0,9</td>
<td>1,7</td>
<td>3,4</td>
<td>4,1</td>
<td></td>
</tr>
<tr>
<td>5,1</td>
<td>7,6</td>
<td>7,6</td>
<td>6,1</td>
<td></td>
</tr>
<tr>
<td>8,9</td>
<td>9,10</td>
<td>7,11</td>
<td>11,9</td>
<td></td>
</tr>
</tbody>
</table>

(a) Edge Embeddings

(b) Flow Network for $P$

(c) Disjoint Edge Mappings

Figure 3.2: Flow Network and Maximum Flow: (a) Edge embeddings for $P$. (B) Flow network for $P$. Boxes show capacity and flow on each edge. Maximum flow has value 3. (c) A possible set of three edge disjoint embeddings corresponding to the maximum flow of 3.

**Definition 1.** The flow-based support of a pattern $P$, denoted $\text{sup}_f(P)$, is defined as the maximum flow in the flow network for $P$.

There are several efficient implementations of maximum flow. We use Dinic’s
algorithm [60] which is based on blocking flows. In special cases where all the edges have a unit capacity it has complexity $O(\min(V^{2/3}, E^{1/2}) \cdot E)$. Fig. 3.2b shows that the maximum flow value is 3, thus $\sup_f(P) = 3$. Fig. 3.2c shows the three disjoint edge mappings for $P$, corresponding to the three disjoint embeddings in Fig. 3.1d.

We now prove that the flow-based support of $P$ is an upper-bound on the edge disjoint support. Let $G = (V, E, L)$, and $P = (V', E', L')$ with $|V'| = n$ and $|E'| = m$. We make the following observations:

- **Lemma 1**: If $\pi$ is an embedding of $P$ in $G$, then there exists a corresponding $s$-$t$ path in the flow network $F$. This follows immediately from the facts that: i) $P$ is connected, ii) for each edge $e_i = (a_i, b_i) \in E'$, there is an edge in the flow network corresponding to the edge mapping for $e_i$, namely $\pi(e_i) = (\pi(a_i), \pi(b_i)) \in \Pi_i$, iii) there exists an edge from $s$ to each start node in $\Pi_1$ (for edge $e_1 \in P$), and from each end node in $\Pi_m$ (for $e_m \in P$) to $t$. Note that the path length can be less than $m + 2$. It is $m + 2$ when all edges in $P$ lie on some path from $s$ to $t$.

- **Lemma 2**: If $\pi_1$, and $\pi_2$ are two edge disjoint embeddings of $P$ in $G$, then the $s$-$t$ corresponding paths are disjoint, ignoring the out-edges of $s$ and in-edges of $t$ (which may be shared).

- **Lemma 3**: If $\Pi = \{\pi_1, \pi_2, \ldots, \pi_k\}$ is a set of edge-disjoint embeddings of $P$ in $G$, then the maximum flow is at least $k$. Let $n_u$ embeddings have the same start vertex $u$, and let $n_v$ embeddings have the same end vertex $v$. From Lemma 2, we know that ignoring $s$ and $t$, there are $k$ disjoint paths in the flow network $k$, corresponding to each of the $k$ embeddings in $\Pi$. If $f(e) = 1$ for all edges $e$ on these paths, and if $f(s, u) = n_u$ and $f(v, t) = n_v$, then we can see that the resulting flow has value at least $k$.

**Theorem 1.** The maximum flow in the flow network $F$ for pattern $P$ is an upper bound on the number of edge disjoint embeddings of $P$ in $G$.

**Proof.** In Lemma 3, if $\Pi$ is the set of all possible edge-disjoint embeddings of $P$, with $|\Pi| = k$, then the maximum flow in $F$ is at least $k$, and $\sup_f(P) \geq k = \sup_e(P)$. Let
$Q$ be an extension of pattern $P$, i.e., $P \subseteq Q$. Since every edge disjoint embedding of $Q$ is also an edge disjoint embedding of $P$ it immediately implies that $sup_e(Q) \leq sup_e(P) \leq sup_f(P)$.

The fact that $sup_f(P)$ is an upper-bound on the edge disjoint support allows us to prune any extension (an immediate supergraph) of pattern $P$ if $sup_f(P) < \text{minsup}$. This follow immediately from the theorem above, since $sup_f(P) < \text{minsup} \implies sup_e(Q) < \text{minsup}$, and thus we can guarantee that no extension of $P$ can be frequent according to edge disjoint support. We will call the flow network constructed using the embeddings of all the edges in the network as the Complete Flow (CF) network.

It is worth noting that the edge-disjoint support of $P$ is equal to the maximum number of edge disjoint $s-t$ paths of length $m+2$ in the flow network. However, [61] proved that finding the maximum disjoint paths with constraints on the length is NP-Complete. For this reason our formulation does not place any restrictions on the length of the paths, and thus we obtain an upper-bound on the edge-disjoint support. It is important to note that Dinic’s algorithm finds the shortest $s-t$ paths that are saturated. Thus the flow-based support is close to the actual support if the shortest $s-t$ path length in the flow network is close to the number of edges in the candidate pattern. While it may not be a beneficial strategy for general (dense) patterns, our formulation is very effective for CMDB graphs, which are sparse, and thus the mined patterns are also sparse. For such patterns the flow-based support is generally close to the edge disjoint support.

### 3.4.1.1 Optimization

The upper bound on the edge disjoint support calculated using the flow network formulation of a candidate pattern may be loose if the length of the shortest path between $s$ and $t$ is smaller compared to the number of edges in the candidate pattern. This situation arises when the candidate pattern is generated as a result of adding a back edge to a vertex close to $p_0$ or by branching from a vertex close to $p_0$. In the extreme case when the back edge is connected to $p_0$ or the branch extends from $p_0$ any flow that originates at $s$ can reach $t$ without any restriction, i.e., the
capacity constraint on the internal edges may have no effect.

Consider the sample database shown in 3.3a and suppose that \( e_1(p_0, p_1) \) and \( e_2(p_1, p_2) \) are the two edges that are added to an empty pattern leading to a frequent pattern \( P' \). In the next step of the random walk, a back edge is added from \( p_2 \) to \( p_0 \) leading to the candidate pattern \( P \) shown in 3.3b. Figure 3.3c shows all the embeddings for the candidate pattern \( P \). It can be seen that the maximum number of edge disjoint embeddings for \( P \) is 1 as the edge mapping of \( e_2 \) is \((2,4)\) and it is in both \( \pi_0 \) and \( \pi_1 \). Figure 3.4 shows the edge mappings and the resulting flow network for \( P \). The maximum \( s-t \) flow of 2 in the network is obtained by pushing a unit flow on the paths \( s-g_0-t \) and \( s-g_1-t \). Notice that there are no internal edges on these paths. In this example, if the flow network is constructed using only edge mappings of \( e'_3(p_0, p_2) \) and \( e'_2(p_2, p_1) \) then the maximum flow is 1. Figure 3.5 shows this alternative. This construction separates the \( s' \) and \( t' \) nodes by at least two edges and hence the capacity constraint on the edges restricts the maximum flow to 1.

### 3.4.2 Longest path network

The key take away from the above example is that flow can be restricted by separating \( s \) and \( t \) in the network. Therefore, to obtain a better upper bound on the edge disjoint support we compute the maximum flow in a network where \( s \) and \( t \) are most separated, but the support upper bound is still guaranteed. We refer to this optimization as the Longest Path (LP) network, and contrast it with the complete flow network given above.

Let \( P' = (V', E') \) be a frequent pattern and \( P = (V, E) \) be the pattern obtained by adding the edge \( e_k = (v_i, v_j) \). Note that \( v_j \notin V' \) if \( e_k \) is a forward edge. To construct the Longest Path network of \( P \), we first find a vertex \( v_s \in V' \) which is farthest from \( v_j \), with \( S \) denoting the longest path, which starts at \( v_j \) and ends at \( v_s \). We place an additional constraint that the first edge on \( S \) is \((v_j, v_i)\). The LP network then is the same as the flow network of a pattern \( Q = (V'', E'') \subseteq P \) where \( E'' \) is the set of all edges present on the path \( S \). The source node \( s \) is connected to the edge mappings of the first edge on \( S \), and the mappings of the last edge on
Figure 3.3: (a) A database graph. (b) A candidate pattern obtained by adding a back edge between \( p_2 \) and \( p_0 \). (c) All embeddings of \( P \). The embeddings \( \pi_0 \) and \( \pi_1 \) of \( P \) in \( G \) are not edge disjoint.

<table>
<thead>
<tr>
<th></th>
<th>( A-B )</th>
<th>( B-C )</th>
<th>( C-A )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( e_1(p_0,p_1) )</td>
<td>( \Pi_1 )</td>
<td>( \Pi_2 )</td>
<td>( \Pi_3 )</td>
</tr>
<tr>
<td>( e_2(p_1,p_2) )</td>
<td>0, 2</td>
<td>2, 3</td>
<td>4, 0</td>
</tr>
<tr>
<td>( e_3(p_2,p_0) )</td>
<td>1, 2</td>
<td>2, 4</td>
<td>4, 1</td>
</tr>
</tbody>
</table>

(a) Edge Embeddings

Figure 3.4: Flow Network and Maximum Flow: (a) Edge embeddings for \( P \). (b) Flow network for \( P \). Boxes show capacity and flow on each edge. Maximum flow has value 2.

\( S \) are connected to the sink \( t \) in the LP network. As before, the maximum flow in the LP network is an upper bound on the edge disjoint support. There is a two fold advantage to using the maximum flow in the LP network: 1) \( Q \) doesn’t have
Figure 3.5: Flow Network and Maximum Flow: (a) Edge embeddings for the edges on the longest path. (b) Flow network for edges on the path. Boxes show capacity and flow on each edge. Maximum flow has value 1. Note that $g_3$ has been omitted from the network as it not connected to any embedding in $\Pi_1$.

a cycle and is branch free, which implies that the $s-t$ flow is more constrained.

2) The size of the LP network is smaller compared to the size of flow network for $P$. This reduces the time required to compute the maximum flow. We will now prove the upper bound property of the LP network. The argument for the proof is similar to theorem 1. We continue to use the notation introduced above and make the following observations about the LP network.

- **Lemma 4**: If $\pi$ is an embedding of $P$ in $G$, then there exists a corresponding $s$-$t$ path in the LP network. This follows immediately from the following facts: i) $Q$ is connected, ii) for each edge $e_i = (a_i, b_i) \in E''$, there is an edge in the LP network corresponding to the edge mapping for $e_i$, namely $\pi(e_i) = (\pi(a_i), \pi(b_i)) \in \Pi_i$, iii) there exists an edge from $s$ to each start node in the edge mappings of $(v_j, v_i)$, and to $t$ from each end node in the mappings of the last edge on the path $S$.

Given the existence of $s-t$ path, the rest of the proof is similar to the proof of
3.4.3 Pruning isomorphic patterns

Given a minimum support threshold $\text{minsup}$, and given $k$, the number of random walks, CMDB-Miner performs $k$ random walks in the pattern space, to yield a set $M$ of exactly $k$ maximal frequent subgraphs, using flow-based support. However, since the walks are random, they may yield isomorphic maximal patterns. Such isomorphic patterns have to be discarded before the infrastructure pattern extraction step. Unfortunately, while graph isomorphism is in NP, it is not known whether it is NP-complete or is in P [62].

Instead of checking for isomorphism between every pair of maximal patterns in $M$, we use a sequence of polynomial-time filters to create equivalence classes of possibly isomorphic patterns. Thus, the worst-case exponential time algorithm for graph isomorphism has to be applied to only pairs of graphs within the same equivalence class. Initially $M$ comprises a single equivalence class. We then apply the following filters:

- **Node Multiset**: Given a pattern $P = (V', E', L')$, define $\rho_V(P) = \{L(v_i) : v_i \in V\}$ to be the *multiset* of node labels in $P$. It is easy to see that two patterns $P$ and $P'$ cannot be isomorphic if $\rho_V(P) \neq \rho_V(P')$. In this case $P$ and $P'$ are added to different equivalence classes, and never have to be checked for isomorphism.

- **Edge Multiset**: Given pattern $P = (V', E', L')$, for each edge $e_i = (a_i, b_i) \in E'$, define a *composite edge label* to be the triple $\mathcal{L}(e_i) = (L'(a_i), L'(b_i), L'(e_i))$, with $a_i < b_i$. Define the filter $\rho_E(P) = \{\mathcal{L}(e_i) : e_i \in E'\}$ to be the *multiset* of composite edge labels for $P$. Two patterns $P$ and $P'$ cannot be isomorphic if $\rho_E(P) \neq \rho_E(P')$.

- **Laplacian Spectrum**: Let $A$ be the adjacency matrix for pattern $P$, i.e., $A(v_i, v_j) = 1$ if $(v_i, v_j) \in E'$, and $A(v_i, v_j) = 0$, otherwise. Let $D$ be the diagonal degree matrix for $P$, defined as $D(v_i, v_i) = \sum_{v_j} A(v_i, v_j)$, and $D(v_i, v_j) = 0$ for all $v_i \neq v_j$. Define the *normalized Laplacian matrix* of $P$ as follows:

$$
\mathcal{L}(e_i) = (L'(a_i), L'(b_i), L'(e_i))
$$
Table 3.3: CMDB Graphs A,B: before and after preprocessing

<table>
<thead>
<tr>
<th>Property</th>
<th>A</th>
<th>B</th>
<th>Before</th>
<th>After</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>V</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>443192</td>
<td>11363</td>
<td>455012</td>
<td>57525</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>E</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>480143</td>
<td>20978</td>
<td>523415</td>
<td>149229</td>
</tr>
<tr>
<td>Avg. Deg.</td>
<td></td>
<td></td>
<td>2.16</td>
<td>3.68</td>
<td>2.3</td>
<td>5.16</td>
</tr>
</tbody>
</table>

Table 3.4: Biggest pattern extracted

<table>
<thead>
<tr>
<th>Database</th>
<th># Vertices</th>
<th># Edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>24</td>
<td>41</td>
</tr>
<tr>
<td>B</td>
<td>54</td>
<td>55</td>
</tr>
</tbody>
</table>

\[ N = D^{-1/2} \cdot (D - A) \cdot D^{1/2}. \]

\(N\) is a \(n \times n\) positive semi-definite matrix, and thus \(N\) has \(n\) (not necessarily distinct) real, positive eigen-values: \(\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_n \geq 0\). Define the Laplacian spectrum of \(P\) as the multiset \(\rho_S(P) = \{\lambda_i : 1 \leq i \leq n\}\). It is known that two isomorphic patterns are iso-spectral, i.e., they have the same Laplacian spectrum [62]. Thus, \(P\) and \(P'\) cannot be isomorphic if \(\rho_S(P) \neq \rho_S(P')\)

After applying the above filters, the set \(M\) is partitioned into smaller equivalence classes of possibly isomorphic graphs. For each pair of graphs in the same class, we perform full isomorphism checking using the VF2 [63] algorithm. The output of this step is the final set \(M\) of non-isomorphic maximal frequent patterns in \(G\). Note that at this stage we can find the actual edge disjoint support of all the maximal patterns by using the maximal independent set approach proposed in [15].

### 3.5 Sampling results

In this section we evaluate CMDB-Miner on real-world CMDB graphs for two multi-national corporations, company A and B (names not revealed due to non-disclosure issues), from HP’s Universal Configuration Management Database (UCMDB). We also conduct experiments to validate some of the design choices in the implementation of CMDB-Miner. All experiments were performed on a machine with 2.67GHz Intel i7 processor with 4GB of memory running Ubuntu Linux version 10.04.
3.5.1 Preprocessing

The raw CMDB graph of company A contains 443,192 vertices and 480,143 edges. Company B also contains a similar number of nodes and edges, and is shown in Table 3.3. We discard uninformative attributes for each composite item, by discarding both high and low entropy attributes. This results in a significant reduction in the number of attributes, as shown in Figure 3.6 for some of the common CI types in the CMDB graph of company A (similar results are obtained for B too). More than 75% of the attributes are pruned in this stage. Further, collapsing leaf nodes reduces the total number of vertices to 11,363. Table 3.3 shows the graph order and size, as well as average degree, both before and after preprocessing. As pointed out earlier, these two preprocessing steps also aid in better interpretation of the final infrastructure patterns.
Figure 3.7: Maximal and Non-Isomorphic Patterns for Company A: (a) sampling time and (b) number of distinct maximal patterns, versus number of random walks.

3.5.2 Sampling maximal patterns

Fig. 3.7a shows the time for sampling maximal patterns versus number of random walks, for two different (absolute) values of minimum support for company A using the full flow algorithm. We can see that as expected time is linear in the number of walks. Fig. 3.7b shows the number of distinct or non-isomorphic maximal patterns versus the number of random walks. We can see that for \( \text{minsup} = 125 \) the fraction of distinct maximal patterns decreases with the number of walks, indicating convergence to the “true” set of maximal patterns. The convergence has not yet been reached for \( \text{minsup} = 110 \) within 2500 walks. These curves suggest an automated method to stop sampling, namely, when the ratio of the number of distinct patterns to the number of walks falls below some threshold. Fig. 3.8a shows the time for
random walks using the LP network. It can be seen that the LP formulation greatly reduces the running time for sampling maximal patterns. Similar results were also obtained for company B. We show the size of the maximal pattern extracted in Table 3.4.

Table 3.5 shows the time to detect the number of distinct maximal patterns. We compare the time taken by our filter based approach versus the cost of running the VF2 algorithm on each pair of patterns in $M$. It is clear that the sequence of filters is very effective in reducing the running time by over an order of magnitude.

3.5.2.1 Baseline Algorithm (BA)

To compare the effectiveness of our random sampling and network flow formulations, we compared them with a baseline approach we implemented. Most of the existing single graph mining algorithms explore the entire pattern space and require an enumeration of all the embeddings to compute the support of a candidate pattern. Instead of comparing with a specific algorithm we formulated a baseline approach that retains a common attribute present in all the existing algorithms, namely, embedding enumeration. The support of a pattern is computed using the maximum flow in the LP network constructed using all the embeddings, and the candidates are generated using random sampling as before.

Unfortunately, on the real datasets from Company A and B, the baseline algorithm could not be run (it crashed after some time) because of the exponential number of embeddings of the candidates, especially given the multiplicity and other characteristics of the CMDB graphs mentioned earlier. CMDB-Miner is essentially the only viable approach for these types of graphs. Nevertheless, to get a better understanding of the baseline methods and CMDB-Miner, we also compared them on synthetic graphs, as described below.
3.5.2.2 Synthetic graph data

Synthetic graph datasets were generated using the graph generator provided by Kuramochi et al. [15]. A simple graph can be generated by using the following parameters: Number of transactions (N), Number of Edge Labels (L), number of vertex labels (V), and other parameters which control the size and number of frequent patterns in the graph. Figure 3.9a shows the time versus number of random walks in the pattern space for four different minimum support values in a graph containing 3000 edges, 1817 nodes, with 200 node labels, and constructed using frequent patterns of size 10. All the edges in the graph have the same label. The figure shows that the time is linear in the number of walks for the different support values. However, note that that due to the random pattern extensions, there is not always a clear relationship between the time for a lower and higher minimum support. In general, it is the case that a lower support values takes more time. However, we can see that extracting 2000 patterns using support of 2 takes slightly less time than that for support 4 for these sparse synthetic graphs. This might be an artifact of the higher probability of finding frequent extensions with lower support values.

The plot in 3.9b shows the number of non-isomorphic maximal patterns for support value of 10, as a function of the number of random walks. Figure 3.9c compares the run times for a given number of random walks using the complete flow network and the longest path optimization. The minimum support used for this experiment is 4 and the graph is generated using the parameters: 7000 edges, 3748 nodes, with 60 distinct vertex labels, 1400 frequent graphs of average size 10. By using the longest path optimization the run time reduces significantly. This is due to the fact that flow in the longest path network is a tighter upper bound, and in general contains fewer nodes and edges. Figure 3.9d shows the time for enumerating 1000 maximal patterns versus the number of nodes/vertices in the graph dataset. Here we also compare with the baseline method. When the dataset size is small the LP approach is marginally better compared to the baseline or the complete flow approach. However, as the dataset size increases the run time for LP becomes significantly less compared to other two algorithms. The figure also shows that the run time for BA increases steeply as the dataset size increases where the increase
for LP is much slower. This also shows the scalability of the LP algorithm. Finally, Figure 3.9e shows that the run time in general decreases as the minimum support is increased. Here again, the longest path network yields the best results.
Figure 3.9: (a) shows times for random walks with different minimum support. (b) shows the number of non-isomorphic maximal patterns. (c) compares the run time using Longest Path and Complete Flow networks. (d) shows the effect of database size (number of nodes) on the run time. (e) shows the effect of minimum support on run time. All the times mentioned are in secs.
CHAPTER 4
Mining Approximate Patterns with Label Mismatches

We propose a novel approach to effectively prune the space of approximate labeled isomorphisms. Instead of enumerating all the possible isomorphisms, we maintain a set of representatives (mappings of the pattern vertex in the database) that is linear in the database and pattern size. Pruning is applied on this set to narrow down the search to only viable mappings. We propose label based iterative pruning methods to compute the representative sets efficiently. These methods are based on $k$-hop labels and neighbor concatenated labels.

4.1 Preliminaries

For convenience, we repeat the definitions of graphs and isomorphisms. Our focus in this chapter is on mining subgraph patterns with exact structure and bounded label mismatch isomorphisms. Therefore the isomorphism $\phi$ refers to the approximate isomorphism $\phi_{\alpha,0}$ as defined in the section 1.4.2.

An undirected labeled graph $G$ is represented as a tuple $G = (V_G, E_G, L)$ where $V_G$ is the set of vertices, $E_G$ is the set of edges and $L: V_G \rightarrow \Sigma$ is a function that maps vertices to their labels. The neighbors of a vertex $v$ are given as $N(v) = \{u|(u,v) \in E_G\}$. A walk in a graph is a sequence of vertices $v_0, \ldots, v_k$ such that there is an edge between adjacent pairs of vertices i.e., $(v_i, v_{i+1}) \in E_G$ and its length is $k$. A walk is a path if every vertex appears at most once in the sequence i.e., $v_i \neq v_j$, for $i \neq j$. We write $u \xrightarrow{k} v$ if there is a path of length $k$ between $u$ and $v$.

Cost matrix: We assume that there is a cost matrix $C: \Sigma^2 \rightarrow \mathbb{R}^{\geq 0}$. The entry $C[l_i][l_j]$ denotes the cost of matching the labels $l_i$ and $l_j$. Typically $C$ is usually

* Portions of this chapter previously appeared as
Approximate subgraph isomorphism: A graph \( S = (V_S, E_S, L) \) is a subgraph of \( G \), denoted \( S \subseteq G \), iff \( V_S \subseteq V_G \), and \( E_S \subseteq E_G \). Given a database graph \( G \) and a pattern graph \( P = (V_P, E_P, L) \), a function \( \phi: V_P \rightarrow V_G \) is called an unlabeled subgraph isomorphism provided \( \phi \) is an injective (or one-to-one) mapping such that \( \forall (u, v) \in E_P \), we have \( (\phi(u), \phi(v)) \in E_G \). That is, \( \phi \) preserves the topology of \( P \) in \( G \). Define the cost of the isomorphism as follows: \( C(\phi) = \sum_{u \in V_P} C[L(u)][L(\phi(u))] \), that is, the sum of the costs of matching the node labels in \( P \) to the corresponding node labels in \( G \). We say that \( \phi \) is an approximate subgraph isomorphism from \( P \) to \( G \) provided its cost \( C(\phi) \leq \alpha \), where \( \alpha \) is a user-specified threshold on the total cost. In this case we also call \( P \) an approximate pattern in \( G \). Note that if \( \alpha = 0 \), then \( \phi \) is an exact subgraph isomorphism between \( P \) and \( G \). From now on, isomorphism refers to approximate subgraph isomorphism unless specified otherwise.

Pattern support: Given a (large) database graph \( G \), a pattern graph \( P \), and the set of all approximate subgraph isomorphisms \( \Phi \) from \( P \) to \( G \), the support of \( P \) is some anti-monotonic function \( sup(P, \Phi) \), i.e., \( sup(P, \Phi_P) \leq sup(Q, \Phi_Q) \) for all subgraphs \( Q \) of \( P \). We discuss some of the common support definitions in Sec 4.3.2. A pattern \( P \) is called frequent if \( sup(P) \geq \text{minsup} \), where \( \text{minsup} \) is a user defined support threshold. \( P \) is maximal iff \( P \) is frequent and there does not exist any
supergraph of $P$ that is frequent in $G$.

**Representative set**: Given a node $u \in V_P$, its *representative set* in the database graph $G$ is the set

$$R(u) = \{v \in V_G | \exists \phi, \text{ such that } C(\phi) \leq \alpha \text{ and } \phi(u) = v\}$$

That is, the representative set of $u$ comprises all nodes $v$ in $G$ that $u$ is mapped to in some isomorphism $\phi$. Figure 4.1 shows an example database, a cost matrix, an approximate pattern, and its approximate subgraph isomorphisms for $\alpha = 0.5$. There are only two possible approximate isomorphisms from $P$ to $G$, as specified by $\phi_1$ and $\phi_2$. For example, for $\phi_1$, we have $\phi_1(1) \rightarrow 30$, $\phi_1(2) \rightarrow 10$, $\phi_1(3) \rightarrow 60$, and $\phi_1(4) \rightarrow 40$, as seen in Table 4.1d. The cost of the isomorphism is $C(\phi_1) = 0.4$, since $C(L(1), L(30)) + C(L(2), L(10)) + C(L(3), L(60)) + C(L(4), L(40)) = C(A, B) + C(B, A) + C(C, C) + C(A, A) = 0.2 + 0.2 + 0 + 0 = 0.4$. The representative set for node $1 \in V_P$ is $R(1) = \{30, 40\}$.

**Outline of our Approach**: The two main steps in approximate graph mining are candidate generation and support computation. With candidate generation the search space of the frequent patterns is explored. For each candidate that is generated, we can check whether it is frequent by computing its support. Given a pattern with $k$ vertices, the maximum number of possible isomorphisms are $k! \times \binom{|V_G|}{k}$. It is therefore infeasible to either enumerate or store the complete set of isomorphisms. Computing and storing the representative sets is a compromise that will enable us to decide efficiently if a candidate is frequent.

### 4.2 Computing representative sets

Representative vertex $v$ of a pattern vertex $u$ implies that there exists an isomorphism $\phi$ for which $\phi(u) = v$. One way to interpret it is that the neighborhood of $u$ matches with that of $v$. By comparing the neighborhoods we can find vertices that are not valid representatives of $u$ without trying to find an isomorphism exhaustively. Therefore, to compute the representative sets we will start with a candidate representative set denoted by $R'(u)$ and iteratively prune some
of the vertices if the neighborhoods cannot be matched. The candidate set is a super set of the representative set, $R'(u) \supseteq R(u)$. An example of candidate set, $R'(u) = \{v | v \in V_G, C[L(u)][L(v)] \leq \alpha \}$ i.e., the isomorphisms of the single vertex pattern with label $L(u)$. In this section, we will describe different notions of neighborhood and show how they help us in computing the representative sets of vertices in a pattern.

The problem of checking whether a vertex $v \in R(u)$ involves solving isomorphism is at least as hard finding unlabeled subgraph isomorphism. Therefore, checking if a vertex is a valid representative is an NP Hard problem. The pruning methods typically do not prune all the invalid vertices. So, we use an exhaustive enumeration method to prune these invalid vertices and reduce $R'(u)$ to $R(u)$.

### 4.2.1 k-hop label

k-hop label is defined as the set of vertices that are reachable via a simple path of length $k$. In other words, k-hop label contains all vertices that are reachable in k-hops starting from $u$ and by visiting each vertex at most once. Note that, we use the word label even though we refer to a set of vertices. Formally, the k-hop label of a vertex $u$ in graph $G$, $h_k(u, G) = \{v | v \in G, u \xrightarrow{k} v \}$. We simply write it as $h_k(u)$ when the graph is evident from the context. For example, for pattern $P$ in Fig. 4.2a, the 0-hop label of vertex 5 is $h_0(5) = \{5\}$, its 1-hop label is the multiset $h_1(5) = \{2, 4, 6\}$ (we omit the set notation for convenience) and its 2-hop label $h_2(5) = \{1, 3\}$. The minimum cost of matching k-hop labels $h_k(u)$ and $h_k(v)$ is

$$C[h_k(u)][h_k(v)] = \min \sum_{u' \in h_k(u)} C[L(u')][L(f(u'))]$$  \hspace{1cm} (4.1)

where the minimization is over all injective functions $f: h_k(u) \rightarrow h_k(v)$ and $C[L(u')][L(f(u'))]$ is the cost of matching the vertex labels. In other words, it is the minimum total cost of matching the vertices present in the k-hop labels. The following theorem places an upper bound on the minimum cost of matching the k-hop labels of a pattern vertex and any of its representative vertices.

**Theorem 2.** Given any pattern vertex $u$, a representative vertex $v \in R(u)$ and cost
threshold $\alpha$, the minimum cost of matching the $k$-hop labels, $C[h_k(u)][h_k(v)] \leq \alpha$ for all $k \geq 0$.

**Proof 1.** Consider any isomorphism $\phi$ such that $\phi(u) = v$. It is enough if we can show an injective function $f : h_k(u) \rightarrow h_k(v)$ with a cost (as defined in equation 4.1) $\leq \alpha$. We will argue that the function $\phi$ on the restricted domain $h_k(u)$ is one such function $f$. First, we know that $\sum C[L(u)][\phi(L(u))] \leq \alpha$, $u \in V_P$, since $\phi$ is an isomorphism. Second, let $u \xrightarrow{k} u'$ then $\phi(u') \in h_k(v)$ because for every edge $(u_1, u_2)$ on a path between $u$ and $u'$ in $P$, $(\phi(u_1), \phi(u_2)) \in E_G$. Therefore the cost of matching the $k$-hop labels using $\phi$ is upper bounded by $\alpha$. Hence, the minimum cost of matching $C[h_u(v)][h_u(\leq)]\alpha$.

Based on the above theorem, a vertex $v$ is not a representative vertex of $u$ if $C[h_k(u)][h_k(v)] > \alpha$ for any $k \geq 0$. However, in practice, it enough to check the condition only for $k \leq |V_P| - 1$ because $h_k(u)$ is the null set $\forall k \geq |V_P|$ and the condition is trivially satisfied.

Figure 4.2 shows an example for the $k$-hop label based pruning of the candidate representative set where the threshold $\alpha = 0.5$. Consider vertex $2 \in V_P$ and vertex $20 \in V_G$, we have, $C[h_0(2)][h_0(20)] = 0$, since the cost of matching vertex labels $C[L(2)][L(20)] = 0$, as per the label matching matrix $C$ in Fig. 4.2c. The $k$-hop labels for $k = 1, 2, 3$ and the minimum of cost matching them are as shown in the table 4.1, and it can be verified that the minimum cost is within the threshold $\alpha$.

Thus far, we cannot prune node 20 from $R(2)$. However, $h_4(2) = 4, 6$ and $h_4(20) = 30, 60$ and the minimum cost of matching them is $0.6 > \alpha$. Thus, from theorem 2 we conclude that $20 \notin R(2)$. This example illustrates that $k$-hop labels can help prune the candidate representative sets.

<table>
<thead>
<tr>
<th>$k$</th>
<th>$h_k(2)$</th>
<th>$h_k(20)$</th>
<th>$C[h_k(2)][h_k(20)]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1, 3, 5</td>
<td>10, 30, 50, 60</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>4, 6</td>
<td>40, 50, 60</td>
<td>0.4</td>
</tr>
<tr>
<td>3</td>
<td>3, 5</td>
<td>40, 30, 50</td>
<td>0.1</td>
</tr>
</tbody>
</table>
4.2.2 Neighbor concatenated label

In Neighbor concatenated label (NL), the information regarding the candidates of a neighbor that were pruned in the previous iteration is used along with the current k-hop label to prune candidates in the current iteration. In contrast, the k-hop label pruning strategy for a vertex $u$ works independently of the result of k-hop label pruning of other vertices in the pattern. This leads us to the following recursive formulation for NL.

The NL of a vertex in the $k + 1^{th}$ iteration, $\eta_{k+1}(u)$, is defined as the tuple $(\{\eta_k(u')|u' \in N(u)\}, h_{k+1}(u))$. The first element (A) of the tuple is the set of NL of the neighbors of the vertex $u$ in the previous iteration ($k$) and the second element (B) is exactly same as the $(k+1)$-hop label defined in the section 4.2.1. We say that $\eta_{k+1}(u)$ dominated by $\eta_{k+1}(v)$, denoted by $\eta_{k+1}(u) = (A, B) \preceq \eta_{k+1}(v) = (A', B')$, if $C[h_{k+1}(B)][h_{k+1}(B')] \leq \alpha$ i.e., the minimum cost of matching the $(k+1)$-hop labels
is within $\alpha$ ii) there exits an injective function $g: A \to A'$ such that $a \preceq g(a)$ for all $a \in A$ i.e., there is a one to one mapping between the NL labels (of the previous iteration, $k$) of neighbors of $u$ and $v$. The base case $\eta_0(u) \preceq \eta_0(v)$ iff $C[L(u)][L(v)] \leq \alpha$. For example, in Fig 4.2 $\eta_1(2) \preceq \eta_1(20)$ because $C[h_1(2)][h_1(20)] \leq \alpha$ and the NL labels of vertices 1, 3, 5 are dominated by the NL labels of vertices 10, 50, 30 respectively. The following theorem states that the NL of a pattern vertex $u$ is dominated by the NL of any of its representative vertex $v \in R(u)$.

**Theorem 3.** Given any pattern vertex $u$, a representative vertex $v \in R(u)$ and cost threshold $\alpha$, $\eta_k(u) \preceq \eta_k(v)$ for all $k \geq 0$.

**Proof 2.** Let $\phi$ be any isomorphism such that $\phi(u) = v$. We prove the theorem by using induction on $k$.

**Base case:** $\eta_0(u) \preceq \eta_0(v) \iff C[L(u)][L(v)] \leq \alpha$ is true because $v \in R(u)$.

**Inductive Hypothesis:** Assume that $\eta_k(u) \preceq \eta_k(v)$ holds true for all $u \in P$ and $v \in R(u)$.

Now consider $\eta_{k+1}(u) = (A, B)$ and $\eta_{k+1}(v) = (A', B')$, from theorem 2 we know that $C[B][B'] \leq \alpha$, for all $k \geq 0$. Let $u' \in N(u)$ and $v' = \phi(u')$. From inductive hypothesis, $\eta_k(u') \preceq \eta_k(v')$. Also, $v' \in N(v)$ because $(u, u') \in E_P \implies (\phi(u) = v, \phi(u') = v') \in E_G$. Therefore, the injective function $\phi$ maps the elements $a \in A$ to $\phi(a) \in A'$. The theorem follows from the definition of the NL label.

Based on the above theorem, a vertex $v$ can be pruned from $R'(u)$ if $\eta_k(u) \not\preceq \eta_k(v)$ for some $k \geq 0$. In Fig 4.2, consider the vertices $3 \in P$, 50 $\in G$ and let $\alpha = 0.5$. The NL labels, $\eta_0(3) \preceq \eta_0(50)$ as $C[B][B] = 0 \leq \alpha$. Similarly it is also true for the pairs (2, 20), (4, 40) etc. It follows that $\eta_1(3) \preceq \eta_1(50)$ as the neighbors 2, 4, 6 can be mapped to 20, 40, 60 respectively and the minimum cost of the matching the 1-hop label is 0.4 which is less than the $\alpha$ threshold. But $\eta_2(3) \not\preceq \eta_2(50)$ because the NL label $\eta_1(6)$ is not dominated by the NL label of 20, 40 or 60 in the first iteration. So, there is no mapping between the neighbors of vertices 3 and 50 in the second iteration. Hence, 50 $\not\in R(3)$. Note that using the k-hop label in the same example will not prune the vertex 50 because the minimum cost of matching the k-hop labels
is within $\alpha$ as shown in table 4.2. Therefore, NL label is more efficient compared to k-hop label as it subsumes the latter label.

4.2.3 Candidate set verification

The pruning methods based on the k-hop and the NL labels start with a $R'(u)$ and prune some of the candidate vertices based on the conditions described in theorems 2 and 3. The verification step reduces $R'(u)$ to $R(u)$ by retaining only those vertices $v$ for which there exists an isomorphism $\phi$ in which $\phi(u) = v$. Informally, it does this by checking if the pattern $P$ can be embedded at $v$ such that total cost of label mismatch is at most $\alpha$.

A vertex $v \in R(u)$ iff for any walk $w_p = (u_0 = u, u_1, \ldots, u_m)$ that covers all the edges in pattern $P$ there exists at least one walk $w_d = (v_0 = v, v_1, \ldots, v_m)$ in the database $G$ and satisfying the following three conditions: i) $u_i = u_j \Rightarrow v_i = v_j$ ii) $(v_i, v_{i+1}) \in E_G$ iii) $\sum C[L(u_i)][L(v_i)] \leq \alpha$. Unlike the NL label condition, the above conditions are necessary and sufficient and can be verified by following the definition of isomorphism.

Now, to check whether $v \in R(u)$, we first map $u$ to $v$ and subtract the cost of $C[L(u)][L(v)]$ from the threshold $\alpha$. We then try to map the remaining vertices in $P$ by following $w_p$ one edge at a time. In any step $(u_i, u_{i+1})$, if $u_i$ and $u_{i+1}$ are mapped to $x$ and $y$ respectively then we ensure that $(x, y) \in E_G$ (condition ii). If on the other hand, $u_{i+1}$ is not mapped then we map it to some vertex in $y \in R'(u_{i+1})$ and subtract the cost $C[L(u_{i+1})][L(y)]$ from the remaining $\alpha$ threshold. We back track if the remaining threshold is less than 0. The vertex $v \in R(u)$, if we can complete the walk $w_p$ satisfying the above three conditions.

Consider checking whether the vertex $30 \in R(1)$ in the pattern in the figure 4.1b and let $\alpha = 0.5$. The sequence $w_p = 1, 2, 4, 3, 1$ is a walk in the pattern that covers all the edges. In general, finding a walk that covers all the edges in a graph is a special case of Chinese postman problem [64]. We first map 1 to 30 and subtract the cost $C[L(1)][L(30)] = 0.2$ from 0.5. In the first step (1, 2), since 2 is not mapped we map it some vertex, say 20. The cost of the mapping is 0.2 and the remaining threshold is $0.3 - 0.2 = 0.1$. It can be verified that these mappings cannot complete
the walk \( w_p \). So we backtrack and map 2 to another vertex say 10. The walk can be completed with the mappings as in \( \phi_1 \) in Table 4.1d and the remaining cost is 0.1. The mappings of the pattern vertices not only implies that 30 ∈ \( R(1) \), it also tells us that 10, 60, 40 represent vertices 2, 3, 4 respectively. The above procedure can be easily extended to enumerate all the isomorphims of the pattern.

4.2.4 Label costs and dominance checking

Candidate representative vertices are pruned by checking for dominance relation between the NL labels of pattern vertex and that of candidate vertex in the database. Comparing the NL labels requires i) computing the cost of matching the k-hop labels ii) matching the neighbors of pattern vertex with neighbors of the candidate vertex. First problem can be formulated as a minimum cost maximum flow in a network and the second as maximum matching in a bipartite graph.

**Computing k-hop label cost:** The minimum cost of matching the k-hop labels \( h_k(u) \) and \( h_k(v) \) is equal to minimum cost for maximum cost in a flow network \( F \) defined as follows. Each edge in \( F \) is associated with a maximum capacity and a cost for sending one unit of flow across it. The network contains a vertex for each label \( l_u = L(u') \) where \( u' \in h_k(u) \) and a vertex for each label \( l_v = L(v') \) where \( v' \in h_k(v) \). There is a directed between between source vertex \( (s) \) and each \( l_u \) with zero cost and a capacity equal to the multiplicity of the \( l_u \) i.e., the number of vertices in \( h_k(u) \) that have the label \( l_u \). Similarly there is a directed edge between \( l_v \) and the sink node \( (t) \). In addition, there is a directed edge from \( l_u \) to \( l_v \) with a cost equal to \( C[l_u][l_v] \) and a capacity equal to the multiplicity of \( l_u \). The cost between the k-hop labels is equal to the minimum cost for maximum flow if the maximum flow is equal to \( |h_k(u)| \) and \( \infty \) otherwise.

Figure 4.3 shows the flow network required to compute the minimum cost of matching the k-hop labels \( h_2(2) = 4, 6 \) and \( h_2(20) = 40, 50, 60 \) as shown in Table 4.1. The labels of vertices in the k-hop labels are \( C, D \) and \( B, B, A \) respectively. There is an edge from \( s \) to each of \( C, D \) with zero cost and maximum capacity of one. Similarly, there is an edge from each of \( A, B \) to the sink vertex \( t \) with zero cost and maximum capacity of one and two respectively. The capacity of the edge betweenn \( B \) and
t is two because both the vertices 40 and 50 have the same label $B$. There is an edge from $C, D$ to each of $A, B$ with cost equal to the corresponding entry in the cost matrix $C$. The maximum flow in the network is two and the minimum cost of sending two units of flow 0.4 is achieved by pushing a unit flow along the paths $s, C, B, t$ and $a, D, A, t$. Therefore, the cost of matching the labels $h_2(2)$ and $h_2(20)$ is 0.4. It implies that the vertex 4 with label $C$ can be matched to either 40 or 50 and the vertex 6 to 60.

**Dominance check:** Consider the NL labels $\eta_{k+1}(u) = (A, B)$ and $\eta_{k+1}(v) = (A', B')$, the cost of matching the k-hop labels $B$ and $B'$ can be computed using the above the network formulation. Finding an injective function $f: A \rightarrow A'$ such that $a \preceq f(a)$, is equivalent to find a matching of size $|N(u)|$ in the bipartite graph with edges $(a, a')$, for all $a \in A$ and $a \preceq a'$. The NL label $\eta_k(u)$ is therefore dominated by $\eta_k(v)$ if the cost between the k-hop labels is within $\alpha$ and the size of maximum bipartite matching is $|N(u)|$.

**Optimization:** The candidate pattern may contain groups of symmetric vertices that are indistinguishable with respect to the k-hop label. In such a scenario, the candidate representative sets of all these vertices are exactly the same. Utilizing the symmetry, we can apply the pruning label strategy only on one vertex per symmetry group and replicate the results for all other vertices in the group. For example, the vertices 1 and 4 in figure 4.1b are symmetric and the representative sets $R(1)$ and $R(4)$ are exactly the same. In abstract algebra terms such groups are called orbits of the graph and can be computed using nauty algorithm [65]. Even though computing the orbits is expensive, we can avoid $(|g| - 1) \times |R'(u)|$ NL label cost computations where $g$ is the size of an orbit. Note that we find the orbits only for the pattern which is usually very small compared to the database graph.

### 4.2.5 Precomputing database k-hop labels

The k-hop label of a database vertex is independent of the candidate pattern. Also, the flow network to compute the cost of matching the k-hop labels requires only the aggregate information about the multiplicity of the vertex label in the k-hop label. Hence, we can precompute the k-hop label of the database vertices and store
them in the memory. The following theorem proves that computing k-hop label is expensive.

**Theorem 4.** k-reachable (KR) : Given a graph $G$, $k$ and $u \in V_G$. Compute $h_k(u)$. KR cannot be solved in polynomial time unless $P = NP$.

**Proof 3.** We prove this by reducing Hamiltonian path (HP) to KR. Hamiltonian Path : Given a graph $G$, is there a simple path of length $|V_G| - 1$ i.e. is there a path that visits each and every vertex exactly once. The problem of finding a Hamiltonian path is NP-Complete [59].

Assume that algorithm $X(k)$ can compute KR in polynomial time. Let $|V_G| = n$ and $u$ be the starting vertex in HP if it exists. Given an instance of HP, we first get a vertex $v$, $u \xrightarrow{n-1} v$ using $X(n-1)$. The vertex $v$ is removed from the graph and we find a vertex $v'$ such that $u \xrightarrow{n-2} v'$ and $(v', v) \in E_G$. We repeat this process $n - 1$ times. If at any stage $X(j) = \emptyset$ then we restart from a different starting vertex. The vertices selected in each iteration lie on a path of length $n - 1$ if it exists. If there is polynomial time algorithm for KR then HP could be solved in polynomial time by reducing it to KR. Therefore, KR is at least as hard as HP. So, KR cannot be solved in polynomial time unless $P = NP$.

To compute k-hop label of a vertex $u$, we check for each vertex $v$ whether $v \in u \xrightarrow{k} v$ by enumerating all possible $k$ length paths until a path is found. This procedure is exponential, we therefore fix a maximum value $k_{max}$ and use the NL label based pruning only for values of $k \leq k_{max}$. It only takes a couple of minutes to compute the k-hop label for $k \leq 6$ for all the vertices in the database graph. This is significantly less than the overall run time of the algorithm. Once $h_k(u)$ is computed we store in memory only the tuples $(l, m)$ where $m$ is the multiplicity of the label $l = L(u')$. The total amount of main memory required to store the precomputed k-hop labels is $O(|V_G| \times |\Sigma| \times k_{max})$.

### 4.3 Mining algorithm

The mining algorithm involves candidate generation, support computation in addition to finding the set of isomorphisms. Representative sets of vertices described
in the previous section are a compact view of all the isomorphisms of the pattern in the input graph. In this section, we will show how the representative sets can be used in conjunction with different candidate generation and support computation techniques to yield approximate graph mining algorithm with different properties.

4.3.1 Candidate generation

The search space of the frequent patterns forms a partial order. It can be explored in a depth first or breadth first order but doing so requires computing canonical code to avoid the duplicates. Since, the search space is exponential, sampling methods have gained traction in recent times [51, 66].

In our experiments we employed the random walk strategy proposed in [26] to mine exact patterns from a single large graph. Each random walk starts with an empty pattern and repeatedly adds new edges (to new vertices) or connects two existing vertices in the pattern to generate a new candidate. At any stage of the walk let $Q$ be the current frequent pattern. A candidate pattern $P$ is generated from $Q$ either by adding a new vertex with label $l$ or by connecting two existing vertices $u, v \in V_Q$. For any vertex $u$, if $u \in V_P \cap V_Q$ then the candidate representative set $R'(u)$ in $P$ is same as representative set $R(u)$ verified for $Q$. Otherwise $u \in V_P \setminus V_Q$ and the candidate representative set is $R'(u) = \{v|v \in V_G, \mathcal{C}[L(u)][L(v)] \leq \alpha\}$ i.e., we start with the current representatives if the vertex is already present otherwise it is the set of vertices in $G$ whose label matching cost is within $\alpha$. Using the label pruning and verification mechanism we compute the representatives of $P$. Then we decide if the pattern is frequent using the support function that we will define in section 4.3.2. If the candidate pattern $P$ is frequent, then we continue the walk by extending $P$. Otherwise, we try another extension from $Q$. If no extension of
leads to a frequent pattern then $Q$ is a maximal and we terminate the current random walk. The algorithm terminates when $K$ walks have been done, or when $K$ distinct maximal approximate patterns have been output. But, if the application requires a complete set of maximal patterns an ordered exploration of the search space may be employed.

### 4.3.2 Support computation

The support of a pattern is an anti-monotonic function on the set of isomorphisms of the pattern. The anti-monotonicity means that the support of a pattern cannot be greater than the support of any of its subgraph. Therefore, if a candidate pattern is found to be infrequent we can prune the entire subtree under it from the search space. This helps in pruning the otherwise exponential search space.

When mining from a database of graphs, a function as simple as the total number of graphs having at least one isomorphism is anti-monotonic. This approach cannot be used when mining from a single graph as it leads to a binary support function which is not very informative. On the other hand, counting the number of isomorphisms is not anti-monotonic because a graph can have more isomorphisms compared to its subgraph.

An anti-monotonic support function for a single graph is the maximum number of vertex disjoint isomorphisms. However, this requires computing the maximum independent set (MIS) of graph where each vertex represents an isomorphism. This is called the MIS support of the pattern. Clearly, it is not feasible to compute the MIS support when the input graphs are large and patterns have large number of isomorphisms. An easy upper bound on the MIS support is the size of the smallest representative set of a vertex in the pattern. Define the support of pattern $P$ in a database graph $G$ as

$$sup(P) = \min_{u \in V_P} \{|R(u)|\}$$

That is, the minimum cardinality over all representative sets of vertices in $P$. This definition of support was previously defined as Minimum Image Support in [24]. The size of representative sets constructed from the disjoint isomorphisms is equal to the MIS support. Hence, $sup(P)$ is at least as large MIS support. Other upper
bounds for the MIS value have been proposed in gApprox [28] and CMDB-Miner [25] algorithms. The support function used in gApprox can be computed from the representative sets by enumerating the isomorphisms as described in the Section 4.2.3. The support function used by the CMDB-Miner algorithm can also be used by constructing the appropriate flow network on the representative sets.

In conclusion, we can mix and match different techniques for candidate generation and support computation to produce different versions of the approximate graph mining algorithm even though the isomorphisms are stored as representatives.

4.3.3 Complexity

*Space Complexity*: At any given stage of the mining process, we need to store the candidate representative sets and the precomputed k-hop labels. For a pattern with \( m \) vertices, the total amount of memory is \( O(m \times |V_G| + k_{max} \times (|\Sigma| \times |V_G|)) \). The first term corresponds to the representative sets and the second to the precomputed k-hop labels. \( k_{max} \) is the maximum value of \( k \) for which we compute k-hop labels.

*Time Complexity*: The cost of matching the k-hop labels requires at most \( |h_k(u)| \) augmentations in the flow network \( F \) which is an upper bound on the min cost assuming the cost on each edge is at most one. Each augmentation involves cycle detection which takes \( O(|\Sigma|^3) \) because the number of vertices in \( F \) is \( O(|\Sigma|) \). The time for the bipartite matching in a graph is proportional to the number of vertices and the edges in it. Since, we try to match the neighbors of pattern and candidate vertex, the number of vertices is bounded by the maximum degree \( d_{max} \) of the pattern and candidate vertices. Therefore, the total time for each dominance check is \( O(|h_k(u)| \times |\Sigma|^3 + d_{max}^3) \). The number of dominance checks performed per candidate are \( O(k_{max} \times n_g \times |V_G|) \) where \( n_g \) is the number of orbit groups in the pattern vertex.

4.4 Experimental evaluation

We ran experiments on several real world datasets to evaluate the performance of our algorithm. All the experiments were run on an 4GB Intel Core i7 machine with a clock speed of 2.67 GHz running Ubuntu Linux 10.04. The code was written
in C++ and compiled using g++ version 4.4 with -O3 optimization flag. The default number of random walks is $K = 500$.

Table 4.3: (a): Input graph statistics, (b): Maximal pattern statistics (the numbers shown are average values).

| Dataset | $|V|$ | $|E|$ | $|\Sigma|$ | Preprocessing time |
|---------|-----|-----|------|------------------|
| CMDB    | 10466 | 15122 | 84   | 329.31s          |
| SCOP    | 39256 | 154328 | 20   | 17.377s          |
| PPI     | 4950  | 16515 | 4950 | 339.45s          |

| Dataset | $|V|$ | $|E|$ | Degree |
|---------|-----|-----|--------|
| CMDB    | 11  | 12.24 | 2.223  |
| SCOP    | 5.965 | 6.725 | 2.225  |
| PPI     | 6.453 | 5.956 | 1.655  |

4.4.1 Configuration management DB (CMDB)

A CMDB is used to manage and query the IT infrastructure of an organization. It stores information about the so-called configuration items (CIs) – servers, software, running processes, storage systems, printers, routers, etc. As such it can be modeled as a single large multi-attributed graph, where the vertices represent the various CIs and the edges represent the connections between the CIs (e.g., the processes on a particular server, along with starting and ending times). Mining such graphs is challenging because they are large, complex, multi-attributed, and have many repeated labels. We used a real-world CMDB graph for a large multi-national corporation (name not revealed due to non-disclosure issues) from HP’s Universal Configuration Management Database (UCMDB). Table 4.3a shows the size of the CMDB graph.

Cost Matrix: The set of labels in a CMDB form a hierarchy which can be obtained from HP’s UCMDB. In the absence of domain knowledge, one way to obtain a cost matrix is by assigning low costs for pairs of labels that share many ancestors in the hierarchy and high costs otherwise. The algorithm is general in that it doesn’t depend on how the label matching costs are assigned, the range of these values
or whether the cost matrix is symmetric. Consider any two labels $l_1$, $l_2$ and their corresponding paths $p_1$, $p_2$ to the root vertex in the hierarchy. We first define the similarity between the labels to be proportional to the number of common labels in $p_1 \cap p_2$, as follows $\text{sim}(l_1, l_2) = \frac{|p_1 \cap p_2|}{\frac{1}{|p_1|} + \frac{1}{|p_2|}}$. The cost of matching the labels is then $C[l_1][l_2] = 1 - \text{sim}(l_1, l_2)$.

<table>
<thead>
<tr>
<th>minsup</th>
<th>Time</th>
<th>Avg Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>7931.56</td>
<td>15.86</td>
</tr>
<tr>
<td>15</td>
<td>7147.11</td>
<td>14.29</td>
</tr>
<tr>
<td>10</td>
<td>5604.35</td>
<td>11.21</td>
</tr>
</tbody>
</table>

**Results:** Table 4.4 shows the time for $K = 500$ random walks for different values of $\text{minsup}$ and $\alpha = 0.5$. Somewhat counter-intuitively, the time increases for higher minimum support values. This may be explained by the fact that the CMDB dataset contains many smaller subgraphs with low support (with few isomorphisms) and relatively fewer large subgraphs with high support (with many isomorphisms). Thus, when we lower the minimum support, the random edge extensions converge to smaller maximal patterns, whereas for high minimum support, many of the random extensions are infrequent, forcing the method to go through many more steps to find the maximal patterns.

**Example Patterns:** Figure 4.4 shows a maximal approximate pattern found in the CMDB graph, representing a typical “de-facto” configuration of the IT infrastructure in this company. It shows the connection between some services running on an NT server, and also the web/ftp services. The node with label $9 \times \text{process}$ indicates
that there are nine nodes in the maximal pattern with label \textit{process} all of which are connected to the \textit{nt} node. This is an example where the run time for computing the representative sets is significantly reduced by the optimization proposed in section 4.2.4. All of the nine nodes belongs to the same orbit and hence their representative sets are identical.

To show the effectiveness of the pruning based on labels, we compared the time taken to enumerate a single maximal pattern in the CMDB graph. We compared the time with and without label-based pruning. Both the methods terminated the random walk with the maximal pattern shown in Figure 4.5. However, the total time taken to enumerate the pattern without using any derived label is 18306 secs whereas by using the NL label the total time reduced to only 15.58 secs. The huge difference between the times arises due to the multiplicity effect in CMDB graphs.

4.4.2 Protein structure dataset (SCOP)

SCOP (scop.mrc-lmb.cam.ac.uk/scop/, accessed 27-July-2015) is a hierarchical classification of proteins based on structure and sequence similarity. The four levels of hierarchy in this classification are: class, fold, superfamily and family. The 3D structure of a protein can be represented as an undirected graph with the vertex labels being the amino acids, with an edge connecting two nodes if the distance between the 3D coordinates of the two amino acids (their \(\alpha\)-Carbon atoms) is within a threshold (we use 7 Angstroms). We constructed a database of 100 protein structures belonging to 5 different families with 20 proteins from each family. We chose the proteins from different levels in the SCOP hierarchy, and we also focused on large proteins (those with more than 200 amino acids). The 3D protein structures were downloaded from the protein data bank (http://www.rcsb.org/pdb, accessed
The database can be considered as a single large graph with 100 connected components. The graph characteristics are shown in Table 4.3a. For the SCOP dataset, the support is redefined as the number of proteins containing the pattern, i.e., even if a protein contains multiple embeddings we count them only once for the support.

Cost Matrix: Since there are 20 different amino acids, we need a $20 \times 20$ cost matrix. BLOSUM62 [67] is a commonly used substitution matrix for aligning protein sequences. The $i,j$ entry in BLOSUM denotes the log-odd score of substituting the amino acids $a_i$ and $a_j$, defined as: $B[i][j] = \frac{1}{\lambda} \log \frac{p_{ij}}{f_i f_j}$, where $p_{ij}$ denotes the probability that $a_i$ can be substituted by $a_j$; $f_i$, $f_j$ denote the prior probabilities for observing the amino acids; and $\lambda$ is a constant. We compute $f_i$ and $f_j$ from the database, and then reconstruct $p_{ij} = f_i f_j e^{\lambda B[i][j]}$. Next, we define the pair-wise amino acid cost matrix as $C[i][j] = 1 - \frac{p_{ij}}{p_{ii}}$, which ensures that the diagonal entries are $C[i][i] = 0$.

<table>
<thead>
<tr>
<th>$\alpha$</th>
<th>NL pruning</th>
<th>no pruning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.01</td>
<td>57.27</td>
<td>115.06</td>
</tr>
<tr>
<td>0.7</td>
<td>228.65</td>
<td>394.93</td>
</tr>
<tr>
<td>1.0</td>
<td>2689.54</td>
<td>5966.53</td>
</tr>
<tr>
<td>1.5</td>
<td>6376.93</td>
<td>12572.96</td>
</tr>
</tbody>
</table>

Table 4.5: SCOP: Effect of $\alpha$

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Time</th>
<th>Avg Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>NL Pruning</td>
<td>2689.54</td>
<td>5.37</td>
</tr>
<tr>
<td>no Pruning</td>
<td>5966.53</td>
<td>11.93</td>
</tr>
<tr>
<td>gApprox</td>
<td>15653.2</td>
<td>31.30</td>
</tr>
</tbody>
</table>

Table 4.6: SCOP: Runtime of Different Algorithms

Results: Table 4.5 shows the time taken for enumerating approximate maximal patterns for different values of $\alpha$ (with fixed $minsup = 20$). The table shows the time for $K = 500$ random walks with and without the label pruning. It can be seen that by using the label-based pruning the time for random walks reduces significantly (by over 100%). As expected, the time increases as the values of $\alpha$ increases, since the
Table 4.7: SCOP: Relative number of matching neighbors, k-hop distance and verification computations performed for different values of minsup.

<table>
<thead>
<tr>
<th>minsup</th>
<th>neighbor checks</th>
<th>k-hop checks</th>
<th>Verify checks</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>20</td>
<td>1.14</td>
<td>1.41</td>
<td>0.03</td>
</tr>
<tr>
<td>25</td>
<td>0.97</td>
<td>1.42</td>
<td>0.87</td>
</tr>
<tr>
<td>40</td>
<td>0.89</td>
<td>2.13</td>
<td>1.52</td>
</tr>
</tbody>
</table>

Table 4.8: SCOP: Relative number of forward, back extensions that were tried and number of extensions that were successful.

<table>
<thead>
<tr>
<th>minsup</th>
<th>Fwd</th>
<th>Fwd Success</th>
<th>Back</th>
<th>Back Success</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>20</td>
<td>1.27</td>
<td>0.47</td>
<td>0.65</td>
<td>0.38</td>
</tr>
<tr>
<td>25</td>
<td>1.32</td>
<td>0.43</td>
<td>0.59</td>
<td>0.29</td>
</tr>
<tr>
<td>40</td>
<td>1.14</td>
<td>0.40</td>
<td>0.47</td>
<td>0.22</td>
</tr>
</tbody>
</table>

Table 4.9: SCOP: Time for Different Steps vs. minsup

<table>
<thead>
<tr>
<th>minsup</th>
<th>k-hop label</th>
<th>NL label</th>
<th>Verification</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>377.10</td>
<td>378.99</td>
<td>629.804</td>
<td>1796.61</td>
</tr>
<tr>
<td>20</td>
<td>476.10</td>
<td>489.59</td>
<td>1188.25</td>
<td>2689.54</td>
</tr>
<tr>
<td>25</td>
<td>462.52</td>
<td>513.25</td>
<td>1055.28</td>
<td>2572.33</td>
</tr>
<tr>
<td>40</td>
<td>461.30</td>
<td>625.52</td>
<td>1148.42</td>
<td>2818.81</td>
</tr>
</tbody>
</table>

number of isomorphisms clearly increases for a more relaxed (larger) cost threshold. When \( \alpha = 0.01 \), the patterns are exact as \( C[i][j] > \alpha \), \( \forall i \neq j \).

Table 4.6 compares the time taken to mine 500 maximal patterns from the SCOP dataset using the NL label algorithm and two other algorithms, with \( \text{minsup} = 20 \) and \( \alpha = 1.0 \). The no pruning algorithm computes the representative sets from the candidate representative sets directly using the verification procedure described in section 4.2.3. The gApprox algorithm is based on [28] and stores all isomorphisms during the course of enumerating a maximal pattern. It can be see that the run time for the NL based algorithm is significantly less as it prunes invalid candidates without performing an expensive verification procedure or storing a large number of isomorphisms.

Table 4.9 shows the time taken for \( K = 500 \) random walks for various values of \( \text{minsup} \), with \( \alpha = 1.0 \). The table shows the time spent in k-hop matching, NL matching and pattern verification steps. In general, the time increases as \( \text{minsup} \)
increases because the representative sets $R(u)$ become larger. However, there is no fixed trend as the total time depends on the regions of pattern space that the random walk explores, as previously explained for the CMDB graph.

<table>
<thead>
<tr>
<th>$\alpha$</th>
<th>NL label</th>
<th>k-hop label</th>
<th>Verification</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.5</td>
<td>119.84</td>
<td>35.53</td>
<td>13.71</td>
<td>169.08</td>
</tr>
<tr>
<td></td>
<td>250.57</td>
<td>36.33</td>
<td></td>
<td>286.9</td>
</tr>
<tr>
<td>0.75</td>
<td>293.40</td>
<td>127.96</td>
<td>224.55</td>
<td>645.91</td>
</tr>
<tr>
<td></td>
<td>368.84</td>
<td>653.14</td>
<td></td>
<td>1021.98</td>
</tr>
</tbody>
</table>

Table 4.10 compares the effectiveness of NL and k-hop labels for different values of the threshold $\alpha$. For each value of $\alpha$, the top row shows the time with NL label, whereas the bottom row shows the time using only the k-hop label. The NL label clearly reduces the time taken. In fact, it reduces the time for both the k-hop matching and the pattern verification steps, since NL is very effective in pruning the representative set. This effect is best seen for $\alpha = 0.75$, where the total time for verification reduces even though matching the neighbors takes more time compared to k-hop matching. This shows the effectiveness of the NL label versus k-hop label in isolation.

![Pattern and Structural Motif](image)

Figure 4.6: SCOP: Approximate Pattern and its Structure

Example Patterns: Figure 4.6 show an example approximate protein graph pattern and its corresponding 3D structure extracted from the SCOP dataset. For example, the graph in 4.6a has support 19, and the structure of one its occurrences, in protein
PDB:1R2E, is shown in 4.6b. The common motif comprises the black colored amino acids some of whom are far apart in sequence but are spatially close in 3D. It is important to note that the cost of this isomorphism is \( C(\phi) = 0.4541 \), indicating that exact isomorphism cannot find the motif.

### 4.4.3 Protein-Protein interaction network (PPI)

We ran experiments on a yeast (Saccharomyces cerevisiae) PPI network. The list of interacting proteins for yeast was downloaded from the DIP database (http://dip.doe-mbi.ucla.edu, accessed 27-July-2015). As seen in Table 4.3a, the PPI network has 4950 proteins and 16,515 interactions. Unlike the other datasets, each node in the PPI network essentially has a unique label, which is the protein name.

**Cost Matrix:** To construct the cost matrix for the protein network we consider the similarity between the protein sequences for any two adjacent nodes. Sequence similarity is obtained via the BLAST alignment score [68], that returns the expected value (E-value) of the match. A low E-value implies high similarity, thus we create a binary cost matrix between the proteins by setting \( C[p_i][p_j] = 0 \) iff the proteins \( p_i \) and \( p_j \) have high similarity, i.e., iff \( E-value(p_i, p_j) \leq \epsilon \). We empirically set \( \epsilon = 0.003 \).

<table>
<thead>
<tr>
<th>minsup</th>
<th>Time</th>
<th>Avg Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>834.55</td>
<td>1.67</td>
</tr>
<tr>
<td>5</td>
<td>377.56</td>
<td>0.76</td>
</tr>
<tr>
<td>10</td>
<td>254.24</td>
<td>0.51</td>
</tr>
</tbody>
</table>

Table 4.11: Time for random walks in PPI Dataset

<table>
<thead>
<tr>
<th>minsup</th>
<th>neighbor checks</th>
<th>Verify checks</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>0.61</td>
<td>0.11</td>
</tr>
<tr>
<td>10</td>
<td>0.43</td>
<td>0.08</td>
</tr>
</tbody>
</table>

Table 4.12: PPI: Relative number of matching neighbors, and verification computations performed for different values of minsup.

**Results:** Table 4.11 shows the time for \( K = 500 \) random walks in the yeast PPI network for different values of \( \text{minsup} \). It can be seen that the time for random walks decreases as the support value increases. One of the differences for the PPI
Table 4.13: PPI: Relative number of forward, back extensions that were tried and number of extensions that were successful

<table>
<thead>
<tr>
<th>minsup</th>
<th>Fwd</th>
<th>Fwd Success</th>
<th>Back</th>
<th>Back Success</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>0.41</td>
<td>0.34</td>
<td>0.18</td>
<td>0.88</td>
</tr>
<tr>
<td>10</td>
<td>0.18</td>
<td>2.14</td>
<td>0.88</td>
<td>3.6</td>
</tr>
</tbody>
</table>

The complexity of matching the k-hop labels depends on the number of literals in the k-hop label. As each label (protein) is unique in the PPI graph, the number of literals in the k-hop label of a vertex $v$ in a PPI network is equal to the number of vertices reachable in $k$ hops. This increases the run time for the k-hop label matching. Therefore, for mining PPI networks we use only the NL labels.

Example Patterns: Figure 4.7a shows a mined maximal frequent approximate pattern (using $\text{minsup} = 5$). The proteins are labeled with their DIP identifiers (e.g., DIP-2818N); the last number in the label is just a sequential node id. It is worth emphasizing that exact subgraph isomorphism would not yield any patterns in this dataset, since each label is unique. However, since we allow a protein to be replaced by a similar protein via the cost matrix $\mathcal{C}$, we obtain interesting approximate pat-
terns. To judge the quality of the mined patterns we use the gene ontology (GO; www.geneontology.org, accessed 27-July-2015), which comprises three structured, controlled vocabularies (ontologies) that describe gene products in terms of their associated biological processes (BP), molecular functions (MF), and cellular components (CC). For each of the mined approximate patterns we obtain the set of all the GO terms common to all proteins in the pattern. This serves as an external validation of the mined results, since common terms imply meaningful biological relationships among the proteins. Figure 4.7b shows the common GO terms for the pattern in Figure 4.7a. This subgraph comprises proteins involved in proteolysis as the biological process, i.e., they act as enzymes that lead to the breakdown of other proteins into amino acids. Their molecular function is endopeptidase activity, i.e., breakdown of peptide bonds of non-terminal amino acids, in particular the amino acid Threonine. These proteins are located in the proteasome storage granule, and most likely comprise a protein complex (proteasome) – a molecular machine – that digests proteins into amino acids.
CHAPTER 5
Coverage Based Pattern Mining in Probabilistic Graphs

In this chapter, we tackle the problem of mining frequent subgraphs from probabilistic graphs. We deal with labeled graphs in which each edge is associated with a probability of existence. Extending the existing methods requires setting of many non-intuitive parameters. Therefore, we propose a novel concept, coverage, which quantifies the possibility that an edge in the input graph is covered by isomorphisms of patterns. Accordingly, we define an objective function whose optimal solutions (set of patterns) possess two highly desired properties: 1. Patterns have high support and 2. Inter-pattern similarity is low. We will prove that the problem is NP-Hard and discuss two greedy algorithms for maximizing the objective function.

5.1 Motivation

In many real-world scenarios, the relationships between pairs of objects are often uncertain. It is important that the uncerly graph models take into account these uncertain relationships. These vague relationships arise in the data due to various reasons as explained below.

- **Data Collection**: Interactions between the proteins in a PPI network are often inferred based on the experimental evidence [69]. However, bulk of the existing work on analyzing PPI networks ignores the fact that the experiments are error prone and that the edges are associated with probabilities [27], [70], [71] etc.

- **Privacy Preservation**: Probabilistic graphs are also used in preserving the privacy of edges in a weighted social network while at the same time maximizing the amount of information shared. In some cases uncertainties are artificially introduced to preserve the privacies.

- **Inferred Social Networks**: Relationships between the users in a network are not always explicit. It is possible to infer these relationships implicitly based
on the historical interactions between the users. Hence these relationships are not certain and almost always are associated with a probability [72].

In this chapter, we discuss the possibility of extending the frequent sub-graph mining methods that were proposed to mine frequent patterns from exact/certain graphs (to distinguish from probabilistic or uncertain graphs). We will show that such an extension is not scalable and requires experimenting with many non-intuitive parameters to extract meaningful frequent patterns.

### 5.1.1 Novelties in our approach

The method that we propose in this chapter differs from the existing approaches in the following ways.

- **Parameter free:** Instead of requiring the user to input multiple non-intuitive parameters, our method requires a single parameter \( k \) which is just the number of patterns to return. The objective function that we propose ensures that the \( k \) patterns have high frequency and are also diverse. The method can be easily made parameter free by stopping the algorithm as it reaches a certain state where the new patterns are relatively similar to the previously returned patterns.

- **Works on single graphs:** Our method works in both single graph and database of graphs setting.

- **Avoids exponential enumeration:** All the existing approaches require pattern embeddings to estimate the support. As we have already seen such methods are not scalable. The method we propose in this chapter estimates the support of a pattern without enumerating its embeddings. This is similar to the algorithms that we have proposed in the previous chapters.

### 5.2 Preliminaries

**Uncertain graph:** An uncertain graph \( G \) is given by the four tuple \( G = (V_G, E_G, L, P) \). The function \( L : V_G \rightarrow \Sigma \) gives the label for each vertex in the
The function \( P : E_G \rightarrow \mathbb{R}^{[0,1]} \) assigns the probability of existence for each edge in the uncertain graph. Probability of the edge \( e = (u, v) \in E_G \) is given by \( P[e] \). We assume that the edge probabilities are independent.

Possible graphs: From an uncertain graph \( G \), a number of certain graphs can be instantiated. Each such certain graph is called a possible world and it has a certain probability of existence. The probability of a possible world \( \mathbb{G} = (V_G, E_G, L) \) is given by \( \mathbb{P}[\mathbb{G}] = \prod_{e \in E_G \cap E_\mathbb{G}} P[e] \times \prod_{e \in E_G - E_\mathbb{G}} (1 - P[e]) \). We denote the set of all possible worlds of the uncertain graph by \( \mathbb{W} = G^1, G^2, \ldots, G^{2^{|E_G|}} \). We denote the possible graph with maximum number of edges by \( \mathbb{G}^{*} \). It is also called the Maximum Certain Graph (MCG) of the uncertain graph. In other words MCG is the certain graph where we include all the edges from \( E_G \). It’s probability is \( \mathbb{P}[\mathbb{G}^{*}] = \prod_{e \in E_G} P[e] \).

Pattern: Pattern is represented by a three tuple \( P = (V_P, E_P, L) \) where \( V_P \) is the set of vertices, \( E_P \subseteq V_P \times V_P \) is the set of edges and the function \( L : V_P \rightarrow \Sigma \) maps each vertex in the pattern to a fixed label. The edges and the vertex labels are fixed in the pattern.

Embedding of a pattern: Embedding of a pattern in a certain graph is function that maps pattern vertices to graph vertices such that the labels and edges are preserved. In other words, a function \( \phi : V_P \rightarrow V_G \) is called an embedding of \( P \) in \( G \) iff \( L(u) = L(\phi(u)) \) and \( (u, v) \in E_P \implies (\phi(u), \phi(v)) \in E_G \).

A function \( \phi : V_P \rightarrow V_G \) is called an embedding in the uncertain graph \( G \) iff \( \phi \) is an embedding of \( P \) in the graph \( \mathbb{G}^{*} \). Additionally, the probability that the embedding \( \phi \) exists in a possible graph is given by \( \mathbb{P}[\phi] = \prod_{e=(x,y) \in P} \mathbb{P}[e] \).

Coverage of an edge: In a certain graph \( G \), we say a pattern \( P \) covers an edge \( e \in E_G \) iff \( \exists \) an embedding \( \phi \) such that \( \phi(e') = e \) for some \( e' \in E_P \). We define a Boolean function \( cov(P, e, G) \) that is equal to 1 if such an embedding exists and 0 otherwise. The definition can be easily extended to the cases where we have a set \( S \) of patterns instead of a single pattern. The function \( cov(S, e, G) \) is equal to 1 if there is at least one embedding of some pattern in \( S \) which covers \( e \).

Coverage in Uncertain graphs: When dealing with uncertain graphs, we define the coverage of an edge in terms of the expected value of the coverage over all
possible worlds. Therefore, the coverage of an edge \( e \) in the uncertain graph \( \mathcal{G} \) is defined by the expected value of following random variable, \( X_e \).

\[
X_e = \begin{cases} 
1 & \text{if } \text{cov}(P, e, G) = 1 \\
0 & \text{otherwise}
\end{cases}
\]

Then, the coverage \( \text{cov}(P, e, \mathcal{G}) \) is given by

\[
\text{cov}(P, e, \mathcal{G}) = \mathbb{E}[X_e] = \mathbb{P}[X_e = 1] = \sum_{G \in \mathcal{W}} \text{cov}(P, e, G) \times \mathbb{P}[G]
\]  
(5.1)

For a set \( S \) of patterns the coverage is equal to the expected value of the random variable \( X'_e \) defined as follows:

\[
X'_e = \begin{cases} 
1 & \text{if } \exists P \in S, \text{cov}(P, e, G) = 1 \\
0 & \text{otherwise}
\end{cases}
\]

Then, the coverage \( \text{cov}(S, e, \mathcal{G}) \) is given by

\[
\text{cov}(S, e, \mathcal{G}) = \mathbb{E}[X'_e]
\]  
(5.2)

\textbf{Goal:} Goal of the mining algorithm is to find a set of \( k \) patterns that maximizes the sum of coverage of edges. Let \( \mathcal{F} \) the set of all possible patterns and \( S \subseteq \mathcal{F} \). Then the value of objective function, \( f \), for a set of patterns \( S \) is given by:
Figure 5.1: shows an uncertain graph with two edges 5.1a and its four possible worlds 5.1b- 5.1e.

We aim to maximize the objective function subject to $|S| \leq k$ for some user given parameter $k$. Therefore, the output of the mining algorithm is a subset of patterns $S$ that maximizes $f(S)$.

$$f(S) = \sum_{e \in E_G} \text{cov}(S, e, G)$$

$$= \sum_{e \in E_G} \mathbb{E}[X'_e]$$

$$= \sum_{e \in E_G} \mathbb{P}[X'_e = 1]$$

(5.3)

Example::

The figure 5.1a shows an uncertain graph with two edges. Four possible worlds (and their probabilities of existence) of the uncertain graph are shown in figures 5.1b, 5.1c, 5.1d and 5.1e. For any given edge $e = (u, v) \in G$, the value $\text{cov}(S, e, G)$ is maximized by the single edge pattern $P = (L(u), L(v))$. However, the summation across all edges is not necessarily maximized by the trivial one edge
patterns. In this simple example, we can see that the optimal $S$ (assuming $k = 1$) is the line graph $B - A - C$. It can be seen that $f(A - B) = 0.89$ and $f(A - C) = 0.9$ but $f(B - A - C) = 2 \times 0.9 \times 0.89$. Note that the optimal solution is $A - C$ if the existence probability of the edge $(10, 30) < 0.5$. This shows that our objective function formulation allows for non-trivial patterns.

5.2.1 Coverage computation is polynomial iff $P = NP$

Given an uncertain graph $G$, a pattern $P$ and an edge $e \in G$. We will show that $cov(P, e, G)$ can be computed in polynomial time iff $P = NP$. To make the proof simpler, we will work with the following variation of the coverage problem.

$H$: Given $G$, $P$. Is $cov(P, e, G) \leq 0$ ?

The problem $H$ is trivial if we can compute the value of $cov(P, e, G)$. Now consider the subgraph isomorphism problem $G$ defined as

$G$: Given graphs $H_1$ and $H_2$. Is $H_1 \cong H_2$ ?

Note that the problem $G$ is NP-Complete. Therefore, $G$ can be solved in polynomial iff $P = NP$.

- $H$ is in $P \rightarrow P = NP$ : Pick a vertex $v \in H_1$ and an edge $e = (v, u)$, where $L(u) \notin L(H_1)$. Similarly connect each vertex $v' \in H_2$, $L(v') = L(v)$ with a vertex labeled $u$. Now, $G$ is true iff $cov(H_1, e, H_2) > 0$. Hence, $H$ is in P iff $P = NP$.

- $P = NP \rightarrow H$ is in $P$ : If $P = NP$ then subgraph isomorphism can be solved in polynomial time. Now given a $cov(P, e, G)$ instance. Use the construction described above and check if $P \cup e \cong G \cup e$. It is easy to see that $cov(P, e, G) > 0$ iff $P \cup e \cong G \cup e$.

5.2.2 Complexity assuming an oracle for subgraph isomorphism

In this section, we will show that maximizing $f$ is NP-Complete even if we assume an oracle that can compute the value of $cov(P, e, G)$ in a certain graph in polynomial time. We do this by via a reduction from a variation of set cover problem.
Subset Cover: Let $U$ be an universe of elements and $\Omega = S_1, S_2, \ldots, S_M$ be a set of subsets of $U$ i.e., $S_i \subseteq U$. The problem is to find a subset $S \in 2^\Omega$, $|S| = k$ such that $|\cup S_i|$ is maximized. If $\exists S_i, S_j \in S$, $S_i \subseteq S_j$ then we can remove $S_i$ from $S$ as the value of any solution that contains $S_i$ cannot be reduced by replacing it with $S_j$. Therefore, we assume that $S_i - S_j \neq \phi$ for all $S_i \neq S_j$. Moreover, we can write the objective function $h$ as:

$$h(S) = \sum_{e \in \cup S_i} \delta(\exists S_i \in S, e \in S_i)$$  \hspace{1cm} (5.5)$$

The function $\delta$ is either 1 or 0 depending on whether the statement it takes is true or false respectively.

5.2.2.1 Reduction to coverage problem

Given an instance of set cover problem we will construct an equivalent instance of coverage problem. For every $S_i \in \Omega$, let $S_i = \{e_1, e_2, \ldots, e_{|S_i|}\}$. We assume that $|S_i|$ is even. The construction is similar in the case when it is odd. We add two sets of edges to the uncertain graph $G$ with each edge having a probability of existence equal to 1. First set $E_{i1} = \{(e_1^o, e_2^o), (e_2^o, e_3^l), \ldots, (e_{|S_i|-1}^o, e_{|S_i|}^o)\}$. Second, $E_{i2} = \{(e_1^o, e_2^l), (e_2^o, e_3^o), \ldots, (e_{|S_i|-1}^o, e_{|S_i|}^o)\}$. Figure 5.2 shows this construction pictorially.

The uncertain graph is the union $G = \bigcup_{i\leq M} E_{i1} \cup E_{i2}$. It is in fact an exact graph as the edge probabilities are 1. The pattern set $\mathbb{F}$ consists of $2 \times M$ patterns. The pair of edge sets $E_{i1}$ and $E_{i2}$ contribute to exactly one pattern in the form of
$E_{i1} \cup E_{i2}$ to the pattern set $\mathcal{F}$. Therefore, $\mathcal{F} = \{E_{11} \cup E_{12}, E_{21} \cup E_{22}, \ldots, E_{M1} \cup E_{M2}\}$.

The objective is to select a subset $S \subseteq \mathcal{F}$, $|S| = k$ such that the following function is maximized

$$f(S) = \sum_{e \in \mathcal{G}} \mathbb{E}[X_e = \exists s \in S, cov(s, e, \mathcal{G}) = 1]$$

$$= \sum_{e \in \mathcal{G}} \mathbb{P}[X_e = \exists s \in S, cov(s, e, \mathcal{G}) = 1]$$

(5.6)

Since, the graph $\mathcal{G}$ is certain the term inside the summation is either 0 or 1.

**Observations:**

From equations 5.5 and 5.6 we make the following observations.

- The terms inside the summations of both the equations are equivalent. This is because whenever a pattern $E_{i1} \cup E_{i2}$ is chosen the corresponding elements in the set $S_i$ are covered.

- Because the number of edges in the graph $\mathcal{G}$ are twice more than the number of elements in $\cup S_i$, we can show that $2 \times h(S) = f(S)$.

Therefore, the problem of maximizing $f$ is equivalent to that of maximizing $h$. Here we are assuming that $cov(S, e, \mathcal{G})$ can be computed in polynomial time.

### 5.3 Properties of the coverage function

In this section we will discuss the properties of the coverage function and also properties of the patterns in the set $S$ that maximizes 5.3. We show that maximizing the coverage implicitly captures three main properties: i) Patterns with high disjoint support ii) Diverse set of patterns iii) Longer patterns (whenever possible)

#### 5.3.1 Patterns with higher support

Consider the case where we restrict $k = 1$ in the objection function 5.4. This helps us to analyze the objective function without having to worry about the overlap among the patterns in the set $S$. 

Figure 5.3: Uncertain graph to show that the objective function prefers diverse patterns.

**Lemma 1.** Consider any pattern $P$. Let $\Phi_P = \phi_1, \phi_2, \ldots$ be a subset of embeddings of $P$ in the MCG graph $G^*$. Consider any subset $\Phi_S$ such that $\phi_i \cap \phi_j = \phi$ for all $\phi_i, \phi_j \in \Phi_S, i \neq j$. Let $\Phi_{S'}$ be the subset of disjoint embeddings that maximizes $p(\Phi_{S'}) = \sum_{\phi_i} P[\phi_i], \phi_i \in \Phi_{S'}$. Then, $cov(P, e, G) \geq p(\Phi_{S'}) \times |P|.$

**Proof.** Each embedding $\phi_i$ of the pattern contributes a value of $P[\phi_i] \times |P|$ towards the overall coverage value, where $|P|$ is the number of edges in the pattern. Now, since the embeddings are disjoint the overall coverage is at least the sum of individual contributions of the embeddings. Therefore, $cov(P, e, G) \geq p(\Phi_{S'}) \times |P|.$

We also note that the quantity $p(\Phi_{S'}) \geq Sup(P) \times P[\phi_{i^*}]$ where $P[\phi_{i^*}] \leq P[\phi_j], \forall \phi_j \in \Phi_{S'}$ where $Sup(P)$ is lower bound of the maximum disjoint support of the pattern. The coverage is maximized when the pattern has many disjoint embeddings with higher probability of existence. Therefore, our formulation prefers implicitly prefers candidates with higher expected support.

### 5.3.2 Diverse patterns

Our objective function formulation also prefers diverse set of patterns in the output. We will give a simple example to show why it prefers diverse patterns. It is difficult to prove the diversity of patterns in a general graph. Consider the $G$ as show in figure 5.3. Let $S_1 = A-B, B-A-C$ and $S_2 = A-B, A-C$ be two sets of patterns. Intuitively, the patterns in the set $S_2$ are more diverse compared to the patterns in $S_1$. We prefer the set $S_2$ to $S_1$ because it is more diverse.

The coverage of the pattern sets $S_1$ and $S_2$ are $f(S_1) = 0.7 + 0.8 \times 0.7 + 0.9 \times 0.7$ and $f(S_2) = 0.7 + 0.8 + 0.9$. Intuitively diverse patterns tend to cover different parts of the graph. Hence the higher coverage value. This shows that our objective function formulation prefers diverse set of patterns.
5.3.3 Non antimonotic property

The support functions proposed for measuring the frequency of a subgraph pattern usually satisfy the antimonotonic property which helps us to efficiently prune the search space. Even though the coverage function effectively captures the notion of frequency and diversity, it is not an anti-monotonic function. In other words, given two patterns $P_1, P_2, P_1 \subseteq P_2$. We cannot conclude if $\sum_e \text{cov}(P_1, e, G) \geq \text{cov}(P_2, e, G)$. We refer to the figure 5.1a for an example. We have already shown that $B - A - C$ has higher coverage compared to the pattern $B - A$ even though $B - A - C$ is a specialization of the pattern $B - A$. We also showed the conditions under which $B - A - C$ has a lesser coverage.

5.4 Greedy algorithms for maximizing the coverage

We have seen that maximizing the objective function in the equation 5.4 is NP-Hard even when we assume a polynomial time subgraph isomorphism checker. In this section, we propose two greedy algorithms to compute a set of patterns whose coverage is close to the optimal value. Both the approaches are similar in the sense that they iteratively construct patterns so that the coverage is increased by maximum in each step.

Algorithm 2 outlines the greedy pattern enumeration. To construct a pattern, we start with an empty pattern and in each step we try to extend it by an edge such that the coverage is increased by maximum. If no 1-edge extension of the current pattern improves the value of the objective function then we terminate the procedure and start from the empty pattern again. We continue the process for $k$ times. The greedy algorithms that we are going to discuss in this section differ only in step 5 i.e., computing the best extension of the current pattern.

5.4.1 Estimating coverage from the pattern embeddings

At any given stage in the algorithm 2 we have already computed a set $S$ of patterns and $P$ is the pattern constructed so far in the current iteration. We wish to estimate the change in the coverage for all 1 edge extensions of $P$ and pick the extension with maximum increase as the next extension. The fundamental
Algorithm 2: Greedy approach for maximizing the coverage of the edges in the uncertain graph

**Require:** Uncertain graph $G$, number of patterns $k$. 

**Ensure:** $S \subseteq F$, $|S| \leq k$ and $f(S)$ is maximized.

$S \leftarrow \emptyset$

for $i \in [1, k]$ do

$P \leftarrow \emptyset$

while TRUE do

5: $P' \leftarrow \arg\max_{P'} f(S \cup P')$, $P' = P \cup e$

if $f(S \cup P') \leq f(S \cup P)$ then

break

else

$P \leftarrow P'$

end if

10: end if

end while

if $P == \emptyset$ then

break

end if

15: $S \leftarrow S \cup P$

end for

step to compute this extension lies in efficiently estimating the coverage of a set of patterns. The estimation we propose in this section assumes that we have access to the embeddings of the patterns $S \cup P$ in the MCG certain graph $G^\ast$.

Given a set of patterns $S$, graph $G$ and an edge $e \in G$. We first define $\Phi_{S,e,G} = \{\phi_1, \phi_2, \ldots\}$ to be the set of embeddings of patterns in $S$ which are mapped to $e$ in the certain graph $G$. Now, given the uncertain graph $G$, we also define $P[\Phi_{S,e,G^\ast}]$ to be the probability that at least one of these embeddings is present in a certain graph sampled from $G$. It is easy to see that $P[\Phi_{S,e,G^\ast}]$ is exactly equal to $cov(S, e, G)$. Therefore, the coverage $f(S)$ can be written as the summation $\sum_{e \in G} P[\Phi_{S,e,G^\ast}]$.

The change in coverage of $e$ by extending the pattern from $P$ to $P'$ can be approximated as in the equation 5.7, where drop the subscripts $e$ and $G^\ast$ from the expressions of the from $P[\Phi_{P,e,G^\ast}]$. 
\[ \delta \text{cov}(e) = \mathbb{P}[\Phi_{S \cup P^*}] - \mathbb{P}[\Phi_{S \cup P}] \]
\[ = \mathbb{P}[\Phi_S \cup \Phi_{P^*}] - \mathbb{P}[\Phi_S \cup \Phi_P] \]
\[ = \mathbb{P}[\Phi_{P^*}] - \mathbb{P}[\Phi_P] + \mathbb{P}[\Phi_S \cap \Phi_{P^*}] - \mathbb{P}[\Phi_S \cap \Phi_P] \]
\[ \sim \mathbb{P}[\Phi_{P^*}] - \mathbb{P}[\Phi_P] + \frac{\mathbb{P}[\Phi_S] \times (\mathbb{P}[\Phi_P] - \mathbb{P}[\Phi_{P^*}])}{\mathbb{P}[c]} \]
\[ \sim (\mathbb{P}[\Phi_{P^*}] - \mathbb{P}[\Phi_P]) \times (1 - \frac{\mathbb{P}[\Phi_S]}{\mathbb{P}[c]}) \]  

(5.7)

Now to compute the quantity in the equation 5.7 we need a way to estimate the probabilities \( \mathbb{P}[\Phi_{P^*}], \mathbb{P}[\Phi_P] \) and \( \mathbb{P}[\Phi_S] \). The common feature is that all these three probabilities denote the probability that at least one event happens from a union of events. The events here are the embeddings that cover the edge \( e \) in the certain graph \( G^* \).

### 5.4.1.1 Upper and lower bounds for union of events

From the basic probability theory we know that computing the probability of union of \( n \) events requires the intersection probability of every proper subset of \( n \) events. There are \( 2^n \) such proper subsets. Here the individual event corresponds to an isomorphism of the pattern i.e., the edges that are part of the isomorphism. Let \( \Phi_X = \{\phi_1, \phi_2, \ldots, \phi_n\} \) be a subset of \( n \) embeddings of the pattern \( P \) that cover the edge \( e \in G^* \). Then, \( \mathbb{P}[\Phi_X] = \mathbb{P}[\phi_1 \cup \phi_2 \cup \ldots \phi_n] \).

\[ \mathbb{P}[\Phi_X] = \mathbb{P}[\phi_1 \cup \phi_2 \cup \ldots \phi_n] \]
\[ = \sum_i \mathbb{P}[\phi_i] - \left( \sum_{i,j,i \neq j} \mathbb{P}[\phi_i \cap \phi_j] \right) \ldots + (-1)^{n+1} \mathbb{P}[\phi_1 \cap \phi_2 \cap \ldots \phi_n] \]
\[ = \sum_{S \subseteq P(\Phi)} (-1)^{|S|+1} \mathbb{P}[S] \]  

(5.8)
We can see that the expression 5.8 is a summation of $2^n$ terms each of which is a probability of intersection of at most $n$ events. Therefore computing $P[\Phi_P]$ is exponential in the number of embeddings that cover $e$.

Instead of computing the probability exactly we use Hunter-Worsley bounds [73] for approximating the probability of a union of events. These bounds gives us tight bounds for the union in terms of probabilities of lower order intersection terms. The equation 5.9 gives an upperbound of $P[\Phi_X]$ using only probabilities of the form $P[\phi_i \cap \phi_j]$. The value of $P[\phi_i \cap \phi_j]$ is simply the product of probabilities of all edges that are common to the isomorphisms $\phi_i$ and $\phi_j$. In the equation 5.9 $MST$ is a maximum weight spanning tree of the graph in which each isomorphism is a vertex and two vertices are connected by an edge with a weight equal to the intersection probability of the corresponding isomorphisms.

$$P[\Phi_X] \leq \sum_i P[\phi_i] - \sum_{(i,j) \in MST} P[\phi_i \cap \phi_j] \quad (5.9)$$

In summary, the greedy algorithm presented in this section maintains the list of embeddings of the current pattern. It then explores the one neighborhood of the embeddings and computes all possible extensions and their corresponding embeddings. For each extension $P'$, it computes the change in coverage of each edge in the uncertain graph. Finally, it picks the extension for which the overall increase in the coverage is maximum. The exploration procedure stops when no extension leads to an increase in the coverage value.

5.5 Computing the coverage using edge representatives

In the Section 5.4.1 we proposed a greedy approach when we have access to the embeddings of the pattern in the certain graph $G*$. An obvious disadvantage of this method is that the memory requirements are high because of potentially exponential number of embeddings that cover an edge in the graph. We now propose a method that upper bounds the coverage of a pattern in polynomial time and space. It is inspired from the idea of representative set that we Introduced in the Section 4.1.
5.5.1 Edge view of graphs

As we are dealing with probabilities on the edges it is more meaningful to think in terms of the edges rather than the vertices of the graph. An edge view of a graph $G$ is a graph $G^E$ where each vertex represents an edge $e \in G$ and $(a, b), (c, d) \in G$ are connected iff at least one of the four edges $(a, c), (a, d), (b, c), (b, d) \in G$. Figure 5.4 shows a graph (figure 5.4a) and its corresponding edge view (figure 5.4b).

Consider a graph with $n_v$ vertices and a maximum degree of $d$. Then the number of vertices in its edge view graph is upper bounded by $n_v \times d$ and the number of edges by $2 \times n_v \times d^2$. It follows from the fact that the each vertex in the edge view has a degree of at most $2 \times d$.

In the rest of the section we work with edge view of the pattern and the certain graph $G^*$. Note that the current definition of edge view is applicable only for certain and unweighted graphs.

5.5.2 Edge representatives

Given a pattern $P$ and a graph $G$ in their edge view forms. The edge representative of a vertex $u \in P$ is the set $R(u) = \{v|v \in G, \exists \phi, \phi(u) = v\}$. In other words the edge representative of a pattern edge is the set of edges in the graph $G$ in which both ends are mapped via an isomorphism of a vertex in the pattern. We can compute the edge representative sets using the k-hop and NL pruning techniques (see section 4.2) that we used for computing the vertex based representative sets.

To use the k-hop pruning technique we must first define the k-hop label for the edge view graphs and set the maximum label mismatch cost $\alpha = 0$. The k-hop can be trivially extended if we label each vertex in edge view graph by the pair
of labels of its end points. Once we extended the k-hop label we can use all the machinery we have developed in the section 4.2 for efficiently computing the edge representatives. Further, we can show that the optimizations are still valid under the edge view definitions.

5.5.3 Estimating coverage from edge representatives

In this section, we present an approximation of \( \text{cov}(P, e, G) \) without explicitly computing the isomorphisms of the pattern \( P \) in the certain graph \( G^* \).

We know that the quantity \( \text{cov}(P, e, G) \) represents the probability that there exists an isomorphism of the pattern \( P \) in which some edge is mapped to \( e \in G \). Obviously this quantity is less than or equal to the sum of the probabilities that \( x \in P \) is mapped to \( e \in G \) i.e., \( \text{cov}(P, e, G) \leq \sum_{x \in P} \text{cov}(P, x, e, G) \).

5.5.3.1 Notation

Let \( x \) be any edge in the pattern and \( y \in G^* \) be a representative of the edge \( x \) in the certain graph \( G^* \). Also, let \( T(x) \) be a spanning tree of the pattern \( P \) in which \( x \) is the root node. The neighbors of a vertex in a graph is given by \( N(x) \) and the child nodes of vertex \( x' \) in the spanning tree \( T(x) \) is given by \( S(x') \). The restricted set of representatives, \( M(z) \), for the pattern edges \( z \in P \) is given by the equation 5.10.

\[
M(z) = \begin{cases} 
\{y\} & \text{if } z = x \\
R(x) \cap N(y), & \text{otherwise} 
\end{cases} 
\]  

(5.10)

5.5.3.2 Approximate coverage

Using the notation from the section 5.5.3.1, we now give an approximation for the coverage of an edge in the uncertain graph \( G \). For any \( x \in P \) and \( y \in G \), the quantity \( c(x, y) \) is defined recursively as in the equation 5.11 and the approximate value of the coverage is given by the equation 5.12.
\[ c(x, y) = \begin{cases} 
\mathbb{P}[y] \times \prod_{x' \in \text{child}(x)} (1 - \prod_{y' \in M(x')} [1 - c(x', y')]), & \text{if } \text{child}(x) \neq \phi \\
\mathbb{P}[y], & \text{otherwise} 
\end{cases} \] (5.11)

\[ \text{cov}(P, y, \mathbb{G}) \leq \sum_{x \in P} \text{cov}(P, x, y, \mathbb{G}) \]

\[ \cong \sum_{x \in P} c(x, y) \] (5.12)

### 5.5.4 Intuition behind the \( c(x, y) \) approximation

We will now give an intuition behind why \( c(x, y) \) gives a meaningful approximation to the quantity \( \text{cov}(P, y, \mathbb{G}) \). First, \( \sum_{x \in P} \text{cov}(P, x, y, \mathbb{G}) \) is a good approximation for \( \text{cov}(P, y, \mathbb{G}) \) whenever the edge \( y \in \mathbb{G} \) is predominantly mapped to an edge \( x \in P \) with high probability across all isomorphisms of \( P \) that cover \( y \).

Now, let us interpret the recursive expression for the approximation given by the equation 5.11. For approximating \( \text{cov}(P, x, y, \mathbb{G}) \), we construct a spanning of the pattern with \( x \) being the root node. If \( x \) is a leaf node then the only possible mapping of \( x, M(x), \) is the singleton set \( \{y\} \). In such a case \( c(x, y) = \mathbb{P}[y] \). On the other hand when \( x \) is not a leaf node, it is connected to nodes \( x' \in \text{child}(x) \). Then the term \( (1 - \prod_{y' \in M(x')} [1 - c(x', y')]) \) denotes the probability that there exists a neighbor of \( y, y' \in \mathbb{G} \), which represents the pattern neighbor \( x' \). Here we loosely interpret the quantity \( c(x, y) \) as representing the probability that \( y \) represents \( x \). It is a positive quantity between 0 and \( \mathbb{P}[y] \). Therefore, \( c(x, y) \) is the product of \( \mathbb{P}[y] \) and terms of the form \( c(x', y') \) where \( x' \in N(x) \) and \( y' \in N(y) \).

**Example:** Figure 5.5 shows an example of the \( c(x, y) \) approximation we have described in this section. An example pattern in its edge view form is shown in the figure 5.5a. Note that the labels \( \{A, B, C, D\} \in \Sigma \times \Sigma \) as each vertex in the edge view form represents an edge from the original pattern. Similarly, 5.5b shows an example uncertain graph in its corresponding edge view form. Figure 5.5c shows a
spanning tree for the vertex $x = 1$ and 5.5d shows the $c(x, y)$ values $x \in T(x)$ and $y \in M(x)$.

### 5.6 Enumerating high coverage patterns

In the previous sections, we described two methods to estimate the coverage of a given pattern. The goal, however, is to find a set $S$ of patterns such that the objective function 5.4 is maximized. In approximate graph mining algorithms, the
space of all possible patterns forms a partial order. Additionally, we now have a hard
constraint that the number of patterns in the output set $S$ is atmost $k$. Obviously,
computing the value of objective function for every combination of $k$ patterns is
prohibitively expensive. In this section, we propose heuristics for enumerating good
pattern sets of size $k$. First, we describe a procedure for enumerating a single good
pattern. Then we show a method for extracting $k$ good patterns.

5.6.1 Single pattern

Our main idea is to explore the partial order of patterns in a depth first
manner. We start with an empty pattern and repetitively extend it as long the
coverage increases. Of all the single edge patterns, the pattern $P_1$ for which the
expression $\sum_{e \in G} \text{cov}(P_1, e, G)$ given by the equation 5.13 is maximized.

$$P_1 = \arg \max_{l_1, l_2 \in \Sigma} \sum_{e = (u,v), L(u) = l_1, L(v) = l_2} P[e] \quad (5.13)$$

We start a depth first exploration with $P_1$. In each iteration, we estimate the
coverage of all 1-edge extensions of the current pattern. If the best extension has
a higher coverage value, we continue the process with that extension as the current
pattern. Otherwise the process is terminated and the current pattern is returned as
the maximal pattern during this exploration.

5.6.2 Enumerating $k$ Patterns

Imagine what happens if we repeat the procedure for enumerating a single
maximal pattern $k$ times. We can see that each run would end-up with the same
maximal pattern. It is due to the fact that the enumeration algorithm again makes
the same decision i.e., edge extension at each step. To avoid this effect and force
the search procedure to explore other paths in the partial order, we need a way
to mark the edges in $G$ that have been covered (at least partially) already. One
simple way would be to temporarily reduce the probability of an edge by its current
coverage. In other words, the probability of an edge $e \in G$, $P[e]$, is reduced from
$P[e]$ to $P[e] - \text{cov}(S, e, G)$. Note that this quantity is always positive (or zero). In
the next iteration, the pattern enumeration procedure prefers to cover those edges
5.6.3 Updated edge probabilities

To enumerate \( k \) patterns, we start with \( S = \emptyset \), an empty set of maximal patterns. The maximal patterns from each iteration are added to this set. In reality, the set might contain duplicates. Let \( S_i \) be the set of patterns enumerated in the first \( i \) iterations. Also, let \( P_{i+1} \) be the maximal pattern enumerated in the \((i + 1)\)th iteration. For each edge \( e \) in the given uncertain graph \( G \), the probability of its existence after \((i + 1)\)th iteration is given by the equation 5.14:

\[
\mathbb{P}[e]^{i+1} = \begin{cases} 
\{ \max(0, \mathbb{P}[e]^i) - \text{cov}(S_{i+1}, e, G) \} & \text{if } i + 1 > 0 \\
\mathbb{P}[e], & \text{otherwise}
\end{cases}
\]  

(5.14)

The value \( \text{cov}(S_{i+1}, e, G) \) depends on the patterns enumerated in the first \((i+1)\) iterations and it can be computed using the equation 5.2. We instead approximate it by the equation 5.15:

\[
\text{cov}(S_{i+1}, e, G) \leq \sum_{P_j \in S_{i+1}, 1 \leq j \leq i+1} \text{cov}(P_j, e, G)
\]  

(5.15)

The inequality follows from the Boole’s inequality, also known as the union bound of a finite set of events. Here, the maximal patterns correspond to the events. Therefore, we update the edge probabilities using the equation 5.16 instead of the equation 5.14:

\[
\mathbb{P}[e]^{i+1} = \begin{cases} 
\{ \max(0, \mathbb{P}[e]^i) - \sum_{P_j \in S_{i+1}, 1 \leq j \leq i+1} \text{cov}(P_j, e, G) \} & \text{if } i + 1 > 0 \\
\mathbb{P}[e], & \text{otherwise}
\end{cases}
\]  

(5.16)

5.7 Preliminary results

To test the effectiveness of the greedy algorithm 2, we ran experiments on real probabilistic graphs obtained from the STRING database. This database contains information about the protein-protein interactions obtained from various sources.
Table 5.1: Properties of sampled probabilistic graphs

| Dataset | $|V|$ | $|E|$ | $|L|$ | $\sum P[e]$ | $\sigma(P[e])$ |
|---------|------|------|------|-------------|--------------|
| $G_1$   | 1001 | 1500 | 14   | 368.54      | 0.19         |

In addition, the database also provides a probability value for the existence of each interaction. We sampled probabilistic a graph from this database. Table 5.1 shows the number of vertices, edges and the size of vertex label set for this probabilistic graph. The table also shows the sum of probabilities of the edges and the standard deviation of the edge probability values.

5.7.1 Single maximal pattern

Figure 5.6 shows the change in the coverage of the first maximal pattern enumerated from $G_1$ using our greedy method for exploring the partial order. The X-axis show the number of edges in the pattern or in other words depth of the pattern in partial order. The Y-axis shows the coverage estimated using the $c(x,y)$ approximation described in Section 5.5.3.2. The procedure chooses Protein $\rightarrow$ Protein as the best extension of empty pattern. Figure 5.8a shows the maximal pattern after 8 edge extensions.

5.7.2 $k$ maximal patterns

Figure 5.7 shows the sum of coverage of the edges in the uncertain graph $G_1$ as we enumerate 8 maximal patterns. We can see that the first maximal pattern has the highest coverage of all the 8 maximal patterns. It is because the edge probabilities are updated by the equation 5.16 after enumerating each pattern. Moreover, Figures 5.8a and 5.8b show the maximal patterns enumerated during the first two iterations of our greedy algorithm 2. We can see that the patterns differ in both the structure and the node labels.
Figure 5.6: Coverage for a single random walk
Coverage of the optimal pattern set

_Greedy pattern growth algorithm_

![Coverage for the optimal set](image)

*Figure 5.7: Coverage for the optimal set*
Figure 5.8: Maximal patterns from greedy algorithm
CHAPTER 6
Summarizing Frequent Patterns

Frequent pattern mining algorithms often generates many more patterns than a user can analyze. Moreover, subsets of frequent patterns provide similar information to the end user because of minor variation in the structure and labels of the patterns. Further, when we explore the search by random extensions the search is not completely explored and the maximal patterns may be similar to each other. In this chapter, we discuss methods to find a small representative set of patterns that have are highly diverse and covers all the frequent patterns generated by mining algorithms.

6.1 Introduction

Given a set of non-isomorphic maximal patterns $M$, we cluster them into groups of similar patterns, and then selects a representative set of infrastructure patterns from each cluster. There are three main steps:

- **Similarity** In section 6.2 we propose a method to compute similarity between any pair of patterns. The metric we propose takes into account both the structural and label similarities between the patterns.

- **Clustering** Once the pair-wise similarities are computed, we cluster the patterns into groups such that patterns in the same group are similar to each other. In Section 6.3 we propose a graph clustering based approach to find clusters of similar patterns.

* Portions of this chapter previously appeared as


• **Representative patterns** From each cluster we extract a small set of patterns that covers all the patterns belonging to the same cluster. These patterns are called the representatives of the corresponding cluster. Representative patterns are of high interest to the end user. In Section 6.4 we propose a greedy approach to compute representative patterns.

Computing similarity between the patterns is the most important step among all the three steps. In section 6.5 we compare our method with other existing approaches.

### 6.2 Pattern similarity

Before clustering the maximal patterns, we have to define a similarity measure between patterns, that takes into account both the structure and label information. Graph edit distance based methods [74] are a popular approach to compute the similarity, however, the vast majority of these methods focus mainly on the structure. For example, a purely structure based method would consider $P_3$ and $P_4$ in Fig. 6.1 to be highly similar. Methods that consider labels include [75, 76]. We propose a novel pattern similarity approach based on diffusion kernels [77], which works well for CMDB graphs. As such the clustering method is independent of the similarity measure, and thus any of the attributed graph similarity measures can also be used. For instance, we compare our method with the similarity flooding method [78] in the experimental section.

We define similarity between two patterns $P = (V_P, E_P, L_P)$ and $Q = (V_Q, E_Q, L_Q)$, as

$$ Sim(P, Q) = Jaccard(P, Q) \times Diffusion(P, Q) $$

(6.1)

Here $Jaccard(P, Q) = \frac{|L_P \cap L_Q|}{|L_P \cup L_Q|}$ is the Jaccard coefficient between the label sets for $P$ and $Q$. The more the labels in common, the higher the Jaccard similarity. $Diffusion(P, Q)$ is the diffusion kernel based similarity between $P$ and $Q$ that considers both the structure and the label information, as described below.

Following a procedure similar to that in [79], given $P$ and $Q$, we first create
Figure 6.1: Sample Maximal Patterns

an augmented weighted graph $R = (V_R, E_R, W_R)$. Here $V_R = V_P \cup V_Q \cup \{l \mid \exists v \in V_P, L_P(v) = l\} \cup \{l \mid \exists v \in V_Q, L_Q(v) = l\}$, i.e., $R$ contains both structural nodes (those in $P$ and $Q$) and attribute nodes (labels for nodes in $P$ and $Q$). $E_R = E_P \cup E_Q \cup \{(v, l) : v \in V_P, L_P(v) = l\} \cup \{(v, l) : v \in V_Q, L_Q(v) = l\}$. In other words, $E_R$ contains both the structural edges (the original edges between vertices in both $P$ and $Q$), as well as the attribute edges (between a node in $P$ and $Q$, and its label). Finally, $W_R : E_R \rightarrow \mathbb{R}$ is a function that assigns a weight to each edge. The weights on structural edges are set to 1, i.e., $W(u, v) = 1.0$ for all $(u, v) \in E_P \cup E_Q$. The weights on attribute edges are set as follows: $W(v, l) = \frac{1}{n_l}$, where $n_l$ is the number of neighbors of node $l$ in $R$. In the augmented graph, two structural vertices that have the same label $l$, are both neighbors of the attribute node $l$. To avoid inflating the similarity purely due to labels (which has already been accounted for by $\text{Jaccard}(P, Q)$), we assign the fractional weight on attribute edges. Fig. 6.2a shows the augmented weighted graph for $P_2$ and $P_3$ from Fig. 6.1.

To compute $\text{Diffusion}(P, Q)$ for each pair of patterns, we use the diffusion kernel approach [77] over their augmented graph. A diffusion kernel mimics the physical process of diffusion where heat, gases, etc., originating from a point diffuse with time. On graphs, it is the local similarity that diffuses via continuous time random walks (i.e., with an infinite number of infinitesimally small steps). Given the augmented graph $R = (V_R, E_R, W_R)$, the matrix $W_R$ is taken to be the weighted adjacency matrix of $R$. Further, define the diagonal degree matrix as $D(v_i, v_i) = \sum_{v_j} W_R(v_i, v_j)$, and $D(v_i, v_j) = 0$ when $i \neq j$. The Laplacian matrix of $R$ is then defined as: $N = D - W_R$. Finally, the diffusion kernel matrix is defined as $K = \ldots$
$e^{\beta L} = \sum_{k=0}^{\infty} \frac{\beta^k}{k!} L^k$, where $\beta$ is a real-valued diffusion parameter, and $e^{\beta L}$ is the matrix exponential (with $L^0 = I$ and $0! = 1$). Since $L$ is positive semi-definite, it has $|V_R| = n$ real and positive eigenvalues $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_n \geq 0$. Let $u_i$ be the eigenvector corresponding to eigenvalue $\lambda_i$. Then the diffusion kernel can easily be computed as the spectral sum [77]: $K = \sum_{i=1}^{n} u_i e^{\beta \lambda_i} u_i^T$. The eigenvalues and eigenvectors of $K$ can be computed in $O(n^3)$ time, where $n = |V_R|$.

The kernel matrix entry $K(v_i, v_j)$ gives the diffusion-based similarity between any two vertices in the augmented graph $R$ for patterns $P$ and $Q$. In particular, we are interested in those entries $K(u, v)$ where $u \in V_P$ and $v \in V_Q$. We define the diffusion similarity between $P$ and $Q$ as follows: If $L_P \cap L_Q = 0$, then we set
$Diffusion(P, Q) = 0,$ otherwise

$$Diffusion(P, Q) = \min_{l \in L_P \cap L_Q} \left\{ \max_{u \in V_P, v \in V_Q} \left( \frac{K(u, v)}{(u, l), (v, l) \in E_R} \right) \right\}$$

In other words, the diffusion similarity between $P$ and $Q$ is defined as the least label similarity over all labels $l$, such that the label similarity is the maximum kernel similarity over pairs of nodes $u, v$ that share a given label $l$. Fig. 6.2b shows the pairwise similarities between all the patterns in Fig. 6.1, based on Eq. (6.1), that combines both the $Jaccard()$ and $Diffusion()$ values.

### 6.3 Clustering

We employ graph clustering to cluster the set of maximal patterns $M$. In particular, given the similarity matrix $S(i, j) = Sim(P_i, P_j)$ between any two patterns $\in M$, we can think of $S$ as the weighted adjacency matrix of a similarity graph, where each maximal pattern is a node, and any two maximal patterns are linked with weight $S(i, j)$. Clustering of the patterns is then equivalent to clustering the nodes in the similarity graph. While many algorithms have been proposed for graph clustering [79], we use the Markov clustering (MCL) [80] approach as opposed to spectral methods [81], since MCL does not require the number of clusters as input.

Let $D$ be the diagonal degree matrix corresponding to the weighted similarity matrix $S$. Let $N = D^{-1}S$ be the normalized adjacency matrix for the similarity graph. The matrix $N$ is a row-stochastic or Markov matrix that specifies the probability of jumping from node $P_i$ to any other node $P_j$. $N$ is thus that transition matrix for a Markov random walk on the similarity graph. As such, the $k$-th power of $N$, namely $N^k$, specifies the probability of transitioning from $P_i$ to $P_j$ in a walk of $k$ steps. MCL [80] takes successive powers of $N$ to expand the influence of a node. However, it damps the extent of a nodes’ influence, by an inflation step, whose goal is to enhance higher and diminish lower transition probabilities. Given transition matrix $N$, define the inflation operator $\Upsilon$, given as follows: $$\Upsilon(N, r) = \left\{ \frac{N(i, j)^r}{\sum_{a=1}^{N(t, a)^r} i, j=1} \right\}$$

In essence, $\Upsilon$ takes each element of $N$ to the $r$-th power, and then re-normalizes the rows to make the matrix row-stochastic.
Given the initial $N$ matrix, and an inflation parameter $r$, MCL is an iterative matrix algorithm consisting of two main steps: i) expansion: $N = N^2$, followed by ii) inflation: $N = \Upsilon(N, r)$. The method converges to a doubly idempotent matrix, and the strongly connected components in the corresponding induced graph yield the final node clusters [80]. The only parameter in MCL is the inflation value $r$ that controls the granularity. Higher values lead to more, smaller clusters, whereas smaller values lead to fewer, larger clusters. MCL runs in $O(tn^3)$ time, where $|M| = n$, and $t$ is the number of iterations until convergence.

6.4 Extracting representatives from clusters

Given a set of clusters $C_i$, $1 \leq i \leq k$ obtained via the MCL approach and a similarity threshold $\theta$, from each cluster $C_i$ we aim to extract as subset of the patterns $R_i \subseteq C_i$, such that for each $P_j \in C_i$, there exists a pattern $P \in R_i$ with $\text{Sim}(P_j, P) \geq \theta$. The task is to find a minimal set of representative patterns for each cluster. However, this problem is equivalent to smallest set cover, an NP-Complete problem, which nevertheless has a greedy $\Theta(\log n)$ approximation algorithm [82]. The greedy heuristic iteratively chooses the pattern that covers or represents the largest number of remaining elements in a cluster, until all the cluster members are covered.

6.5 Pattern similarity

We now show the effectiveness of the our new similarity metric $\text{Sim}(P, Q)$ defined in (6.1), by comparing it with the simflood algorithm [78], which is a widely used technique for graph matching. Given two patterns, $P$ and $Q$, simflood first computes a match score between nodes in $P$ and $Q$ based on the string matching of the labels. The match score of a pair of labels depends on the prefixes and suffixes of the labels. This matching criterion is not meaningful in the case of maximal patterns extracted from CMDB graphs because all the labels are defined in the same context, and every pair of labels is either the same or different. Hence, we used a 0/1 score for the initial match. In the second step, simflood constructs an induced propagation graph in which each node is a pair of matchable nodes obtained
Table 6.1: Similarity Flooding Variations: $\sigma^i$ is the similarity between a pair of nodes in the $i$-th iteration, and $\varphi$ is the weighted sum of the similarities of all the predecessors of a given node pair.

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Fixpoint formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic</td>
<td>$\sigma^{i+1} = \text{normalize}(\sigma^i + \varphi(\sigma^i))$</td>
</tr>
<tr>
<td>A</td>
<td>$\sigma^{i+1} = \text{normalize}(\sigma^0 + \varphi(\sigma^i))$</td>
</tr>
<tr>
<td>B</td>
<td>$\sigma^{i+1} = \text{normalize}(\varphi(\sigma^i + \sigma^i))$</td>
</tr>
<tr>
<td>C</td>
<td>$\sigma^{i+1} = \text{normalize}(\sigma^i + \sigma^i + \varphi(\sigma^i + \sigma^i))$</td>
</tr>
</tbody>
</table>

from $P$ and $Q$. The number of nodes in the induced propagation graph can be as many as $|E_P| \times |E_Q|$ depending on the edge labels. In contrast, the order of the Augmented Weighted Graph constructed by our algorithm is equal to the sum of the number of nodes and labels present in both the patterns. Though the payoff by using Augmented Graph is marginal when the task is just to compare a single pair of patterns, there is a significant improvement in the overall runtime when similarity between every pair of patterns is required. In simflood, the similarity between nodes is computed by propagating the initial match scores to the neighbors in a manner similar to the pagerank algorithm [83]. Simflood uses four different formulas (shown in Table 6.1) for computing the match score in any iteration using the score in previous iteration and the initial match score. Our algorithm on the other hand propagates the similarity using the idea of diffusion. To make a fair comparison between the algorithms, similarity score between $P$ and $Q$ is computed similar to Diffusion($P,Q$), except that the similarity between nodes $v_i$ and $v_j$ is the fixed point value of the node pair $(k_i,k_j)$ in the induced propagation graph instead of the kernel matrix entry $K(v_i,v_j)$.

6.5.1 Comparing similarity metric

We compare our $\text{Sim}(P,Q)$ based kernel, with the $\text{simflood}(P,Q)$ kernel, by comparing the quality of the clusters the MCL algorithm generates using these kernels. The quality of a clustering of maximal patterns is measured using coverage and conductance [84], which are frequently used in graph clustering literature. The value of the minimum cut of a cluster is a measure of its quality. A low value means that the cluster contains large number of dissimilar pairs of nodes, which in turn
implies that the cluster is of bad quality. Given a clustering \( C = \{C_1, C_2, \cdots, C_k\} \) of the graphs into \( k \) clusters, the conductance \( Con(C_i) \) of a cluster \( C_i \) is a generalization of the minimum cut that considers the size of the cut and also the similarity between a node and its neighbor. Given a set of clusters, the conductance of the clustering is defined as the minimum of the conductance of all clusters, and the inter-cluster conductance \( Con(C) \) [85] can be defined as the complement of the maximum conductance over all the clusters.

\[
Con(C_x) = \sum_{i \in C_x, j \notin C_x} \frac{a_{ij}}{\min \{a(C_x), a(V \setminus C_x)\}}
\]

\[
Con(C) = 1 - \max_{i \in 1,\ldots,k} \phi(C_x)
\]  

(6.2)

where \( a_{ij} \) is the similarity between the vertices \( i \) and \( j \) in the graph and \( a(C_x) \) is the sum of similarities between all pairs of vertices with at least one end in cluster \( C_x \).

The other quality measure we use is the coverage (\( Cov \)), defined as the ratio of sum of the similarities between vertices in the same cluster and sum of similarities between all pairs of vertices in the graph. A clustering with a high value of coverage is preferred over a clustering with low coverage.

Table 6.2 and 6.3 compare our similarity metric with simflood in terms of time, the number of resulting clusters, cluster coverage, conductance, and the number of representative patterns extracted for different values of inflation parameter used in the MCL algorithm. In the tables, Basic, A, B, and C refer to the different update formulas of simflood, whereas Diffusion refers to our similarity method.

It is evident from the table that computing the similarity using our approach is significantly faster compared to the simflood method. We achieve this efficiency without sacrificing the quality of the resulting clusters. With inflation \( I = 2 \) (Table 6.2) the coverage of the clustering obtained using our similarity metric is very close to the maximum possible value 1. This means that the similarity between the patterns in the same cluster is very high compared to the similarity between patterns in different clusters. With inflation \( I = 4 \) (Table 6.3) our coverage is less than the coverage obtained using simflood. However, note that our method finds more
Table 6.2: $I = 2$: Time for computing the pair similarity between maximal patterns using Diffusion Kernel approach and variations of the similarity flooding.

<table>
<thead>
<tr>
<th>Method</th>
<th>Time(s)</th>
<th>Cov\textsuperscript{1}</th>
<th>Con\textsuperscript{2}</th>
<th>Clusters</th>
<th>#Repr</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic</td>
<td>148.82</td>
<td>0.97</td>
<td>0.29</td>
<td>2</td>
<td>13</td>
</tr>
<tr>
<td>A</td>
<td>240.09</td>
<td>0.99</td>
<td>0.28</td>
<td>2</td>
<td>25</td>
</tr>
<tr>
<td>B</td>
<td>45.47</td>
<td>0.97</td>
<td>0.33</td>
<td>2</td>
<td>20</td>
</tr>
<tr>
<td>C</td>
<td>36.84</td>
<td>0.99</td>
<td>0.28</td>
<td>2</td>
<td>25</td>
</tr>
<tr>
<td>Diffusion</td>
<td>9.68</td>
<td>0.98</td>
<td>0.33</td>
<td>2</td>
<td>24</td>
</tr>
</tbody>
</table>

\textsuperscript{1} Cov: Coverage of the clustering.
\textsuperscript{2} Con: External Conductance of the clustering.

Table 6.3: $I = 4$: Time for computing the pair similarity between maximal patterns using Diffusion Kernel approach and variations of the similarity flooding.

<table>
<thead>
<tr>
<th>Method</th>
<th>Time(s)</th>
<th>Cov\textsuperscript{1}</th>
<th>Con\textsuperscript{2}</th>
<th>Clusters</th>
<th>#Repr</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic</td>
<td>148.82</td>
<td>0.89</td>
<td>0.17</td>
<td>4</td>
<td>16</td>
</tr>
<tr>
<td>A</td>
<td>240.09</td>
<td>0.99</td>
<td>0.28</td>
<td>2</td>
<td>25</td>
</tr>
<tr>
<td>B</td>
<td>45.47</td>
<td>0.94</td>
<td>0.31</td>
<td>3</td>
<td>19</td>
</tr>
<tr>
<td>C</td>
<td>36.84</td>
<td>0.99</td>
<td>0.33</td>
<td>2</td>
<td>26</td>
</tr>
<tr>
<td>Diffusion</td>
<td>9.68</td>
<td>0.86</td>
<td>0.21</td>
<td>5</td>
<td>26</td>
</tr>
</tbody>
</table>

\textsuperscript{1} Cov: Coverage of the clustering.
\textsuperscript{2} Con: External Conductance of the clustering.

clusters, and coverage is biased towards fewer clusters, which is the main reason for lesser coverage value. In terms of the conductance of the clustering, our similarity metric gives the best value when the value of inflation is $I = 2$, when all methods find the same number of clusters. Ideally one should compare the coverage and the conductance values when the number of resulting clusters is same using both the methods. We tried MCL with a high values for inflation so that the number of clusters returned using simflood is 5. Even with an inflation of 20, the number of clusters generated using method $C$ remained at 2, whereas our algorithm generated 9 clusters. This shows that \textit{Diffusion} is better compared to simflood at discriminating the maximal patterns. The tables also show the number of representative patterns extracted with a similarity threshold value of $\theta = 0.8$. It can be seen that the number of representatives extracted using our similarity metric is close to the number of representatives extracted using simflood with maximum coverage, and when $I = 4$ it achieves that with a higher number of clusters.
CHAPTER 7
Future Work

The chapter concludes the thesis with a discussion of potential future research areas in the domain of graph mining. We grouped the future work into four domains 1. Applications 2. Algorithms 3. Scalability 4. Evaluation. All these four aspects are equally important in analyzing any real-world linked data.

7.1 Applications

In this thesis, our main focus is on models and algorithms for mining approximate patterns from graph databases. We showed how our methods are very effective at extracting interesting patterns from Biological and IT networks. Our methods find a lot of use in other domains as well. In this section, we present two contrasting applications of approximate graph mining.

7.1.1 Anti-money laundering

With the advent of digital currency and the global use of Internet, financial institutions are able to transact with each other huge amounts of money in a short span. It is a huge challenge to analyze these networks manually for possible fraud. Therefore, we need automatic methods that help authorities in detecting and preventing fraudulent transactions.

Often, the transactions are inter-linked with each other. Fraud is evident only when the transactions are placed in the context of other transactions. Therefore, the transactions cannot be looked at in isolation to detect fraud. Graphs are ideal for modeling the long range interactions between two financial parties. Consider an uncertain graph, where the edges represent transactions between two financial institutions. Probability of each edge represents the chance that the corresponding transaction is legitimate. We can obtain a set of candidate fraud transactions by running a coverage based pattern mining algorithm on this uncertain transaction graph. The key idea stems from the fact that a pattern with high coverage value
Toy example: Table 7.1a shows transactions occurring among three institutions $A$, $B$, and $C$ over a period of 3 weeks. Each transaction between these institutions is marked as either high, medium or low based on our initial estimation of the degree of legitimacy of the transaction. One way to compute it is by considering the average transaction amount between the two parties that are involved. We will construct a graph out of the transactions happening during each week. Figures 7.1b - 7.1d shows the transaction subgraphs for each of the 4 weeks. In each of the graphs, thicker edges represent more legitimate transactions.

With $k = 2$, our greedy algorithm 2 returns $S = \{P_1, P_2\}$ where $P_1 = B - A - C$ and $P_2 =$ three clique $A - B - C$. In practice, we use a probability measure for the edges instead of the three class classification. For each edge in the transaction graphs, we compute the difference between its probability and its coverage by the pattern set $S$ as given by the equation 5.2. Edges with higher difference are candidates for fraudulent transactions. The intuition here is that the patterns with...
high coverage value correspond to a subnetwork of institutions that are engaged in similar types (in terms of the amount) of transactions over a long period of time. This heuristic will flag transaction 2 instead of transaction 4 as fraud even though the initial confidence in the legitimacy of transactions is the same.

### 7.1.2 Analyzing infrastructure networks

We are seeing huge drops in the costs of hardware and this is leading towards a massive increase in the number of devices a person is using. It is especially true of modern smart cities. Inter and intra relationships between people and devices can be analyzed by using graphs. Sophisticated graph analytics will give us better understanding of these relationships. Methods and models that we proposed in this thesis are not optimized to handle such complex networks.

### 7.2 Meta-heuristic methods for graph isomorphism

Meta-heuristic algorithms are those methods that try to find good solutions to combinatorial problems by orchestrating an interaction among local heuristic methods. These methods try to escape from local optima and do a more robust search of the solution space. Nature-inspired meta-heuristic algorithms are a subset of methods where multiple interacting agents work together to find an optimal solution to the combinatorial problem [86]. The behavior of these agents is inspired from biological agents such as birds, ants, and fireflies. However, meta-heuristic algorithms are seldom applied for graph problems. A notable exception is the work by Sammoud et. al [87]. The authors have proposed an Ant Colony Optimization solution for computing similarity between two graphs. Finding an appropriate subgraph isomorphism between a pattern and the database graph is the computationally most intensive step in all of the approximation algorithms that we proposed in this thesis. The problem is even harder when we relaxed the constraints for subgraph isomorphism as in Chapters 4 and in Section 7.5. These sub-steps can be solved efficiently by borrowing some ideas from nature-based meta-heuristic algorithms.
7.3 Systems

Apart from novel heuristic methods, graph mining methods can also be scaled by parallel algorithms. There is not much work in the space of parallel frequent subgraph mining. Kessl et. al [88] recently proposed a method for parallel graph mining on GPUs. Their method shows a speedup of 8 for some of protein and chemical graph datasets. Speedup drops to 4 on citation and random networks. It is due to unequal workload during different iterations of the algorithm. Such a difference in workload is quite often observed when parallelizing graph algorithms. This effect can be attributed to the structure and label distribution of the input graphs. More recently, McLaughlin et. al [89] showed load imbalance in computing betweenness centrality on various real-world networks. The authors proposed a hybrid method that scales better compared to a graph agnostic load division. Some of load balancing techniques can also be used when parallelizing frequent subgraph mining. Better scalability can also be obtained by following hybrid programming paradigm where various steps of the mining algorithm are parallelized via the best suited methods: MPI, OpenMP or GPU programming.

7.4 Evaluation

To use graph mining methods for analyzing real-world graphs, we often need to bring domain experts into the analysis loop. It is important for two reasons. First, large scale graph mining algorithms are usually heuristic based methods. Moreover, the ground truth about the patterns is unknown. Experts will help us evaluate the efficacy of the methods in uncovering interesting patterns from the underlying database graph. Second, the domain knowledge and the feedback from the experts will also help in developing more efficient heuristics for the mining algorithm. Bringing experts into the loop is possible only when they are presented with efficient tools for interacting and querying the graph databases. Another possible direction for the future is to develop better graph visualization and querying tools.
7.5 Mining approximate patterns with edge mismatches

Finally, we propose AMiner, an algorithm to mine frequent subgraph patterns with bounded label-structural mismatches. The label mismatch mining algorithm in chapter 4, which we refer to as LMiner in this section, is applied on the connected subgraphs of a candidate pattern. We use the property that the isomorphisms of $P$ under $\phi_{\alpha,\beta}$ conditions is equal to the union of isomorphisms of $P'$, $P'$ is connected and $|E_P - E_{P'}| \leq \beta$, under $\phi_{\alpha,0}$ conditions. AMiner explores all feasible subgraphs until either the candidate is frequent or all the subgraphs have been explored. We also argue why a derived label technique used to mine $\phi_{\alpha,0}$ isomorphisms is not feasible in mining $\phi_{\alpha,\beta}$ isomorphisms.

7.5.1 Preliminaries

Refer to the definition of approximate subgraph isomorphism in section 1.4.2

Mapping indicator and support For each pair $(u \in V_P, v \in V_G)$, we define a function $I(P, G): (V_P, V_G, \{\alpha\}, \{\beta\}) \rightarrow \{0, 1\}$. We ignore the subscript $(P, G)$ when the pattern and database is evident. The value of $I(P, u, v, \alpha, \beta) = 1$ if $\exists \phi_{\alpha,\beta}, \phi_{\alpha,\beta}(u) = v$ and 0 otherwise. That is to say, the indicator function is 1 if there exists at least one isomorphism in which the pattern vertex $u$ is mapped to $v$ in the database. The representative set of a pattern vertex $u$, $R(u)$, is the set $\{v|v \in V_G, I(P, u, v, \alpha, \beta) = 1\}$ i.e., the set of all possible mappings for the vertex $u$.

Support of pattern is a measure of its frequency of occurrence in the database. One way to define the support is using the following formula:

$$\sup P = \min_{u \in V_P} \sum_{v \in V_G} I(P, u, v, \alpha, \beta)$$  \hspace{1cm} (7.1)

7.5.2 Mining label mismatch isomorphisms

The algorithm, LMiner, described in the chapter 4 can compute frequent approximate patterns from $G$ only when $\beta = 0$. Put another way, LMiner only counts the isomorphisms of a pattern $P$ with exact structure and bounded label costs. The core of LMiner is computing the indicator function $I : (V_P, V_G, \{\alpha\}, \{0\}) \rightarrow \{0, 1\}$. The value $I(P, u, v, \alpha, 0)$ is 1 if $\exists \phi_{\alpha,0}$ isomorphism of $P$ in which $u$ is mapped to $v$
i.e., $\phi_{\alpha,0}(u) = v$. In other words, $v \in R(u)$ if and only if $I(P, u, v, \alpha, 0) = 1$. Once the value of indicator function is computed for all pairs $(u, v) \in V_P \times V_G$, the support can be easily computed using the formula 7.1 and the candidate pattern will be reported frequent if its support is at least $\theta$.

Consider a $u \in V_P$ and $v \in V_G$, we can immediately conclude that $v \notin R(u)$ if $C[L(u)][L(v)] > \alpha$ because the label mismatch cost $C(\phi_{\alpha,0})$ of any $\phi_{\alpha,0}$ isomorphism is at least more than $\alpha$. Alternatively, the indicator function $I(P, u, v, \alpha, 0) = 0$ if $C[L(u)][L(v)] > \alpha$. LMiner algorithm generalizes this idea by using two derived labels for each node. These labels are the so called k-hop, NL labels.

**k-hop label** k-hop label of a vertex $u$ is the set of the vertices that can be reached via k length paths and is denoted by $h_k(u)$. The cost of matching two k-hop labels is the minimum cost of mapping the vertices in k-hop label, the cost of mapping two vertices is equal to the cost of matching their corresponding labels in the graph. Since, the structure of the pattern remains intact under the $\phi_{\alpha,0}$ isomorphism, it can be shown that the k-hop label match cost should be at most $\alpha$. Therefore, for all $v \in R(u)$, $h_k(u) \preceq h_k(v)$. Consider the example in figure 7.2. The cost of matching labels $h_3(1) = \{2, 3\}$ and $h_3(30) = \{10, 20, 40, 60\}$ is equal to 6. The k-hop labels of the database vertices are the same irrespective of the pattern. Therefore, they can be precomputed.

**NL label** NL label is composed of k-hop labels of a vertex and its neighbors.
The NL labels of $u$ and $v$ can be matched only if there exists an injective mapping between the k-hop labels of neighbors of $u$ and $v$. This label has the same effect as that of propagating the k-hop labels in the graph.

The derived labels are basically polynomial time algorithms that can guarantee $I(P, u, v, \alpha, 0) = 0$ for some pairs $(u, v)$, $u \in P, v \in G$. As the subgraph isomorphism problem is an NP-Complete problem, there is no polynomial time algorithm that can compute the exact value of indicator function for all the $(u, v)$ pairs. Therefore, for the remaining pairs LMiner performs an exhaustive enumeration to compute the value of $I(P, u, v, \alpha, 0)$.

In the next section, we present our algorithm that computes frequent patterns under the general definition of isomorphism, $\phi_{\alpha, \beta}$ ($\beta > 0$).

### 7.5.3 Algorithm to mine label and structure-based approximate patterns

Our algorithm, AMiner, for mining frequent patterns with label mismatch threshold $= \alpha$ and edge mismatch threshold $= \beta$ is based on the LMiner algorithm described in the section 7.5.2. Similar to LMiner, core of the AMiner algorithm is computing the value of indicator function $I(P, u, v, \alpha, \beta)$, $\beta \geq 0$. As we have seen already, the parameter $\beta$ is the number of edge mismatches that are allowed in an isomorphism from a pattern into the database. For instance, when $\beta = 1$ checking whether indicator function $I(P, u, v, \alpha, 1) = 1$ is equivalent to asking the question *Is $I(P, u, v, \alpha, 0) = 1$ or can we delete an edge $e$ in the pattern $P$ such that $I(P^*, u, v, \alpha, 0) = 1$ for the pattern $P^* = P - e$, $P^*$ is connected?* Therefore, $I(P, u, v, \alpha, \beta)$ can be written as follows:

$$I(P, u, v, \alpha, \beta) = \begin{cases} 1 & \text{if } I(P, u, v, \alpha, 0) = 1 \\ 1 & \text{if } \beta \geq 1 \text{ and } \sum_{e, P^* = P - e} I(P^*, u, v, \alpha, \beta - 1) \geq 1 \\ 0 & \text{otherwise} \end{cases}$$

where the summation is over all edges $e$ that don’t disconnect the pattern. In other words, $e$ is a cycle edge.

The value of indicator function with $\beta$ edge mismatch threshold is defined
recursively in terms of the function with $\beta - 1$ edge mismatches. The candidate edges $e$ that can be deleted from the pattern are the cycle edges. Note that the function value is not unique for the pair $P, G$. It depends on the specific pair of vertices $u, v$. For example, consider the database graph 7.2 and a triangle pattern with vertices 0, 1, 2 and labels $A, B, C$ respectively. For $I(P, 0, 20, \alpha, 1) = 1$, the edge between 1 and 2 has to be deleted. But $I(P, 1, 30, \alpha, 1) = 1$ if the edge $(0, 2)$ is deleted from the pattern. Therefore, we cannot simply run the LMiner algorithm on all connected subgraphs of $P$ with at most $\beta$ edges missing and report that $P$ is frequent if any of the subgraphs is frequent in the $\phi_{\alpha,0}$ sense.

**Summary** The key observation is that $I(P, u, v, \alpha, \beta) = 1$ if and only if $\exists P^*$ such that $I(P^*, u, v, \alpha, 0) = 1$ and $P^*$ satisfies: i) $P^* \subseteq P$ ii) $|E_P - E_{P^*}| \leq \beta$ iii) $V_P = V_{P^*}$. AMiner explores all feasible edge deletions in $P$ until it find at least $\theta$ vertices with $I(P, u, v, \alpha, \beta) = 1$ for each $u$ in the pattern.

**Search Space** The space of possible edge deletions can be expressed as a layered lattice of height at most $\beta$. The candidate pattern $P$ is at the root and each node is a connected subgraph of $P$. The edge between nodes $P_x$ and $P_y$ represents the difference $E_{P_x} - E_{P_y}$ i.e., the edge(s) removed from $P_x$ to obtain $P_y$. Figure 7.3 shows an example. At depth 0 the edges $e_1, e_2, e_3$ and $e_4$ are the cycle edges and can be deleted without disconnecting the pattern. Also, we assign a random order to the edges. An edge is deleted only if it is greater than all the other edges deleted already. In the example, the order of edges is $e_1 < e_2 < e_3 < e_4$. Therefore, $e_1$ cannot be deleted from the node $P_z$.

![Figure 7.3: Search space of edge deletions](image)

**Organization** We begin by explaining what happens at each node in the
search space (section 7.5.3.1). Then we show how automorphisms in the lattice are detected (section 7.5.3.3). Finally, we show how AMiner explores all non-automorphic connected subgraphs of a candidate pattern (7.5.3.4).

7.5.3.1 Processing at a node

Assume that we are at a node \( P^* \) in the search space. For each node \( u \in P \), we maintain two disjoint subsets of \( V_G \) denoted by \( T_c(u) \) and \( T_v(v) \). \( T_c(u) \) is a set of candidate vertices \( v \) for which \( I(P^*, u, v, \alpha, d) \), \( d \) is the depth of subgraph \( P^* \) in the lattice, is possibly 1 and \( T_v(v) \) is the set of vertices for which \( I(P, u, v, \alpha, \beta) \) is verified to be 1. We run the LMiner algorithm as if \( P^* \) is the candidate pattern, \( T_c(u) \) as its candidate representative vertices. It returns the vertices for which \( I(P^*, u, v, \alpha, d) \) is verified to be 1 which are added to the set \( T_v(u) \). However, the vertices with \( I(P^*, u, v, \alpha, d) = 0 \) are not removed from the set \( T_c(u) \) because the value of indicator function can be 1 via different subgraph.

Unlike the LMiner algorithm that computes the representative sets by pruning from a candidate set, AMiner builds the representative set of a vertex by explicitly enumerating isomorphisms. A vertex can be deleted from the set \( T_c(u) \) only when none of the edge deletions lead to a valid isomorphism. Since, our goal is just to check whether a candidate pattern is frequent we stop the exploration process when the pattern is frequent using the current state of \( T_v(u) \). In this section, we use the example in 7.4 as a running example to explain the various steps of AMiner.

Running Example: Figure 7.4a, 7.4b shows an example database and the associated cost matrix. Part of the search space lattice is shown in the figure 7.4c and the candidate pattern under consideration is at the root. Each of the directed edges represents the pattern edge that is deleted. Suppose, we are processing the subgraph \( P_y \) , \( \alpha = 3 \). The LMiner algorithm finds the isomorphism \( \{1 : 10, 2 : 20, 3 : 40, 4 : 40, 5 : 50, 6 : 70\} \). Therefore, 30 is added to the set \( T_v(3) \). However, it returns that 3 cannot be mapped to 60 as it violates the k-hop label constraint. To be specific, \( h_{30}(2) \) is 1, 2, 4, 5, 6 whereas \( h_{60}(2) \) is 30, 40, 70, 90. It is easy to see that \( h_{30}(2) \not\preceq h_{60}(2) \). The processing at a node can be summarized in terms of VerifyCandidates function as below:
VerifyCandidates : \((P^*, T_c(u)) \rightarrow \text{Set } T_v(u) \text{ such that } I(P^*, u, v, \alpha, 0) = 1 \forall v \in T_v(u)\)

### 7.5.3.2 Search space exploration

The space of possible edge deletions can be exponential in the worst case. Suppose \(P\) is a K-clique candidate pattern with each node having a unique label. The database \(G\) is a connected subgraph of \(P\) with \(d(< k)\) edges missing. Let \(\alpha = 0\), \(\beta = d\) and boolean cost matrix \(B, B[i][j] = 0\) iff \(i = j\) else 1. We show that in the worst case the number of non-isomorphic subgraphs can be exponential in the parameter \(\beta\).

**Property 1.** For any two nodes \(P_x\) and \(P_y\) in the lattice, \(P_x\) is not \(\phi_{0,0}\) isomorphic to \(P_y\).

*Proof.* From the way lattice is constructed there is at least one edge \((u, u') \in P_x\) such that no edge in \(P_y\) has the labels \(L(u)\) and \(L(u')\) on its end points. □

**Property 2.** Only nodes at depth \(d\) can have a \(\phi_{0,d}\) isomorphism in the database \(G\).

*Proof.* Recall that \(\alpha = 0\) and \(\beta = d\). All nodes at depth less than \(d\) have at least one edge more than the number of edges in each connected component of the database. Therefore, no isomorphism can exist as we don’t allow structural mismatches during the search process. □

**Property 3.** Number of leaf nodes is exponential.

*Proof.* Each combination of \(d\) edge deletions leads to a unique pattern. Therefore, the number leaf nodes is exponential in \(d\). □

Based on the above properties, we can conclude that the lattice is exponential in the parameter \(\beta\). In practice, the algorithm usually finds enough verified representatives before exhausting the search space to conclude that a candidate pattern is frequent.
7.5.3.3 Detecting $\phi_{\alpha,0}$ automorphic nodes in lattice

Nodes in the same layer of the lattice can be $\phi_{0,0}$ isomorphic to each other, even if the corresponding set of edges deleted from the pattern $P$ to generate the nodes are not equal. Since, the vertex sets for these isomorphic nodes are the same we call them automorphic nodes. Any automorphism is simply a non-identity permutation of the vertices. We will show that the output of the function VerifyCandidates (see section 7.5.3.1) is the same for all automorphisms. Therefore, we can detect all $\phi_{\alpha,\beta}$ isomorphisms of a pattern by running VerifyCandidates function on any automorphism in a group of automorphisms.

**Theorem 7.5.1.** Let $P_x$, $P_y$ be any two automorphic nodes in the lattice. Then for all $u \in P_x$ $\text{VerifyCandidates } (P_x, T_c(u)) = \text{VerifyCandidates } (P_y, T_c(v))$

**Proof.** Since, $P_x$ and $P_y$ are automorphic, let $\pi$ be the permutation of vertices that maps $u \in P_x$ to $v \in P_y$. Therefore, $\pi(u) = v$. Without loss of generality, assume that $I(P_y, u, v, \alpha, 0) = 1$. then we will prove that $I(P_x, \pi^{-1}(u), v, \alpha, 0) = 1$. Intuitively, it proves that $P_y$ doesn’t introduce any representatives for $u$ that are not detected by $P_x$.

From $I(P_y, u, v, \alpha, 0) = 1$ we know that $\exists \phi_{\alpha,0}$ such that $\phi_{\alpha,0}(u) = v$. Now consider the composite function $\phi'_{\alpha,0} = \phi_{\alpha,0} \circ \pi$ for mapping $P_x$ into the database. A vertex $u$ is mapped to $\phi'_{\alpha,0}(u) = \phi_{\alpha,0}(\pi(u))$. It is enough if we show that the $\phi'_{\alpha,0}$ so defined is valid isomorphism of $P_x$ because the mapping of $\pi^{-1}(u)$ is equal to $\phi'_{\alpha,0}(\pi^{-1}(u)) = \phi_{\alpha,0}(\pi(\pi^{-1}(u))) = \phi_{\alpha,0}(u) = v$ and consequently $I(P_x, \pi^{-1}(u), v, \alpha, 0) = 1$

We will argue that the three conditions (refer the definition in section 1.4.2) for the $\phi_{\alpha,0}$ approximate isomorphism are met.

- **$\phi'_{\alpha,0}$ is injective function** : Suppose $\phi'_{\alpha,0}(u_1) = \phi'_{\alpha,0}(u_2)$, then $\phi_{\alpha,0}(\pi(u_1)) = \phi_{\alpha,0}(\pi(u_2))$ which is a contradiction because $\phi_{\alpha,0}$ is isomorphic.

- **Label mismatch cost is at most $\alpha$** : The label mismatch cost of the $\phi'_{\alpha,0}$ can be written as $\sum_{u' = \pi^{-1}(u), u \in P_y} C[L(u')][L(\phi'_{\alpha,0}(u'))]$. Replacing $u'$ with $\pi^{-1}(u)$, the expression reduces to $\sum_{u \in P_y} C[L(\pi^{-1}(u))][L(\phi_{\alpha,0}(\pi(\pi^{-1}(u'))))]$ which is equal to $\sum_{u \in P_y} C[L(u)][L(\phi_{\alpha,0}(u))] \leq \alpha$
• **Structure mismatch cost is 0**: We can see that $\phi'_{\alpha,0}$ preserves the structure of $P_x$ because of the transitive property of the $\phi_{0,0}$ isomorphism.

We need a way to find and eliminate duplicate automorphisms during the search space exploration. We can simply maintain a list of all the nodes visited in the lattice and for each node in the search we check if it is automorphic to any of the visited nodes. If yes, we remove the entire subtree under it from the search space. Instead of pairwise comparisons we store a hash to prune non automorphic nodes immediately without the costly isomorphism checker.

| IsVisited : $(P_x)$ $\rightarrow$ Returns true if no automorph of $P_x$ has been visited in the search process till now |

### 7.5.3.4 Enumerating connected subgraphs:

Finally, we describe the procedure for enumerating all connected subgraphs of the pattern $P$ with at most $\beta$ edges missing. The outgoing nodes from any node $P^*$ in the lattice correspond to all the edges that lie on a cycle in $P^*$. The cycles can be easily detected via a Depth First Search (DFS) starting from any vertex. Instead of running DFS after each edge deletion, we construct a DAG of strongly connected components (SCC) in a DFS tree and use it to easily compute the edges that lie on a cycle.

Consider any connected graph $G$, we define the following concepts. *Spanning Tree*: DFS starting from any vertex $u$ results in a directed tree $T$ also called as spanning tree of $G$. The DFS tree, $D$, corresponding to $T$ is constructed by adding an edge from $u$ to $v$ for all $(u, v) \in G - T$. It can be shown that there are no cross edges between different branches in the DFS tree $D$.

*DAG of connected components*: SCCs of a directed graph are set of vertices for which there is a directed path between every pair of vertices. By performing a DFS on the tree $D$, we can compute its SCCs. Let $D'$ be a graph in which each node corresponds to a SCC in $D$ and there is an between two nodes if the corresponding component nodes are connected.
Candidate Edges: All the edges that belong to a connected component with at least 2 edges can be deleted without disconnecting the graph \( G \).

**Theorem 7.5.2.** For all \( e = (u, v) \in G \), \( e \) lies on a cycle iff \( e \in CC \) for some connected component \( CC \) of \( D' \) with at least 2 edges.

**Proof.** We will show that there exists \( x \neq u, v \) such that all three vertices \( u, v \) and \( x \) belong to the same connected component in \( D' \). A vertex \( x \neq u, v \) has to exist because \( e \) lies on a cycle. Now, the DFS on \( G \) reaches one of the three vertices first and the search visits the other two vertices before returning. It is easy to prove that these vertices are connected to each other and therefore lie in the same SCC of \( D \). Backward direction of the proof is trivial. \( \square \)

**CandidateEdges**: \((G) \rightarrow \) edges \( e \) such that \( G - e \) is connected

### 7.5.4 Derived vertex for edge deletion case

The k-hop label matching condition used in the LMiner algorithm states that in any \( \phi_{\alpha,0} \) isomorphism that maps a pattern vertex \( u \) to a database vertex \( v \) the cost matching the k-hop labels \( h_k(u) \) and \( h_k(v) \) is at most \( \alpha \). An equivalent of k-hop label matching condition in the case of \( \phi_{\alpha,\beta} \) can be stated as

*In any \( \phi_{\alpha,\beta} (u) = v \) there exists a subset \( S, \emptyset \subseteq S \subset E_P \) such that \( h_k(u)' \leq h_k(v) \) where \( h_k(u)' \) is the k-hop label of \( u \) in the pattern \( P^* = P - S \).*

We will now argue that such a condition is not effective in pruning candidate representatives. First, note that if \( h_k(u) \leq h_k(v) \) then obviously \( S = \emptyset \) and verification step is required unless the mapping from \( u \) to \( v \) is pruned by the NL matching condition. Now assume that \( h_k(u) \not\leq h_k(v) \). If we can efficiently compute the change in the k-hop label of a vertex under an edge deletion then we would be able to compute a list of possible edge deletions such that \( h_k(u) \leq h_k(v) \). To see this, consider the following scenario: Let \( h_k(u) = \{1, 2, 3\} \) and \( h_k(v) = \{10, 20\} \) such that labels of vertices \([1, 2, 3, 10, 20]\) are \([A, B, C, A, B]\) respectively. Also, the cost matrix is boolean and \( \alpha = 0 \). It can be seen that the k-hop label matching condition is violated because of the vertex \( 3 \in h_k(u) \). If we can somehow guarantee that some edge \((x, 3)\) is the last edge on all \( k \) length paths between \( u \) and \( 3 \) then the
Figure 7.4: (a): sample database graph $G$, (b): cost matrix. (c): Partial search space lattice.

$(x, 3)$ is the first edge that we would try and delete because the k-hop label after the deletion would be $h_k(u)' = \{1, 2\}$ which can matched to $h_k(v)$. As it turns out, computing the edges that lie on all the $k$ length paths between two vertices is an NP-Complete problem as we show below.

**Theorem 7.5.3.** Let $G$ be any graph, $u, v \in G$. Define the set $A(u, v, k, G) = \{(x, y) \mid \forall$ k-length paths $P$ between $u,v, (x, y) \in P, E_G\}$

**Proof.** We reduce this problem to decision version of a variant of Hamiltonian path.
Define, *HPath*: *Is there path that visits each vertex exactly once?* *HPath*\(^*\): *Is there an HPath that begins that at u and ends in v?* *HPath*\(^*\) is an NP-Complete problem, can be proved by reducing to *HPath*.

Let the graph \( H = (V_H, E_h) \) be an instance of *HPath* problem then construct a graph \( H' = (V_H \cup \{u, v\}, E_{h'} \cup \{(u, u'), (v, v')\}) \) for some \( u, v \notin V_H \) and \( u', v' \in V_H \).

Let \(|H| = n\). We will prove the statement: \( A(u, v, n + 1, H') = E_{H'} \) or \( \emptyset \) if and only if \( H' \) is just path (line graph) or *HPath*\(^*\) returns false for *HPath* between \( u \) and \( v \) in the graph \( H' \).

\[ \rightarrow A(u, v, n + 1, H') = E_{H'} \text{ or } \emptyset : \text{if } A(u, v, n + 1, H') = E_{H'} \text{ then } H' = \bigcup E_{H'} = \bigcup A(u, v, n + 1, H'). \] Therefore, \( H' \) is just line graph connecting \( u \) and \( v \). On the other hand if \( A(u, v, n + 1, H') = \emptyset \) then *HPath*\(^*\) is false because by construction \((u, u')\) and \((v', v)\) are on every *HPath* that exists in \( H' \).

\[ \leftarrow H' \text{ is path or no HPath*}: \text{If } H' \text{ is a path then obviously } A(u, v, n + 1, H') = E_{H'}. \text{ Or, if } H' \text{ has no HPath between } u \text{ and } v \text{ then obviously } A(u, v, n + 1, H') = \emptyset. \]

Finally, given an instance of *HPath* problem, we construct an instance *HPath*\(^*\) problem as above. If computing \( A(u, v, n + 1, H) \) is polynomial then we can use it check whether \( H \) has a Hamiltonian path, a contradiction. Therefore, the edges that lie on all the \( k \) length paths cannot be computed in polynomial time. \( \square \)

### 7.5.4.1 Fixing a pair vs pattern

The AMiner algorithm explores the lattice composed of subgraphs of \( P \). At a node \( P* \) in the lattice, it tries to verify candidate representatives by using LMiner algorithm to find \( \phi_{\alpha,0} \) isomorphisms of \( P* \) in the database. Any vertex for which \( I(P*, u, v, \alpha, 0) = 1 \) implies that \( I(P, u, v, \alpha, \beta) = 1 \) because from the construction of lattice we know that \(|E_P| - |E_{P*}| \leq \beta\). In other words, for each \((u_1, u_2) \in E_P - E_{P*}\) if we add the edge \((\phi_{\alpha,0}(u_1), \phi_{\alpha,0}(u_2))\) in the database then we find an exact structural isomorphism with bounded label mismatch cost in the database.

Alternatively, we could fix a pair \((u, v)\) and explore all possible edge deletions i.e., the same lattice until a \( \phi_{\alpha,0} \) isomorphism such that \( \phi_{\alpha,0}(u) = v \) is found. This method is inferior to to our approach because when the pair \((u, v)\) is fixed we can apply only k-hop pruning because NL is based on label propagation in the pattern.
LMiner algorithm shows that NL with k-hop performs better compared to k-hop label.
REFERENCES


Hsu, “Prefixspan: mining sequential patterns efficiently by prefix-projected


description length and background knowledge,” J. Artif. Int. Res., vol. 1,


IEEE Int. Conf. on Data Mining, 2001, pp. 313–320.

[16] X. Yan and J. Han, “gspan: graph-based substructure pattern mining,” in
Proc. 2nd IEEE Int. Conf. on Data Mining, 2002, pp. 721–724.


[18] J. Huan, W. Wang, and J. Prins, “Efficient mining of frequent subgraphs in
the presence of isomorphism,” in Proc. 3rd IEEE Int. Conf. on Data Mining,
2003, pp. 549–552.

[19] X. Yan and J. Han, “Closegraph: Mining closed frequent graph patterns,” in
Proc. 9th ACM SIGKDD Int. Conf. on Knowledge Discovery and Data

approach to pattern mining: data mining template library,” Data Mining and


[49] Z. Zou, H. Gao, and J. Li, “Discovering frequent subgraphs over uncertain graph databases under probabilistic semantics,” in Proc. 16th ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining, 2010, pp. 633–642.


