EXPLORING THE POWER OF FREQUENT NEIGHBORHOOD PATTERNS ON EDGE WEIGHT ESTIMATION

by

Li Xiong

B.Eng., Sichuan University, 2013

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in the
School of Computing Science
Faculty of Applied Sciences

© Li Xiong 2015
SIMON FRASER UNIVERSITY
Summer 2015

All rights reserved.
However, in accordance with the *Copyright Act of Canada*, this work may be reproduced without authorization under the conditions for "Fair Dealing." Therefore, limited reproduction of this work for the purposes of private study, research, criticism, review and news reporting is likely to be in accordance with the law, particularly if cited appropriately.

APPROVAL

Name: Li Xiong

Degree: Master of Science

Title of Thesis: Exploring the Power of Frequent Neighborhood Patterns on Edge Weight Estimation

Examining Committee:

Dr. Binay Bhattacharya,
Professor, Chair

Dr. Jian Pei,
Professor, Senior Supervisor

Dr. Joseph Peters,
Professor, Supervisor

Dr. Ramesh Krishnamurti,
Professor, Internal Examiner

Date Approved: June 9th, 2015

Abstract

Since links on social networks model a mixture of many factors, such as acquaintances and friends, the problem of link strength prediction arises: given a social tie $e = (u, v)$ in a social network, how strong the tie e is? Previous work tackles this problem mainly by node profile-based methods, i.e., utilizing users' profile information. However, some networks do not have node profiles. In this thesis, we study a novel problem of exploring the power of frequent neighborhood patterns on edge weight estimation. Given a labeled graph, we estimate its edge weights by applying its structural information as features. We develop an efficient pattern-growth based mining algorithm to mine frequent neighborhood patterns as features to estimate edge weights. Our experimental results on two real datasets show the efficiency of our method and the effectiveness of the frequent neighborhood pattern based features.

Keywords: graph mining; frequent pattern mining; social network; link strength

To my parents and my brother.

"I'm not young enough to know everything."

— J. M. Barrie, *(1860-1937)*

Acknowledgments

I would like to express my deepest gratitude to my senior supervisor Dr. Jian Pei, for his great patience, warm encouragement and continuous support throughout my Master's study. He shared with me not only valuable knowledge and the attitude to research, but also the wisdom of life. Without his help, never can I accomplish this thesis.

I would like to thank Dr. Joseph Peters for being my supervisor and giving me helpful suggestions on my thesis. I also thank Dr. Ramesh Krishnamurti and Dr. Binay Bhattacharya for serving in my examining committee.

I am also very grateful to my friends during my Master's study at Simon Fraser University for their kind help. A particular thank goes to Junyi Shao, Yu Yang, Lin Liu, Beier Lu, Jiaxing Liang, Lumin Zhang, Xiang Wang, Yu Tao, Xiaoning Xu, Guanting Tang, Xiangbo Mao, Xiao Meng, Chuancong Gao, Juhua Hu, Zicun Cong, Xiaojian Wang, Xuefei Li and Tong He.

I am so thankful for having my friend Peng Zhang at Purdue University for her encouragement during my Master's study and insightful comments on my thesis.

Last but not least, my sincerest gratitude goes to my parents and my brother for their endless love and support through all these years.

Contents

Approval ii

Abstract iii

Dedication iv

Quotation v

Acknowledgments vi

Contents vii

List of Tables ix

List of Figures x

1 Introduction 1

2 Related Work 4

2.1 Frequent Subgraph Mining 4

2.1.1 Subgraph Mining in Graph Transactions 4

2.1.2 Subgraph Mining in a Single Large Graph 5

2.2 Link Strength Prediction 7

3 Problem Definition 9

3.1 Preliminaries 9

3.2 Problem Statement 13

4 The Pattern Mining Algorithm **14**
4.1 The Pattern Growth Based Mining Algorithm 14
4.2 Pivoted Subgraph Isomorphism Test 17
4.3 Pattern Extend Algorithm Revisited 26
4.3.1 Pivoted Graph Isomorphism Test 26
4.3.2 Pattern Children Generation 27

5 Edge Weight Estimation **29**
5.1 Framework 29
5.2 Edge Feature Construction 30
5.3 Regression Problem Formulation 32
5.4 Gradient Boosting Overview 33
5.4.1 The Gradient Boosting Model 33
5.4.2 Model Parameters 35
5.4.3 Time Complexity 36

6 Experiments **37**
6.1 Environments and Datasets 37
6.2 Baseline Algorithms 39
6.3 Efficiency of Our Pattern Mining Algorithm 40
6.4 Effectiveness of Frequent Neighborhood Pattern Features 45

7 Conclusions **49**

Bibliography **51**

List of Tables

4.1 DFS Codes for the Pivoted Graphs in Figures 4.2 (a) - 4.2 (c) 22

5.1 Parameters of the Gradient Boosting Tree Regression Model 36

6.1 DBLP Dataset Statistics 38
6.2 Tencent Weibo Dataset Statistics 38
6.3 The Effect of τ and r on Estimation Performance on the DBLP Dataset . . . 46
6.4 The Effect of τ and r on Estimation Performance on the Tencent Weibo Dataset 47
6.5 Parameters of Gradient Boosting in Experiments 48
6.6 RMSE Comparison of Edge Weight Estimation on the DBLP Dataset 48
6.7 RMSE Comparison of Edge Weight Estimation on the Tencent Weibo Dataset 48

List of Figures

3.1 An example of labeled graph and neighborhood pattern. 10

4.1 Example of one edge growth of neighborhood pattern. 17
4.2 DFS trees of pivoted graph and the forward/backward edge sets. 18
4.3 All the children of the pivoted graph $\mathcal{P}_{pa}$ in Figure 4.1 by rightmost extension. 28

5.1 An example of counting the number of pattern matches. 32

6.1 Edge weight distribution of the DBLP dataset. 38
6.2 Edge weight distribution of the Tencent weibo dataset. 39
6.3 The number of patterns under different supports and pattern sizes on the DBLP dataset. 41
6.4 Running time comparison between our algorithm and Apriori base algorithm under 4 support threshold values on the DBLP dataset. In each subfigure, the curves show the running time under different pattern size constraints. . . 42
6.5 The number of patterns under different supports and pattern sizes on the Tencent weibo dataset. 43
6.6 Running time comparison between our algorithm and Apriori base algorithm under 4 support threshold values on the Tencent weibo dataset. In each subfigure, the curves show the running time under different pattern size constraints. 44

Chapter 1

Introduction

With the popularity of social networks, such as Facebook, Twitter, and LinkedIn, people get connected more easily than ever before because of the low cost of link formation. Interestingly, recent studies [14, 9] show that the number of followers and followees of a user in a social network does not indicate the number of friends the user has. In fact, users have a much smaller number of friends compared with the number of connections they declare in social networks. For example, Huberman *et al.* [14] showed that users only interact with a small number of people on Facebook while they have a large number of connections. Social network ties model a mixture of factors, such as acquaintances and friends. To better understand the links in social networks, the problem of tie strength prediction arises [33, 8, 17]: given a tie e with two endpoints u and v, tie strength prediction is to predict how strong the connection between u and v.

Existing approaches on the tie strength prediction rely highly on node profile information. For example, Xiang *et al.* [33] proposed an unsupervised model to estimate relationship strength based on profile similarity and interaction activities. A profile consists of information such as hometown, school, employer, etc. Interaction activities on Facebook include poke, picture tagging, and so on. However, there are two limitations to this: (1) there is no straightforward profiles for nodes in some networks, such as DBLP bibliographical network and, (2) there is a potential for incompleteness of node profile information which is vital for profile based methods. For example, Abel *et al.* [1] showed that, in the sample dataset they crawled from Twitter, only 48.9% users on average complete their profile attributes. Therefore, a follow-up question arises naturally: can we develop some *profile free* methods to estimate tie strength?

Most recently, topological features were successfully applied to link prediction, i.e., predicting the existence of a link in the future based on the current network structure [28]. Taskar *et al.* [30] showed that the principle of homophily [25] can be used to improve the performance of link prediction in relational data. Intuitively, the homophily suggests that an interaction between similar people occurs at a higher rate than dissimilar people. The curiosity about link strength, the intuitiveness of homophily, and the success of topological features on link prediction motivate the research question in this thesis: *given an edge* $e = (u, v)$*, if* e *can be represented by topological features, can we estimate the edge weight of* e *effectively without the profiles of* u *and* v*?*

Thus, to address our research question, we further ask: (1) how can we find structural features to represent an edge in a network and, (2) given the topological features of edges, how do we estimate edge weights?

For the first question, we develop a method to construct the topological features of an edge $e = (u, v)$ based on u's and v's frequent neighborhood patterns defined by Han and Wen [11]. Details about feature construction will be explained in Chapter 5. Frequent neighborhood patterns carry rich semantics about the behaviours and the roles of their pivot nodes in the patterns. For example, in the DBLP bibliographical network, a frequent neighborhood pattern may indicate that most authors cite their own papers. In social networks, a frequent neighborhood pattern may represent a person who is a professor and a hockey fan as well. However, the frequent pattern mining algorithm described by Han and Wen [11] is inefficient in both time and memory, since it adopts an Apriori [2] based, breadth first search pattern mining scheme. In this thesis, inspired by gSpan [34] which is a frequent subgraph mining algorithm under graph transaction setting, we propose a pattern growth based frequent neighborhood pattern mining algorithm. We further accelerate the pivoted subgraph isomorphism test [11] process by applying *minimum DFS encoding*, a canonical labeling system for graph. Experimental results show the superiority of our pattern mining algorithm compared with the Apriori based pattern mining algorithm [11].

To address the second question, we formulate the edge weight estimation problem as a regression problem. The features are structural information of edges, and the target values are the corresponding edge weights. In our problem, considering that the feature vectors are represented by a sparse 0-1 matrix, we apply a gradient boosting regression [7] model to it. It has two advantages compared with other frequently used regression models, such as general linear regression [23], support vector machine (SVM) [29] based regression and neural

network [27] based regression: (1) gradient boosting is able to learn complicated relations among features, which is the advantage of tree based regression over linear regression and, (2) training a gradient boosting regression model is more efficient than training a SVM or neural network based regression model.

We summarize our contributions as follows.

- To the best of our knowledge, we are the first to formalize the problem of exploring the power of frequent neighborhood patterns on edge weight estimation.
- We design an efficient pattern growth based frequent neighborhood pattern mining algorithm.
- We explore the ways to construct structural features for edge weight estimation.
- We empirically show the efficiency of our proposed pattern mining algorithm and the effectiveness of frequent neighborhood pattern based features on real datasets.

The rest of the thesis is organized as follows. We review the related work in Chapter 2. In Chapter 3, we formulate the problem of exploring the power of frequent neighborhood patterns on edge weight estimation. In Chapter 4, we present our pattern growth based frequent neighborhood pattern mining algorithm. In Chapter 5, we discuss the method to construct edge features based on frequent neighborhood patterns, and introduce the regression model for edge weight estimation. We report the experimental results in Chapter 6. Finally, Chapter 7 concludes this thesis.

Chapter 2

Related Work

Our problem of exploring the power of frequent neighborhood patterns on edge weight estimation is related to the existing work on frequent subgraph mining and link strength prediction. In this chapter, we provide a brief review on some major related work and discuss the differences between our work and the existing ones.

2.1 Frequent Subgraph Mining

The problem of frequent subgraph mining (a.k.a. frequent pattern mining), is to find subgraphs, which occur frequently from either a graph transaction database or a single large graph. In the following sections, we will not distinguish subgraphs from patterns when there is no ambiguity. The existing work can be divided into two categories: *graph transaction based* and *single large graph based* frequent subgraph mining.

2.1.1 Subgraph Mining in Graph Transactions

Mining frequent subgraphs from graph transaction databases is well-investigated in literature. In this section, we briefly review the most representative greedy search based methods [13, 15, 18, 26, 34].

The definition of *occurrence counting measure* (support) under the graph transaction database scenario is straightforward. Given a subgraph G' and a graph database $\{G_i, i = 1, 2, \ldots, n\}$, the support of G' is the size of the set $\{i \mid G' \simeq G_i, i = 1, 2, \ldots, n\}$. The symbol $\simeq$ stands for subgraph isomorphism. Obviously, it satisfies the *downward closure property*

(also called *anti-monotonicity*) [2], which requires that the support of a pattern does not exceed that of its sub-patterns.

Inokuchi *et al.* [15] proposed an efficient algorithm, called AGM, on finding all frequent *induced* subgraphs in a graph database. An *induced* subgraph is a subset of the nodes of a graph G together with any edges whose endpoints are both in this subset. AGM utilizes the anti-monotonicity of the support to prune infrequent subgraphs. In this way, a lot of search space is reduced. Kuramochi *et al.* [18] further developed the idea of AGM using a more efficient graph representation structure. Their algorithm is called FSG, which is to find all the frequent connected subgraphs. FSG focuses on memory efficiency and algorithm scalability. Both AGM [15] and FSG [18] follow the Apriori based candidate expansion scheme. A size $k+1$ candidate is generated by joining two size k ones. AGM [15] extends a candidate by adding one vertex, while FSG [18] extends a candidate by adding one edge. The deficiencies are: (1) the pattern join operation is costly and, (2) a lot of false positives need to be pruned.

In order to make subgraph mining more efficient, McPherson *et al.* [26] investigated the use of the *quickstart principle*. The intuition is that searching frequent structures separately in an increasing complexity order (frequent paths, frequent trees and frequent cyclic graphs) can speed up the whole mining process. Yan and Han [34] proposed a pattern growth based approach to extend the candidates. Under the pattern growth approach, a size $k+1$ candidate is extended from a size k candidate directly (either by adding one edge or one node, which depends on the definition of pattern size) instead of joining two size k ones. Huan *et al.* [13] and Nijssen *et al.* [26] also adopted the pattern growth mechanism to extend a subgraph. In addition, Yan and Han [34] designed a new canonical labeling system called *minimum DFS encoding* to accelerate the process of subgraph isomorphism test.

Although the frequent subgraph mining algorithms under graph transaction setting can not be applied to the problem under a single large graph setting directly, the optimization techniques used in the subgraph isomorphism test and subgraph candidate generation methods are applicable with minor modifications.

2.1.2 Subgraph Mining in a Single Large Graph

Different from the research status quo of graph transaction based frequent subgraph mining, the problem under a single large graph setting is still under exploration. It is tricky because there is no straightforward support definition of a subgraph, which satisfies the downward

closure property. However, it is the key to do pruning in the candidate generation stage and further leads to an efficient mining algorithm.

Kuramochi *et al.* [19] and Vanetik *et al.* [32] proposed the maximum independent set (MIS) based support measure (MIS-support) and the corresponding frequent subgraph mining algorithms in a single large labeled graph. However, in order to compute the support of a subgraph, it needs to first solve the MIS problem which is NP-complete. Therefore, this method is computationally expensive. Fiedler and Borgelt [5] defined a support based on the MIS-support called *harmful overlap support* (HO-support). The HO-support relaxes the MIS-support by allowing overlapping embeddings which do not destroy the anti-monotonicity of support. However, the HO-support still suffers from high computational cost. Another anti-monotonic support measure, the *minimum image based support* (MNI-support), was defined by Bringmann and Nijssen [4]. It is based on the number of unique nodes in the graph $G = (V, E)$ to which a node of the pattern p (a subgraph of G) is mapped. The MNI-support is a support measure which avoids expensive MIS computations, thus it can be efficiently computed. Its result set is a superset of the MIS-support and HO-support. Therefore, by excluding unqualified subgraphs from the result set of the MNI-support measure, we can get the HO-support based results and the MIS-support based results. Han and Wen [11] also proposed an anti-monotonic support measure, which counts the number of nodes that can match the given subgraph with a pivot node specified. Under this support definition, we can see that the frequent patterns mined reveal the local topological information of the matching nodes. However, since Han and Wen [11] applied an Apriori based pattern generation paradigm, it is not efficient in both time and memory when extending a pattern, especially when a graph gets larger. In this work, we adopt the support measure defined in [11]. There are two reasons: (1) the support of a pattern can be computed much more efficiently than the other support measures discussed above and, (2) the frequent patterns under this support measure carry the structural information of the matching nodes, and we can further make use of this semantics. The detailed computational complexity of this support will be discussed in Chapter 4.

In this thesis, we explore the power of frequent neighborhood patterns on edge weight estimation. The first step of our work is to mine frequent neighborhood patterns (pivoted subgraphs) from a single large graph, which is the same as the problem in [11]. However, our work differs from theirs in two aspects. Firstly, we improve the pattern mining algorithm proposed in [11]. Instead of extending subgraphs in an Apriori based breadth first search

manner, we apply a pattern growth based depth first search scheme which is more efficient in both space and time. Secondly, we further improve our pattern growth based algorithm by applying *minimum DFS encoding* introduced in [34] to pattern generation and subgraph isomorphism test, which makes our proposed algorithm even faster.

2.2 Link Strength Prediction

In literature, *tie strength prediction* [8] is also called *edge weight estimation* in graphs or *link strength prediction* in heterogeneous information networks [17]. Let S denote a function which maps a link to its strength, and u, v denote two endpoints of a link, respectively. The strength can either be a discrete value, such as $S : (u, v) \rightarrow \{strong, medium, weak\}$, or a continuous value, such as $S : (u, v) \rightarrow \mathbb{R}_0^+$. The problem of tie strength prediction is to predict how strong the connection $S(u, v)$ is for two connected users u and v in a network (or two endpoints of an edge in a graph).

Most existing methods are based on the property of homophliy [25] in social networks. Homophily in social network theory postulates that similar users in a social network tend to establish a social relation. The higher the similarity, the stronger the tie. It is likely that the strength of link influences the frequency of interaction between users directly. Under the assumption of homophily, Kahanda and Neville [17] proposed a supervised learning approach to predict link strength on Facebook. They extracted features from four categories to train classifiers, i.e., attribute similarity, topological connectivity, transactional connectivity, and network-transactional connectivity. However, this work only differentiates between strong ties and weak ties. In order to predict continuous relationship strength from weak to strong, Xiang *et al.* [33] proposed a latent variable model based on profile similarity and social network interactions.

The existing approaches to predict link strength all construct features from node profile information. For example, node profile is user's profile information such as hometown, school, etc., on Facebook. To the best of our knowledge, there does not exist any previous work which predicts continuous link strength from weak to strong based on network structural information rather than node profiles. In this thesis, we bridge the gap.

Most recently, Sun *et al.* [28] exploited meta-path in predicting co-author relationships between existing authors in a heterogeneous bibliographic network. They declared higher

prediction accuracy than traditional homogeneous topological feature based models introduced in [22]. In this thesis, we study how topological information influences the formation of links in a heterogeneous network. However, Sun *et al.* [28] focused on predicting the existence of a new link using topological features and we focus on predicting the strength of existing links.

There also exists another line of research similar to ours which aims to predict the sign of relationship, i.e., positive (true friends) or negative (possible frenemy) [10, 20, 21, 35]. Guha *et al.* [10] applied a random walk based label propagation algorithm to predict the sign of link. In [35], a latent model called *Behavior Relation Interplay* (BRI) was proposed by Yang *et al.* to infer the sign of link. The basic idea in BRI is to infer the signs of social ties based on the decision making behaviors of users. Leskovec *et al.* [20] solved the link sign inference problem in a leave-one-out setting: given a social network with signs on all links except for that from node u to node v, how reliably can one infer this sign $s(u, v)$? They designed a classifier to predict $s(u, v)$ which was trained by two classes of features, i.e., node degree based features and *balance theory* based features. The balance theory of social networks implies the intuition that "a friend of my friend is my friend" and "an enemy of my enemy is my friend" [12]. Given the discovery of the correlation between frequent neighborhood patterns and edge weights in this thesis, we can further explore how those patterns affect the signs of links in the future.

Chapter 3

Problem Definition

In this chapter, we formulate our problem of exploring the power of frequent neighborhood patterns on edge weight estimation. For the sake of simplicity, we follow the terminology definitions in [11] throughout the thesis. At first, let us revisit the key concepts in [11] and some essential preliminaries of our problem with a running example, followed by the detailed statement of our problem.

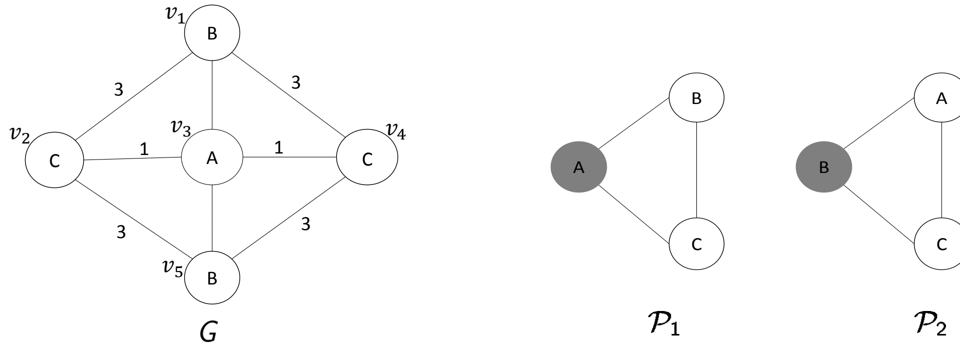
3.1 Preliminaries

Definition 3.1 (Labeled Graph)**.** *Let Σ_v denote the node label alphabet. An undirected labeled graph is a 4-tuple $G = (V, E, l, w)$, where V is the set of nodes in G, $E \subseteq V \times V$ is the set of edges in G, l is a labeling function $l : V \rightarrow \Sigma_V$ and w is a weight function $w : E \rightarrow \mathbb{N}$. Every node in G has exactly one label.*

For example, Figure 3.1a shows a **labeled graph**, where the integers on edges are edge weights, and uppercase letters are the labels of nodes.

Definition 3.2 (Pivoted Graph)**.** *A pivoted graph $\mathcal{G} = (G, v_p)$ is a tuple of a labeled graph G and a node $v_p \in V$. v_p is called the* ***pivot*** *of $\mathcal{G}$.*

We define a **neighborhood pattern**, or a **pattern** for short, $\mathcal{P}$ to be a connected pivoted graph. For instance, Figure 3.1b shows two pivoted graphs $\mathcal{P}_1$ and $\mathcal{P}_2$, their pivots are marked in grey. In terms of neighborhood pattern, we ignore the edge weights, and only consider the topological structure of the graph. Without explicitly mentioning, the pivoted graphs discussed in this thesis are connected pivoted graphs.



(a) A labeled graph. Integers on edges are edge weights and uppercase letters are the labels of nodes.

(b) Two pivoted graphs, pivots are marked in grey.

Figure 3.1: An example of labeled graph and neighborhood pattern.

Definition 3.3 (Pivoted Subgraph Isomorphism). *A pivoted graph $\mathcal{G}_1 = (G_1, v_{p1})$ is pivoted subgraph isomorphic to a pivoted graph $\mathcal{G}_2 = (G_2, v_{p2})$, denoted by $\mathcal{G}_1 \subseteq_p \mathcal{G}_2$, if there exists an **injective** function $f : V_1 \to V_2$ such that:*

- $\forall v \in V_1, l_1(v) = l_2(f(v))$
- $\forall (u, v) \in E_1, (f(u), f(v)) \in E_2$
- $f(v_{p1}) = v_{p2}$

Definition 3.4 (Pivoted Graph Isomorphism). *A pivoted graph $\mathcal{G}_1 = (G_1, v_{p1})$ is pivoted graph isomorphic to a pivoted graph $\mathcal{G}_2 = (G_2, v_{p2})$, denoted by $\mathcal{G}_1 \simeq_p \mathcal{G}_2$, if there exists a **bijective** function $f : V_1 \to V_2$ such that:*

- $\forall v \in V_1, l_1(v) = l_2(f(v))$
- $\forall (u, v) \in E_1, (f(u), f(v)) \in E_2$
- $f(v_{p1}) = v_{p2}$

From Definition 3.3 and Definition 3.4, it is easy to see that pivoted graph isomorphism is a special case of pivoted subgraph isomorphism.

Theorem 3.1. [11] *The problem of pivoted subgraph isomorphism test is NP-complete.*

Theorem 3.2. *The problem of pivoted graph isomorphism test is in GI.*

Proof. We can reduce the problem of graph isomorphism to the problem of pivoted graph isomorphism in polynomial time as follows. Given two graphs G_1 and G_2, we add a node v_{p1} to G_1, connect v_{p1} with every node in G_1, and set v_{p1} as the pivot of the pivoted graph $\mathcal{G}_1 = (G_1, v_{p1})$. We then add a node v_{p2} to G_2, connect v_{p2} with every node in G_2, and set v_{p2} as the pivot of the pivoted graph $\mathcal{G}_2 = (G_2, v_{p2})$. It is easy to see that if we solve the problem of pivoted graph isomorphism test on $\mathcal{G}_1$ and $\mathcal{G}_2$, we solve the graph isomorphism problem on G_1 and G_2, too. This is a polynomial time reduction. $\square$

Therefore, the problem of graph isomorphism test is no harder than the problem of pivoted graph isomorphism test. Since there does not exist a polynomial time algorithm for the graph isomorphism problem, it is also computationally expensive to solve the pivoted graph isomorphism problem.

Property 3.1. *The relations $\subseteq_p$ and $\simeq_p$ are transitive.*

Proof. Consider three arbitrary graphs $\mathcal{G}_1$, $\mathcal{G}_2$ and $\mathcal{G}_3$ such that $\mathcal{G}_1 \subseteq_p \mathcal{G}_2$, $\mathcal{G}_2 \subseteq_p \mathcal{G}_3$. Let f_{12} be the injective function $f_{12} : V_1 \to V_2$ and f_{23} be the injective function $f_{23} : V_2 \to V_3$ both satisfying the requirements in Definition 3.3. Consider the injective function $f_{13}(v) = f_{23}(f_{12}(v))$, which maps V_1 to V_3. We then show that f_{13} certificates $\mathcal{G}_1 \subseteq_p \mathcal{G}_3$. First, for any $v \in V_1$, $l_1(v) = l_2(f_{12}(v)) = l_3(f_{23}(f_{12}(v))) = l_3(f_{13}(v))$. Second, for any $(u, v) \in E_1$, we have $(f_{12}(u), f_{12}(v)) \in E_2$. By the definition of f_{23}, we have $(f_{23}(f_{12}(u)), f_{23}(f_{12}(v))) \in E_3$, which means that $(f_{13}(u), f_{13}(v)) \in E_3$. Third, we know that $f_{12}(v_{p1}) = v_{p2}$ and $f_{23}(v_{p2}) = v_{p3}$, thus $f_{13}(v_{p1}) = f_{23}(f_{12}(v_{p1})) = v_{p3}$. Therefore, $\mathcal{G}_1 \subseteq_p \mathcal{G}_3$ holds and $\subseteq_p$ is transitive. By a similar way, we can prove that the relation $\simeq_p$ is transitive, too. $\square$

Based on the definition of pivoted subgraph isomorphism, we define **match** between a pivoted graph $\mathcal{P}$ and a node v of data graph G: $\mathcal{P}$ matches v in G if $\mathcal{P}$ is pivoted subgraph isomorphic to the pivoted graph $\mathcal{G} = (G, v)$. For instance, in Figure 3.1, $\mathcal{P}_1$ matches node v_3 in G, and $\mathcal{P}_2$ matches node v_1 and node v_5 in G.

Definition 3.5 (Support)**.** *Given a labeled graph $G = (V, E, l, w)$ and a pivoted graph $\mathcal{P}$, let $M_G(\mathcal{P}) = \{v \in V \mid \mathcal{P} \subseteq_p (G, v)\}$ be the set of all nodes in G that $\mathcal{P}$ matches. The support of $\mathcal{P}$ in G is the size of $M_G(\mathcal{P})$.*

Corollary 3.1. *The problem of computing whether the support of a pattern is greater than or equal to τ is NP-complete.*

Proof. Consider the case when $\tau = 1$. Given an instance $\langle \mathcal{G}, \mathcal{P}, \tau \rangle$, this problem is equivalent to the pivoted subgraph isomorphism test problem which is NP-complete. In other words, by solving the support checking problem, we are able to solve the pivoted subgraph isomorphic problem $\langle \mathcal{G}, \mathcal{P} \rangle$. Thus our problem is NP-hard. The solution of an instance of our problem is verified in polynomial time. Therefore, our problem is NP-complete. □

A **frequent neighborhood pattern** is a neighborhood pattern whose support is greater than or equal to a threshold τ. In this thesis, we represent each node by a 0-1 feature vector. The ith coordinate of this feature vector is 1 if the node matches the ith pattern. The length of the feature vector is the total number of frequent neighborhood patterns. Han and Wen [11] showed that the support in Definition 3.5 satisfies the *downward closure property*. Specifically, if two neighborhood patterns $\mathcal{P}_i$ and $\mathcal{P}_j$ satisfy $\mathcal{P}_i \subseteq_p \mathcal{P}_j$, then $\left|M_G(\mathcal{P}_i)\right| \geq \left|M_G(\mathcal{P}_j)\right|$. This property is very important in designing an efficient frequent neighborhood pattern mining algorithm.

Definition 3.6 (Pattern Size)**.** *Given a labeled graph $G = (V, E, l, w)$ and a pattern $\mathcal{P} = (G, v_p)$, the pattern size of $\mathcal{P}$ is the size of E.*

Example 3.1 (Frequent Neighborhood Pattern and Pattern Size)**.** *Assume the support threshold τ is 2. In Figure 3.1, according to Definition 3.3 and Definition 3.5, we get $M_G(\mathcal{P}_1) = \{v_3\}$, and $M_G(\mathcal{P}_2) = \{v_1, v_5\}$. The support of $\mathcal{P}_2$ is 2, and the support of $\mathcal{P}_1$ is 1. Therefore, $\mathcal{P}_2$ is a frequent neighborhood pattern and $\mathcal{P}_1$ is not. The pattern sizes of both $\mathcal{P}_1$ and $\mathcal{P}_2$ are 3.*

Definition 3.7 (Frequent Neighborhood Pattern Mining)**.** *Given a labeled graph $G = (V, E, l, w)$, a minimum support τ, and a maximum pattern size r , let*

$$\zeta(\mathcal{P}, v) = \begin{cases} 1 & \text{if } \mathcal{P} \subseteq_p (G, v) \\ 0 & \text{otherwise} \end{cases}$$

where $\mathcal{P} = (G', v)$ is a neighborhood pattern, and G' is a subgraph of G. Let

$$\sigma(\mathcal{P}, G) = \sum_{v \in V} \zeta(\mathcal{P}, v)$$

where $\sigma(\mathcal{P}, G)$ *denotes the support of the pattern* $\mathcal{P}$ *in* G*, and let* $\kappa(\mathcal{P})$ *denote the pattern size of* $\mathcal{P}$*. Frequent neighborhood pattern mining is to find all the connected pivoted graphs such that* $\sigma(\mathcal{P}, G) \geq \tau$ *and* $\kappa(\mathcal{P}) \leq r$*.*

3.2 Problem Statement

Given a labeled graph G, our problem is to explore the power of frequent neighborhood patterns on edge weight estimation. Concretely, it consists of two steps: (1) mining frequent neighborhood patterns from G and, (2) estimating edge weights by applying frequent neighborhood patterns as features.

The problem in the first step is well-defined in Definition 3.7. For the second step, we first formulate edge weight estimation as a regression problem since the edge weight is a continuous variable. Then we apply gradient boosting to train a regression model to estimate edge weights. The features for training the regression model are constructed using the frequent neighborhood patterns. The method of applying frequent neighborhood patterns to feature construction will be presented in Chapter 5. To measure the effectiveness of the frequent neighborhood pattern based features on edge weight estimation, we compare the root mean square error (RMSE) of our model with the RMSEs of three baseline algorithms. The baseline algorithms will be introduced in Chapter 6.

In this thesis, we use the pattern size to constrain the number of frequent neighborhood patterns. There are two reasons: (1) we focus on the semantics of neighborhood patterns of a node rather than arbitrarily large patterns and, (2) the larger a pattern, the more running time needed to compute its support and to check pivoted subgraph isomorphism. We set a pattern size constraint to make a tradeoff between semantics carried by a neighborhood pattern and time cost of pattern mining.

Chapter 4

The Pattern Mining Algorithm

In this chapter, we present the frequent neighborhood pattern mining algorithm which follows the pattern growth based depth first search (DFS) paradigm [34]. First of all, we introduce the overall picture of the algorithm, and then discuss the details of *minimum DFS encoding* based pivoted subgraph isomorphism test as well as the corresponding pattern extension algorithm.

4.1 The Pattern Growth Based Mining Algorithm

Algorithm 2 is the framework of this pattern growth based frequent pattern mining algorithm. The algorithm starts from the frequent edge patterns. Each frequent edge pattern is a frequent pivoted graph and has only one edge. Frequent edge patterns have been pre-computed by Algorithm 1. Firstly, it constructs all possible 1-edge graphs. The edge in each 1-edge graph is constructed by enumerating all possible combinations of node labels (line 2 - line 4). For example, consider two node labels x and y, the edge constructed by them is $e = (u, v)$ with $l(u) = x$ and $l(v) = y$. The pivot is the first node u in each 1-edge pivoted graph. Then the support of each 1-edge pivoted graph will be computed (line 5). The frequent ones will be added to the frequent edge pattern set $fedge$ (line 6). For each frequent edge pattern, we extend it by adding one edge to it in a depth first search manner. Let k denote the size of a pattern, the algorithm which extends a size k pattern to a size $k + 1$ pattern is displayed in Algorithm 3.

Algorithm 1 Frequent Edge Pattern Mining Algorithm

Input: G: labeled graph, τ: support threshold

Output: $fedge$: frequent edge patterns

1: $fedge \leftarrow \emptyset$
2: **for** each node label $x \in \Sigma_v$ **do**
3:    **for** each node label $y \in \Sigma_v$ **do**
4:       Construct a pivoted graph $\mathcal{G}_e$ with only one edge $e = (u, v)$,
        where $l(u) = x$, $l(v) = y$, pivot $v_p = u$.
5:       **if** $\sigma(\mathcal{G}_e) \geq \tau$ **then**
6:          $fedge \leftarrow fedge \cup \{\mathcal{G}_e\}$
7:       **end if**
8:    **end for**
9: **end for**
10: **return** $fedge$

Algorithm 2 Frequent Neighborhood Pattern Mining Algorithm

Input: G: labeled graph, $fedge$: frequent edge patterns, τ: support threshold, r: maximum pattern size

Output: $\boldsymbol{S}$: frequent neighborhood patterns

1: $\boldsymbol{S} \leftarrow \emptyset$
2: **for** each frequent edge $e \in fedge$ **do**
3:    Pattern_Extension(e, G, τ, r)
4: **end for**

Algorithm 3 Pattern Extension Algorithm

Input: s: frequent neighborhood pattern candidate, G, τ, r

Output: $\boldsymbol{S}$

1: **if** $\exists \mathcal{P} \in \boldsymbol{S}, s \simeq_p \mathcal{P}$ **then**
2: return;
3: **end if**
4: $\boldsymbol{S} \leftarrow \boldsymbol{S} \cup \{s\}$
5: Generate all the potential children of s with one edge growth;
6: **for** each child c **do**
7: **if** $\kappa(c) \leq r$ **and** $\sigma(c) \geq \tau$ **then**
8: Pattern_Extension(c, G, τ, r)
9: **end if**
10: **end for**

In Algorithm 3, we extend a frequent neighborhood pattern s (line 5). Before we generate the children of the current pattern s, we will first check whether the current frequent neighborhood pattern candidate s already exists in $\boldsymbol{S}$ via a pivoted graph isomorphism test (line 1). If s passes the test, which means it is a newly discovered frequent neighborhood pattern, we add it to $\boldsymbol{S}$ (line 4) and generate the children of s by adding one edge (line 5). The newly added edge can be an edge carrying a new node or an edge connecting two existing nodes. For instance, in Figure 4.1, when we extend the parent frequent neighborhood pattern $\mathcal{P}_{pa}$, we can generate its children like $\mathcal{P}_{c1}$, and $\mathcal{P}_{c2}$. The dashed lines are the newly added edges in $\mathcal{P}_{c1}$ and $\mathcal{P}_{c2}$, respectively.

For every child generated, if it satisfies the size and support constraints, we will continue extending it by calling the pattern extension procedure recursively (line 8); otherwise, the procedure terminates.

Computing the size of a neighborhood pattern is trivial. The method we apply to compute the support is enumerating all the occurrences of a neighborhood pattern s in G. An optimization is the following. For each pattern, we maintain a set of nodes which are the pivots of its occurrences. Next time, when we compute the support of its children, according to the downward closure property of the support measure, we only check their parent's pivots set, which reduces the size of the search space substantially.

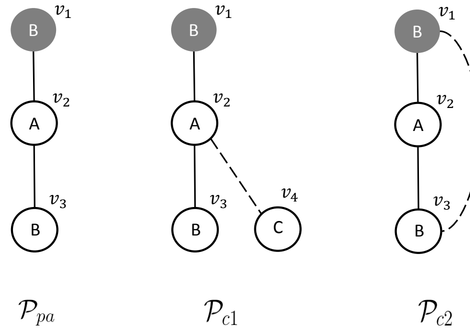


Figure 4.1: Example of one edge growth of neighborhood pattern.

4.2 Pivoted Subgraph Isomorphism Test

In order to accelerate the pivoted graph isomorphism test process, we borrow the idea of checking whether two pivoted graphs have the same canonical labels from McKay *et al.* [24]. A canonical labeling system maps a graph into a unique sequence. Yan and Han [34] developed a canonical labeling system called *minimum DFS encoding* to map a graph. If two graphs share the same *minimum DFS code*, these two graphs are isomorphic. However, there exists an obvious difference between our problem and the problem solved by Yan and Han [34]. In our problem we are checking pivoted graph isomorphism rather than graph isomorphism. Therefore, we need different DFS code to represent a pivoted graph.

Before we define our *minimum DFS encoding* of pivoted graph, let us first refresh some related concepts defined by Yan and Han [34]. We modify these concepts to fit our pivoted graph context.

Definition 4.1 (DFS Tree of Pivoted Graph)**.** *Given a pivoted graph $\mathcal{G}$, a* ***DFS tree*** *of $\mathcal{G}$ is a spanning tree that results from performing a depth first search in $\mathcal{G}$. The starting node of a depth first search in $\mathcal{G}$ is the pivot of $\mathcal{G}$.*

It is obvious that a pivoted graph can have more than one DFS tree with different edge growing orders. For example, Figures 4.2 (a) - 4.2 (c) are all pivoted graphs isomorphic to the pivoted graph $\mathcal{G}$ on the left. The thickened edges in Figures 4.2 (a) - 4.2 (c) represent three different DFS trees of $\mathcal{G}$. Note that for all the different DFS trees of a pivoted graph $\mathcal{G}$, their pivots are the same with $\mathcal{G}$, otherwise, they cannot be pivoted graphs isomorphic

to $\mathcal{G}$ according to Definition 3.4.

Let T denote a DFS tree of a pivoted graph. We use subscripts to label the order of depth first discovery of nodes.

DFS Subscripting. Given a DFS tree T of a pivoted graph $\mathcal{G}$, let $v_i \prec_T v_j$ indicate that v_i is discovered before v_j. $\forall i, j, v_i \prec_T v_j$ if $i < j$. Each node is assigned a subscript from 0 to $n-1$, where n is the total number of nodes in $\mathcal{G}$. We call such a pivoted graph $\mathcal{G}$ with subscripts to denote the order of node discovery as a **subscripted pivoted graph**, denoted by $\mathcal{G}_T$. v_0 is called the **root** of T and v_{n-1} is called the **rightmost node**. Note that, in a DFS tree of a pivoted graph, v_0 is always the pivot of $\mathcal{G}$. The straight path from v_0 to v_{n-1} is called the **rightmost path**.

Example 4.1 (DFS Subscripting)**.** *In Figure 4.2 (a), there are 5 nodes in total. In the DFS tree, the order of node discovery in the depth first search is v_0, v_1, v_2, v_3, v_4, which is consistent with the order of subscripts, namely, 0, 1, 2, 3, 4. For every pair of nodes v_i and v_j, if v_i is discovered before v_j, i.e., $v_i \prec_T v_j$, we have $i < j$, and vice versa. The root is v_0, and the rightmost node is v_4. The rightmost path is (v_0, v_1, v_2, v_4).*

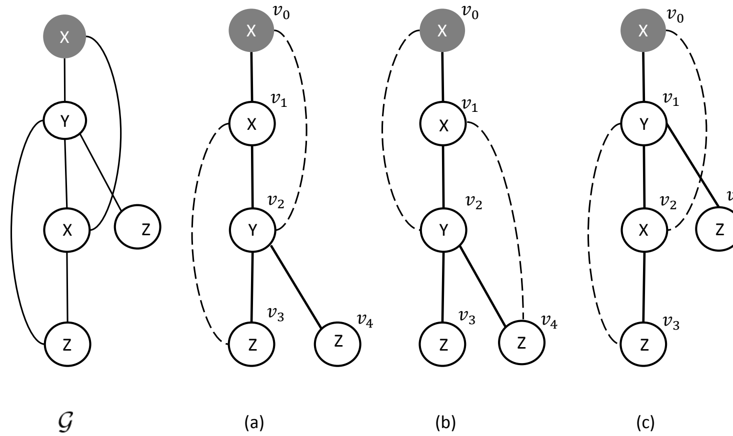


Figure 4.2: DFS trees of pivoted graph and the forward/backward edge sets.

Definition 4.2 (Forward Edge Set and Backward Edge Set)**.** *Given a subscripted pivoted graph $\mathcal{G}_T$, the **forward edge set** contains all edges in the DFS tree, denoted by $E_{f,T} = \{e \mid$*

$\forall i, j, i < j, e = (v_i, v_j) \in E\}$, and the ***backward edge set*** *contains all edges which are not in the DFS tree, denoted by $E_{b,T} = \{e \mid \forall i, j, i > j, e = (v_i, v_j) \in E\}$.*

In Figures 4.2 (a) - 4.2 (c), all the dashed edges are backward edges and all the thickened edges are forward edges.

We define three partial orders $\prec_{f,T}$, $\prec_{b,T}$ and $\prec_{bf,T}$ on $E_{f,T}$ and $E_{b,T}$. Consider $e_1 = (v_{i_1}, v_{j_1})$, $e_2 = (v_{i_2}, v_{j_2})$. We define

$$e_1 \prec_{f,T} e_2, \forall e_1, e_2 \in E_{f,T}, \tag{4.1}$$

if $j_1 < j_2$.

$$e_1 \prec_{b,T} e_2, \forall e_1, e_2 \in E_{b,T}, \tag{4.2}$$

if one of the following holds:

(1) $i_1 < i_2$;

(2) $i_1 = i_2$ and $j_1 < j_2$.

We also define

$$e_1 \prec_{bf,T} e_2, \tag{4.3}$$

if one the the following holds:

(1) $e_1 \in E_{b,T}, e_2 \in E_{f,T}, i_1 < j_2$;

(2) $e_1 \in E_{f,T}, e_2 \in E_{b,T}, j_1 \leq i_2$.

Let the relation $\prec_{E,T}$ denote the combination of the three partial orders 4.1, 4.2 and 4.3.

Theorem 4.1. [34] *The relation $\prec_{E,T}$ is a linear order.*

Example 4.2 (Linear Order $\prec_{E,T}$ on Edges)**.** *In Figure 4.2 (a), consider three edges $e_1 = (v_1, v_2)$, $e_2 = (v_2, v_0)$, and $v_3 = (v_2, v_3)$. According to the partial order 4.3, we have $e_1 \prec_{E,T} e_2$, and $e_2 \prec_{E,T} e_3$. Since $e_1 \prec_{E,T} e_2$ and $e_2 \prec_{E,T} e_3$, we have $e_1 \prec_{E,T} e_3$ according to the property of transitivity of $\prec_{E,T}$.*

Based on the concepts defined above, we define our key concepts for pivoted graph isomorphism test below. For simplicity, we use an ordered pair of subscripts (i, j) to denote an edge (v_i, v_j).

Definition 4.3 (DFS Code of Pivoted Graph). *Given a pivoted graph $\mathcal{G}$ and a DFS tree T of $\mathcal{G}$, the DFS code of $\mathcal{G}$ is an edge sequence $\alpha = \{e_i\}$ constructed based on the linear order $\prec_{E,T}$ such that $e_i \prec_{E,T} e_{i+1}$, where $i = 0, ..., |E| - 2$, and the pivot v_0 is in the first edge $e_0 = (0, 1)$ in a DFS code.*

Example 4.3 (DFS Code of Pivoted Graph). *Consider the pivoted graph in Figure 4.2 (a). The corresponding DFS code is an edge sequence. Let $\alpha = \{e_i\}$ denote the edge sequence. We have $\alpha = \{(0, 1), (1, 2), (2, 0), (2, 3), (3, 1), (2, 4)\}$. Every two adjacent edges e_i and e_{i+1} satisfy $e_i \prec_{E,T} e_{i+1}$, where $i = 0, \ldots, |E| - 2$.*

Definition 4.4 (DFS Edge). *A DFS edge is a 4-tuple, denoted by $a = (i, j, l(i), l(j))$, which represents an edge in a subscripted pivoted graph $\mathcal{G}_T = (G, v_p)$, where i and j are the subscripts of node v_i and node v_j, respectively, $l(i)$ and $l(j)$ indicate the labels of node v_i and node v_j, respectively.*

Example 4.4 (DFS Edge). *In Figure 4.2 (a), consider the edge $(1, 2)$. Its corresponding DFS edge is $(1, 2, X, Y)$, where 1 and 2 indicate the nodes v_1 and v_2, respectively, and X and Y are the labels of v_1 and v_2, respectively.*

To encode node labels into the DFS code of a subscripted pivoted graph, we represent an edge in a DFS code as a DFS edge. For example, the DFS code of Figure 4.2 (a) with DFS edge representation is $\alpha = \{(0, 1, X, Y),\ (1, 2, X, Y),\ (2, 0, Y, X),\ (2, 3, Y, Z),\ (3, 1, Z, X),\ (2, 4, Y, Z)\}$.

Given a subscripted pivoted graph, the method to construct its DFS code is presented in Algorithm 4. Let S_{old} denote the set of nodes which are put into the DFS code, and α denote the edge sequence of the DFS code. Firstly, add the pivot v_0 to S_{old}, $S_{old} = \{v_0\}$ (line 1). Then add the nodes $v_j \in V \setminus S_{old}$ to S_{old} one by one: (1) find the parent node v_i of v_j (line 3), add a forward edge (v_i, v_j) to α (line 4) and, (2) for each node $v_k \in S_{old}$, if $(v_j, v_k) \in E$, add a backward edge (v_j, v_k) to α (line 5 - line 9). After finishing adding all the forward and backward edges introduced by v_j, we add v_j to S_{old} (line 10). Repeat this procedure to grow the code until all edges are included in the DFS code. The order of edge addition guarantees the relation $\prec_{E,T}$ between two consecutive edges in a DFS code.

Algorithm 4 DFS Code Construction

Input: $\mathcal{G}_T = (G, v_p)$: subscripted pivoted graph

Output: α: DFS code of $\mathcal{G}_T$

```
1: Initialize S_old ← {v_0}
2: for j = 1 to n − 1 do
3:    Find the parent v_i of v_j
4:    Add a forward edge (v_i, v_j) to DFS code α ← α ∪ {(i, j, l(i), l(j))}
5:    for each v_k ∈ S_old with subscript k in ascending order do
6:       if (j, k) ∈ E then
7:          Add a backward edge to DFS code α ← α ∪ {(j, k, l(j), l(k))}
8:       end if
9:    end for
10:   S_old ← S_old ∪ {v_j}
11: end for
12: return α
```

Claim 4.1. *The Algorithm 4 is correct.*

Proof. Let $\alpha = \{a_t\}$ denote the DFS code outputted by Algorithm 4 from input–a subscripted pivoted graph $\mathcal{G}_T$. We prove that, for each $t = 0, 1, \ldots, |E| - 2$, $a_t \prec_{E,T} a_{t+1}$. Suppose there are t DFS edges in α so far, the tth edge is denoted by $a_t = (i_t, j_t, l(i_t), l(j_t))$. Suppose that v_j is the next node to be added to S_{old}, and let $\mathcal{E}_j$ denote the set of edges introduced by adding the new node v_j to S_{old}, and $k_{\mathcal{E}_j}$ denote the size of the set $\mathcal{E}_j$. The edge(s) in $\mathcal{E}_j$ can be:

1. only one forward edge $a_{t+1} = (i_{t+1}, j_{t+1}, l(i_{t+1}), l(j_{t+1}))$, where $v_{i_{t+1}}$ is the parent of v_j and $j_{t+1} = j$. We prove that, $a_t \prec_{E,T} a_{t+1}$. If a_t is a forward edge, since $j_t < j_{t+1}$, $a_t \prec_{E,T} a_{t+1}$ holds. If a_t is a backward edge, according to the partial order 4.3, $a_t \prec_{E,T} a_{t+1}$ holds because $i_t < j_{t+1}$.

2. one forward edge $a_{t+1} = (i_{t+1}, j_{t+1}, l(i_{t+1}), l(j_{t+1}))$ and $k_{\mathcal{E}_j} - 1$ backward edges. Let $a_{t+r} = (i_{t+r}, j_{t+r}, l(i_{t+r}), l(j_{t+r}))$, where $2 \le r \le k_{\mathcal{E}_j}$.

   We first prove $a_{t+1} \prec_{E,T} a_{t+r}$, where $2 \le r \le k_{\mathcal{E}_j}$. Since all the backward edges are grown from node v_j, we have $i_{t+r} = j$ for $2 \le r \le k_{\mathcal{E}_j}$. Because $j_{t+1} = j$, we

have $j_{t+1} = i_{t+r}$, where $2 \leq r \leq k_{\mathcal{E}_j}$. According to the partial order 4.3, we have $a_{t+1} \prec_{E,T} a_{t+r}$, where $2 \leq r \leq k_{\mathcal{E}_j}$.

Then we prove for arbitrary two backward edges $a_{t+r} = (i_{t+r}, j_{t+r}, l(i_{t+r}), l(j_{t+r}))$ and $a_{t+r+1} = (i_{t+r+1}, j_{t+r+1}, l(i_{t+r+1}), l(j_{t+r+1}))$, where $2 \leq r \leq k_{\mathcal{E}_j} - 1$, we have $a_{t+r} \prec_{E,T} a_{t+r+1}$. Since $i_{t+r} = i_{t+r+1} = j$ and $j_{t+r} < j_{t+r+1}$ (line 5), according to the partial order 4.1, we have $a_{t+r} \prec_{E,T} a_{t+r+1}$.

Finally, the relation $a_t \prec_{E,T} a_{t+1}$ can be proved in the same way as that in the first case.

And all the edges in the pivoted graph $\mathcal{G}_T$ have been included in this DFS code (line 2 - line 11), thus the Algorithm 4 is correct. □

By following Algorithm 4, we construct the DFS codes for the DFS trees shown in Figures 4.2 (a) - 4.2 (c), which are shown in Table 4.1. The *edge no.* in the first column indicates the order of DFS edges in the corresponding DFS code.

Table 4.1: DFS Codes for the Pivoted Graphs in Figures 4.2 (a) - 4.2 (c)

edge no.	(a) α	(b) β	(c) γ
0	(0, 1, X, X)	(0, 1, X, X)	(0, 1, X, Y)
1	(1, 2, X, Y)	(1, 2, X, Y)	(1, 2, Y, X)
2	(2, 0, Y, X)	(2, 0, Y, X)	(2, 0, X, X)
3	(2, 3, Y, Z)	(2, 3, Y, Z)	(2, 3, X, Z)
4	(3, 1, Z, X)	(2, 4, Y, Z)	(3, 1, Z, Y)
5	(2, 4, Y, Z)	(4, 1, Z, X)	(1, 4, Y, Z)

Property 4.1 (DFS Code's Neighborhood Restriction)**.** *Given a pivoted graph $\mathcal{G}$, a DFS tree T, a DFS code $\alpha = code(\mathcal{G}, T) = (a_0, a_1, ..., a_m)$, $m \geq 2$, and two consecutive elements on the code a_k and $a_{k+1} (0 \leq k < m)$. Let $a_k = (i_k, j_k, l(i_k), l(j_k))$, and $a_{k+1} = (i_{k+1}, j_{k+1}, l(i_{k+1}), l(j_{k+1}))$, then a_k and a_{k+1} must agree with the following rules:*

rule 1. if a_k is a backward edge, then one of the following holds.

- *if a_{k+1} is a forward edge, then $i_{k+1} \leq i_k$ and $j_{k+1} = i_k + 1$;*
- *if a_{k+1} is a backward edge, then $i_{k+1} = i_k$ and $j_k < j_{k+1}$.*

rule 2. if a_k is a forward edge, then one of the following holds.

- *if a_{k+1} is a forward edge, then $i_{k+1} \leq j_k$ and $j_{k+1} = j_k + 1$;*
- *if a_{k+1} is a backward edge, then $i_{k+1} = j_k$ and $j_{k+1} < i_k$.*

Proof. The definition of DFS code of a pivoted graph requires $a_k \prec_{E,T} a_{k+1}$. For *rule 1*, (1) if a_k is a backward edge and a_{k+1} is a forward edge, according to the partial order 4.3, there should be $i_k < j_{k+1}$. According to Algorithm 4, Note that the DFS code is unique for a given DFS tree. i_{k+1} must be a node in S_{old}, thus $i_{k+1} \leq i_k$ holds; and j_{k+1} must be the next node which will be added to S_{old}, since previously added node is i_k, $j_{k+1} = i_k + 1$ holds; (2) if a_{k+1} is a backward edge, according to Algorithm 4, two consecutive backward edges must grow from the same node, which in this case is i_k, hence $i_{k+1} = i_k$ holds; because $a_k \prec_{b,T} a_{k+1}$, $j_k < j_{k+1}$ holds. By a similar way, we can prove *rule 2*. □

From above discussion, it is clear that the positions of the edges in a DFS code cannot be exchanged randomly, otherwise the relation $\prec_{E,T}$ between two consecutive DFS edges in a DFS code will not hold. For instance, we cannot change the position of edge no. 2 and edge no. 3 of code α in Table 4.1. Moreover, we can see that a pivoted graph can have more than one DFS code with different node permutations. The *minimum DFS encoding*, as a canonical labeling system, must select one as the unique representation of a pivoted graph. The following discussion provides a design of linear order among all DFS codes of a pivoted graph. Based on that linear order, the smallest DFS code of a pivoted graph will be selected as the canonical label.

Definition 4.5 (DFS Lexicographic Order)**.** *Suppose $Z = \{code(\mathcal{G}, T) \mid T$ is a DFS tree of $\mathcal{G}\}$, Suppose there is a linear order ($\prec_L$) in the label set Σ_V, then the combination of $\prec_{E,T}$ and $\prec_L$ is a linear order $\prec_e$ on the set $E \times \Sigma_V \times \Sigma_V$.* ***DFS Lexicographic Order*** *is a linear order defined as follows. If $\alpha = code(\mathcal{G}_\alpha, T_\alpha) = (a_0, a_1, ..., a_m)$ and $\beta = code(\mathcal{G}_\beta, T_\beta) = (b_0, b_1, ..., b_n)$, $\alpha, \beta \in Z$, then $\alpha \leq \beta$ if one of the following is true:*

- $\exists t, 0 \leq t \leq \min(m, n)$, *s.t.* $a_k = b_k$ *for* $k < t$, *and* $a_t \prec_e b_t$
- $a_k = b_k$ *for* $0 \leq k \leq m$, *and* $n \geq m$.

More specifically, the ***DFS Lexicographic Order in*** Z *is defined as follows. Consider two DFS codes $\alpha = code(G_\alpha, T_\alpha) = (a_0, a_1, \ldots, a_m)$ and $\beta = code(G_\beta, T_\beta) = (b_0, b_1, \ldots, b_n)$, $\alpha, \beta \in Z$, then $\alpha \leq \beta$ if one of the following is true. Assume the forward edge set and*

backward edge set for the subscripted pivoted graphs $\mathcal{G}_{T_\alpha}$ *and* $\mathcal{G}_{T_\beta}$ *are* $E_{\alpha,f}$, $E_{\alpha,b}$, $E_{\beta,f}$ *and* $E_{\beta,b}$, *respectively. Let* $a_t = (i_a, j_a, l(i_a), l(j_a))$ *and* $b_t = (i_b, j_b, l(i_b), l(j_b))$,

1. *for* t, $0 \le t \le \min\{m, n\}$, *we have* $a_k = b_k$ *for* $k < t$, *and*

$$a_t \prec_e b_t = \begin{cases} \textit{true}, & \textit{if } a_t \in E_{\alpha,b}, \textit{ and } b_t \in E_{\beta,f} \\ \textit{true}, & \textit{if } a_t \in E_{\alpha,b}, b_t \in E_{\beta,b}, \textit{ and } j_a < j_b \\ \textit{true}, & \textit{if } a_t \in E_{\alpha,f}, b_t \in E_{\beta,f}, \textit{ and } i_b < i_a \\ \textit{true}, & \textit{if } a_t \in E_{\alpha,f}, b_t \in E_{\beta,f}, i_a = i_b \textit{ and } l(i_a) \prec_L l(i_b) \\ \textit{true}, & \textit{if } a_t \in E_{\alpha,f}, b_t \in E_{\beta,f}, i_a = i_b, l(i_a) = l(i_b), \textit{ and } l(j_a) \prec_L l(j_b) \end{cases} \tag{4.4}$$

2. $a_k = b_k$ *for* $0 \le k \le m$, *and* $n \ge m$

For the subscripted pivoted graphs in Figures 4.2 (a) - 4.2 (c) and their corresponding DFS codes in Table 4.1, if the linear order $\prec_L$ on their node labels is X $\prec_L$ Y $\prec_L$ Z, according to Definition 4.5, the order among them is $\alpha < \beta < \gamma$. To show how the comparison between two DFS codes works, we take the comparison between the two DFS codes α and β from Table 4.1 as an example. The first four edges of α and β are all the same, so we compare the 5th edges of them, which are $\alpha_4 = (3, 1, Z, X)$ and $\beta_4 = (2, 4, Y, Z)$. Since α_4 is a backward edge while β_4 is a forward edge, according to the first rule in Equation 4.4, we get $\alpha_4 \prec_e \beta_4$. Therefore, $\alpha < \beta$.

Note that, the linear order $\prec_{E,T}$ is used to compare two DFS edges within a DFS code, and the linear order $\prec_e$ is used to compare two DFS edges in two different DFS codes. The relation $<$ used between two DFS codes in Definition 4.5 indicates the DFS lexicographic order between them. For example, consider two DFS codes α, β. If $\alpha < \beta$, it means that the DFS code α is DFS lexicographic less than the DFS code β.

Definition 4.6 (Minimum DFS Code of Pivoted Graph)**.** *Given a pivoted graph* $\mathcal{G}$, $Z(\mathcal{G}) = \{code(\mathcal{G}, T) \mid \forall T,\ T \textit{ is a DFS tree for } \mathcal{G}\}$, *based on the DFS lexicographic order, the minimum DFS code, denoted by* $\min(Z(\mathcal{G}))$, *is called the* ***Minimum DFS Code of Pivoted Graph*** $\mathcal{G}$, *which is also the canonical label of* $\mathcal{G}$.

For example, for the pivoted graph $\mathcal{G}$ in Figure 4.2, DFS code α is its minimum DFS code, i.e., $\alpha = \min(Z(\mathcal{G}))$. For two pivoted graphs $\mathcal{G}_1$ and $\mathcal{G}_2$, if $\min(Z(\mathcal{G}_1)) < \min(Z(\mathcal{G}_2))$, we say $\mathcal{G}_1 < \mathcal{G}_2$.

Theorem 4.2. *Given two pivoted graphs $\mathcal{G}_1$ and $\mathcal{G}_2$, $\mathcal{G}_1$ is pivoted graph isomorphic to $\mathcal{G}_2$ if and only if* $\min(Z(\mathcal{G}_1)) = \min(Z(\mathcal{G}_2))$.

Proof. Let $\mathcal{G}_1$ and $\mathcal{G}_2$ be two pivoted graphs satisfying $\mathcal{G}_1 \simeq_p \mathcal{G}_2$. By traversing these two pivoted graphs, we get the same set of subscripted pivoted graphs, and then we get the same set of DFS codes $Z(\mathcal{G}_1)$ and $Z(\mathcal{G}_2)$. Therefore, $\min(Z(\mathcal{G}_1)) = \min(Z(\mathcal{G}_2))$.

Let $\min(Z(\mathcal{G}_1))$ and $\min(Z(\mathcal{G}_2))$ denote the minimum DFS codes of two pivoted graphs $\mathcal{G}_1$ and $\mathcal{G}_2$, respectively, such that $\min(Z(\mathcal{G}_1)) = \min(Z(\mathcal{G}_2))$. It is obvious that all the pivoted graphs represented by the set of DFS codes $Z(\mathcal{G}_1)$ are mutually pivoted graph isomorphic, so do the pivoted graphs represented by $Z(\mathcal{G}_2)$. Because $\min(Z(\mathcal{G}_1)) = \min(Z(\mathcal{G}_2))$, the two pivoted graphs represented by $\min(Z(\mathcal{G}_1))$ and $\min(Z(\mathcal{G}_2))$ are identical. So the pivoted graph represented by $\min(Z(\mathcal{G}_1))$ is pivoted graph isomorphic to every pivoted graph represented by $Z(\mathcal{G}_2)$. And $\mathcal{G}_2$ is a pivoted graph represented by one of the DFS codes in $Z(\mathcal{G}_2)$. Therefore, $\mathcal{G}_1 \simeq_p \mathcal{G}_2$. □

Definition 4.7 (DFS Code's Parent and Child.)**.** *Given a DFS code $\alpha = (a_0, ..., a_m)$, a valid DFS code $\beta = (a_0, ..., a_m, b)$ is called a* ***child*** *of α and α is called β's* ***parent****, where b is the newly added DFS edge. We denote $children(\alpha) = \{\beta \mid \forall\beta, \alpha$ is β's parent $\}$.*

Definition 4.8 (DFS Code Tree)**.** *A* ***DFS Code Tree*** *is a tree in which each node represents a DFS code, the relation between parent node and child node complies with the relation described in Definition 4.7, and the relation between siblings is consistent with the DFS lexicographic order. That is, the pre-order search of DFS Code Tree follows the DFS lexicographic order. The tree is denoted by $\mathbb{T}$.*

Let α be a DFS code for a pivoted graph, We use $\min(\alpha)$ to denote the minimum DFS code of the pivoted graph represented by α.

Theorem 4.3. [34] *Given a DFS code β, if $\alpha = \min(\beta)$, then $\alpha < \beta$. Let $\mathbb{D}_\gamma = \{\eta \mid \forall\eta, \eta < \gamma\}$. $\forall\delta, \delta \in children(\beta)$, i.e., any valid DFS code generated by one edge growth from β, $\min(\delta) \in \mathbb{D}_\alpha \cup children(\alpha) \subseteq \mathbb{D}_\beta$.*

Theorem 4.3 implies that given a DFS code β, if $\alpha = \min(\beta)$, then α must appear before β in the DFS Code Tree in the pre-order traversal of the DFS Code Tree. This property is very important in pruning duplicate neighborhood patterns.

So far, we build all the theoretical foundations for the *minimum DFS encoding* based pivoted graph isomorphism test. An example is provided below to illustrate how to apply

DFS code to a pivoted graph isomorphic test. The naive algorithm to compute the minimum DFS code of a pivoted graph is to enumerate all the DFS codes of pivoted isomorphic graphs and compare them according to the lexicographic order defined in Definition 4.5. Note that it is different when enumerating isomorphic graphs and enumerating pivoted isomorphic graphs. For isomorphic graph enumeration, every node in the graph can be the starting node, i.e., v_0, so the time complexity of enumeration is $O(n!)$, where n is the number of nodes of the graph. However, v_0 in pivoted graph can only be the pivot. Therefore, the time complexity for enumeration is $O((n-1)!)$.

Example 4.5 (DFS Code Based Pivoted Graph Isomorphism Test)**.** *Consider two pivoted graphs shown in Figure 4.2 (b) and Figure 4.2 (c). Now we check whether they are pivoted graph isomorphic through the method described in this section. Firstly, we compute their minimum DFS codes. For both of them, we get their minimum DFS codes equal to α shown in Table 4.1. Therefore, these two pivoted subgraphs are pivoted graph isomorphic.*

4.3 Pattern Extend Algorithm Revisited

4.3.1 Pivoted Graph Isomorphism Test

In Algorithm 3, the most time-consuming parts are pivoted graph isomorphism (line 1) test and pattern support computation (line 7). Neither of them is known to be solved in polynomial time. Note that the purpose of checking whether the current pattern s is pivoted graph isomorphic to a pattern discovered before is to avoid duplicate pattern support computing, and further avoid duplicate patterns in the result set.

Armed with *minimum DFS encoding*, we develop a DFS code based pivoted graph isomorphism test algorithm. The new pattern extension algorithm is shown in Algorithm 5. In Algorithm 5, we represent a pattern by its corresponding DFS code. Now our method to check pivoted graph isomorphism (line 1) becomes: given a pattern candidate s, we compute its minimal DFS code, denoted by $\min(s)$. If s is not equal to $\min(s)$, which means that s has been computed before according to Theorem 4.3, we can prune it immediately. Once s is pruned, all the descendants of s can be pruned. We do not need to compute the support for every pattern generated, thus the computation cost is reduced dramatically.

Another method for solving the pivoted graph isomorphism problem is the following. Given two pivoted graphs $\mathcal{G}_1$ and $\mathcal{G}_2$, the algorithm tries to find a bijective function f :

$V_1 \to V_2$ through a recursive backtracking procedure proposed by J. R. Ullmann [31]. For convenience, we refer this algorithm as *direct searching*. This is the algorithm used by Han and Wen [11]. However, in this thesis, we adopt the canonical labeling based algorithm which outperforms the *direct searching* in practice for the following reasons. Firstly, canonical labeling based algorithm concentrates on one graph at a time, powerful ideas from the realm of group theory can be brought to bear on the problem, significantly decreasing the running time [6]. In addition, canonical labeling based algorithm can provide more information than the *direct searching*. For example, it can list all the automorphisms of the pivoted graph.

Algorithm 5 Pattern Extend Algorithm (DFS Code)

Input: s, G, τ, r

Output: $\boldsymbol{S}$

1: **if** $s \neq min(s)$ **then**
2:     return;
3: **end if**
4: $\boldsymbol{S} \leftarrow \boldsymbol{S} \cup \{s\}$
5: Generate all the potential children of s with one edge growth;
6: **for** each child c **do**
7:     **if** $\kappa(c) \leq r$ **and** $\sigma(c) \geq \tau$ **then**
8:         Pattern_Extension(c, G, τ, r)
9:     **end if**
10: **end for**

4.3.2 Pattern Children Generation

According to DFS code's neighborhood restriction stated in Property 4.1, to construct a valid DFS code, backward edges can only grow from the rightmost node and forward edges can only grow from the nodes on the rightmost path. We call it **rightmost extension**. In Algorithm 5, we represent every pattern in the form of its corresponding DFS code. Therefore, the children generation of a pattern in line 5 in fact is DFS code extension. The relation between the parent node and its children complies with Definition 4.7.

Take pivoted graph $\mathcal{P}_{pa}$ in Figure 4.1 as an example. Assume the node label alphabet is $\{A, B\}$. All the possible children generated by $\mathcal{P}_{pa}$ via the rightmost extension are listed in Figure 4.3. The dashed lines indicate their newly grown edges, respectively. The

lexicographic order of the minimum DFS codes of them is (a), (b), (c), (d), (e), (f), (g) increasingly. The precondition that we will extend such a child pattern $\mathcal{P}_c$ of a parent pattern $\mathcal{P}_a$ is that there exists at least one embedding of $\mathcal{P}_c$ in the labeled graph G based on which we are computing support.

Note that, in this thesis, we extend a pattern through a depth first search (DFS) manner. It has two main advantages over the Apriori [2] based breadth first search (BFS) scheme. Firstly, our pattern extension algorithm avoids expensive candidate generation cost by pattern joining operation. Secondly, DFS is more space efficient than BFS, especially when the graph is large.

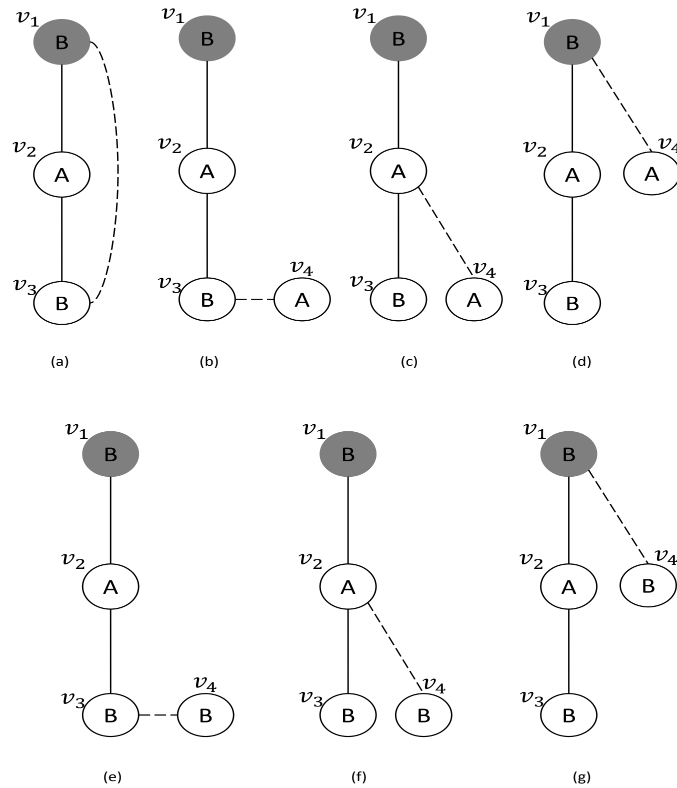


Figure 4.3: All the children of the pivoted graph $\mathcal{P}_{pa}$ in Figure 4.1 by rightmost extension.

Chapter 5

Edge Weight Estimation

Edge weight estimation, or link strength prediction attracts a lot of attention in the areas of graph mining and social network analysis. Methods are proposed from the perspective of studying the correlation between the node profile information and friendship strength, estimating the edge weight by graph structural information such as random walk, personalized PageRank [16], etc. Inspired by the property of homophily [25] in social network, we are interested in the correlation between the local structures of two endpoints of an edge $e = (u, v)$, and the weight of the edge e. To the best of our knowledge, we are the first to pose this problem and estimate continuous edge weights.

The main purpose of this chapter is to connect frequent neighborhood patterns and edge weights. We present it in the following way. We introduce the framework to address this problem in Section 5.1. In Section 5.2 we give the details of how to construct edge features based on frequent neighborhood patterns. We formalize the regression problem in Section 5.3 and lastly we give an overview of the regression model we apply in this thesis.

5.1 Framework

In our problem, edge weights are the target values we predict. Since the edge weight is a continuous variable, given edge features, edge weight estimation is a regression problem. To utilize frequent neighborhood patterns in edge weight estimation, we need to first involve them into the regression model: convert frequent neighborhood patterns as edge features. Below is the framework of studying the effectiveness of frequent neighborhood patterns on edge weight estimation.

1. Convert nodes' frequent neighborhood patterns into structured edge features;

2. Train a regression model with edge features and their corresponding target values;

3. Measure the effectiveness of frequent neighborhood pattern based features on edge weight estimation by comparing the root mean square error (RMSE) of our proposed approach with the RMSEs of three baseline algorithms.

We elaborate on the first two parts of the framework in Section 5.2 and Section 5.3. The third part will be discussed in Chapter 6.

5.2 Edge Feature Construction

For an edge $e = (u, v) \in E$ in a labeled graph G, after we run the pattern mining algorithm described in Chapter 4, we get a list of patterns for node u, denoted by $P(u) = \{\mathcal{P}_i \mid \mathcal{P}_i \subseteq_p (G, u)\}$, and a list of patterns for node v, denoted by $P(v) = \{\mathcal{P}_i \mid \mathcal{P}_i \subseteq_p (G, v)\}$. The question is: how can we construct the edge features of $e = (u, v)$ given $P(u)$ and $P(v)$, and then train a regression model for edge weight estimation? The method we apply in this thesis is discussed below.

Consider a list of patterns $\{\mathcal{P}_j, j = 1...m\}$ mined by the frequent neighborhood pattern mining algorithm described in Chapter 4, where m is the total number of frequent neighborhood patterns. Each $\mathcal{P}_j$ describes the local structure of its pivot node in G. We represent each node $u \in V$ as an $m-$dimensional vector $\mathcal{N}_u = (n_1, n_2, ..., n_i, ..., n_m)$,

$$n_i = \begin{cases} 1, & \text{if } \mathcal{P}_i \subseteq_p (G, u) \\ 0, & \text{otherwise} \end{cases} \tag{5.1}$$

where $1 \leq i \leq m$.

Based on $\mathcal{N}_u$, we represent each edge $e = (u, v) \in E$ as a $2m-$dimensional vector $\boldsymbol{x}_e = (\boldsymbol{x}_{e_1}, \boldsymbol{x}_{e_2})$, where $\boldsymbol{x}_{e_1} = (x_1^1, ..., x_k^1, ..., x_m^1)$ and $\boldsymbol{x}_{e_2} = (x_1^2, ..., x_k^2, ..., x_m^2)$. Let

$$x_k^1 = \begin{cases} 1, & \text{if } \mathcal{N}_u(k) \wedge \mathcal{N}_v(k) = 1 \\ 0, & \text{otherwise} \end{cases} \tag{5.2}$$

$$x_k^2 = \begin{cases} 1, & \text{if } \mathcal{N}_u(k) \vee \mathcal{N}_v(k) = 1 \\ 0, & \text{otherwise} \end{cases} \tag{5.3}$$

where $1 \leq k \leq m$.

$\boldsymbol{x}_{e_1}$ is the result of logic operation AND on $\mathcal{N}_u$ and $\mathcal{N}_v$. $\boldsymbol{x}_{e_2}$ is the result of logic operation OR on $\mathcal{N}_u$ and $\mathcal{N}_v$. By combining $\boldsymbol{x}_{e_1}$ and $\boldsymbol{x}_{e_2}$, we get the edge feature vector $\boldsymbol{x}_e$. For convenience, we call this edge feature construction method *Pattern based Feature Construction* (PFC). The features are called PFC features.

Example 5.1 (Edge Feature Construction)**.** *Given an edge* $e = (u, v)$*, suppose* $\mathcal{N}_u = (1, 1, 0, 1, 0)$ *and* $\mathcal{N}_v = (0, 1, 1, 1, 0)$*. After applying the PFC method, we get edge e's feature vector* $\boldsymbol{x}_e = (0, 1, 0, 1, 0, 1, 1, 1, 1, 0)$*.*

Property 5.1. *Consider two edges* $e = (u, v)$ *and* $e' = (v, u)$*. The PFC feature vectors of* e *and* e' *are the same.*

Proof. Let $\mathcal{N}_u$ and $\mathcal{N}_v$ denote the corresponding node features of u and v, respectively. According to the PFC feature construction method, we have $\boldsymbol{x}_{e_1} = \mathcal{N}_u \wedge \mathcal{N}_v$ and $\boldsymbol{x}_{e_2} = \mathcal{N}_u \vee \mathcal{N}_v$. Then we switch the order of N_u and N_v to compute $\boldsymbol{x}'_{e_1}$ and $\boldsymbol{x}'_{e_2}$. Now we have $\boldsymbol{x}'_{e_1} = \mathcal{N}_v \wedge \mathcal{N}_u$ and $\boldsymbol{x}'_{e_2} = \mathcal{N}_v \vee \mathcal{N}_u$. Because the operations logical AND $\wedge$ and logical OR $\vee$ are commutative, which means $\boldsymbol{x}_{e_1} = \boldsymbol{x}'_{e_1}$ and $\boldsymbol{x}_{e_2} = \boldsymbol{x}'_{e_2}$. Therefore $\boldsymbol{x}_e = \boldsymbol{x}_{e'}$. □

From Property 5.1, we can see that the PFC features of an edge $e = (u, v)$ are insensitive to the order of the feature vectors of u and v. In this thesis, we deal with undirected edges. Property 5.1 shows that the PFC method works for undirected edges.

We also consider other alternative approaches to construct edge features.

- Given two feature vectors of two endpoints, i.e., $\mathcal{N}_u$ and $\mathcal{N}_v$, we construct the corresponding edge features by combining them into one vector. Specifically, let the corresponding edge feature vectors be $\boldsymbol{x}_{e1} = (\mathcal{N}_u, \mathcal{N}_v)$, $\boldsymbol{x}_{e2} = (\mathcal{N}_v, \mathcal{N}_u)$. However, this method does not work for undirected edges since it implies the directions of edges. In other words, the feature vectors are different when we switch the order of $\mathcal{N}_u$ and $\mathcal{N}_v$.

- For each dimension of the feature vector $\mathcal{N}_u$, we use the number of times that the ith pattern matches node u in a labeled graph G.

  For example, in Figure 5.1, $\mathcal{P}$ represents a frequent neighborhood pattern, and the pivot is the node in grey. G is the labeled graph. Let u denote the node with label X in G. $\mathcal{P}$ matches u in G for 8 times because every node with label Y together with u constitute a match between $\mathcal{P}$ and u in G. So the ith dimension of $\mathcal{N}_u$ is 8.

The advantage of the count based features is that they carry richer semantics than the existence based features. However, the computation cost becomes much higher. In order to compute the number of patterns, we need to enumerate all the matches of a pattern which is extremely time-consuming. On the contrary, if we only compute the existence of a pattern, no extra effort is needed. We can mark whether a node in the labeled graph matches a certain pattern when we compute the support of the pattern.

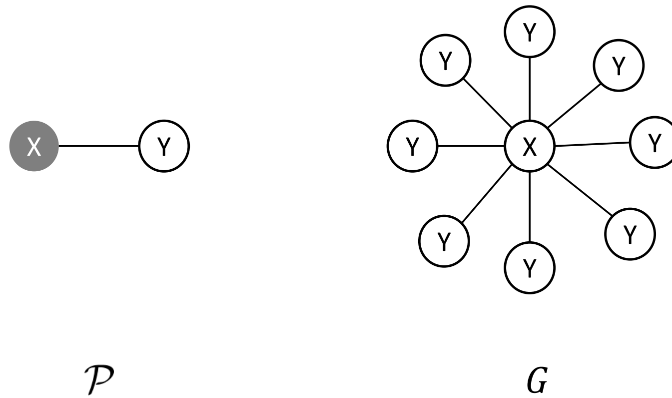


Figure 5.1: An example of counting the number of pattern matches.

We leave other effective feature construction methods as future work.

5.3 Regression Problem Formulation

With the PFC features, we formalize the regression problem of edge weight estimation as follows.

Input: $\mathcal{D}_{train} = \{(\boldsymbol{x}_i, y_i) \mid i = 1...n\}$, where $\boldsymbol{x}_i$ is the feature vector of the ith edge in G, y_i is the target value, i.e., edge weight, and n is the size of the training data. And $\mathcal{D}_{test} = \{(\boldsymbol{x}_i, y_i) \mid i = 1...k\}$, where k is the size of the test data. $\mathcal{D}_{train}$ and $\mathcal{D}_{test}$ are disjoint.

Output : the RMSE on $\mathcal{D}_{test}$.

Let $\hat{\boldsymbol{y}}$ denote the estimated edge weights by our regression model. Below is the equation

to compute the RMSE.

$$RMSE = \sqrt{\frac{1}{k}\sum_{i=1}^{k}(\hat{y}_i - y_i)^2} \tag{5.4}$$

In this thesis, we use gradient boosting regression [7]. The reasons are: (1) the training data is a sparse 0-1 matrix, boosting tree is more effective than the general linear model [23] to learn the higher order interactions between features and, (2) gradient boosting is more efficient than other regression models which can also learn complicated relations from data such as SVM [29] and neural networks [27] based regression.

In this thesis, we adopt the implementation of gradient boosting tree regression called *XGBoost* [1]. We give an overview of gradient boosting regression in *XGBoost* in the next section.

5.4 Gradient Boosting Overview

Gradient boosting was first introduced by Friedman [7] in 2001. It is one of the techniques which aim to improve the regression performance of a single model by fitting multiple models and combining them for prediction. It produces a prediction model in the form of an ensemble of weak models. Gradient boosting is usually used with decision trees [3].

5.4.1 The Gradient Boosting Model

The prediction model is shown as follows.

$$\hat{y}_i = \sum_{t=1}^{T} f_t(\boldsymbol{x}_i), \quad f_t \in \mathcal{F} \tag{5.5}$$

where T is the number of trees, $\mathcal{F}$ is the space of functions which contains all the decision trees, and each f_t is a decision tree which maps attributes to a regression score.

In order to learn $\mathcal{F}$, a regularized objective function $\mathcal{L}$ is defined.

$$\mathcal{L} = \sum_{i=1}^{n} l(y_i, \hat{y}_i) + \sum_{t=1}^{T} \Omega(f_t) \tag{5.6}$$

[1] http://github.com/tqchen/xgboost

where l is a differentiable convex loss function which measures the difference between the predicted $\hat{\boldsymbol{y}}$ and target $\boldsymbol{y}$. In our problem, l is square loss defined as follows.

$$l(\hat{y}_i, y_i) = (\hat{y}_i - y_i)^2 \tag{5.7}$$

The second term Ω is a function of tree, which controls the complexity of the model to avoid overfitting. Ω is defined in Equation 5.8.

$$\Omega(f_t) = \gamma N + \frac{1}{2}\lambda \sum_{j=1}^{N} w_j^2 \tag{5.8}$$

N indicates the number of leaves in a tree f_t, and the second part in Equation 5.8 is the L_2 norm of leaf score which is denoted by w_j. γ and λ are two regularization parameters. They will be explained in Subsection 5.4.2.

Therefore, a model employing simple and predictive functions will be selected as the best model. Generally, the gradient boosting algorithm adds functions to minimize Equation 5.6 in T rounds iteratively, where T is user specified.

The algorithm to learn $\mathcal{F}$ is shown in Algorithm 6. At first, it initialize the tree space $\mathcal{F}$ as an empty set. Then it adds T trees to $\mathcal{F}$ in T rounds (line 2 - line 7). In order to get the tree f_t in round t, there are two steps (line 3).

1. Grow a tree f'_t.

   f'_t is a full binary tree with K levels. Consider a node in level k of f'_t, where $k < K$, denoted by l_e. Let $gain_j$ denote the maximum gain obtained by splitting the node l_e on the jth feature, where $1 \leq j \leq d$, and d is the number of features. In order to compute the $gain_j$ for the jth feature, the splitting criterion on the jth feature is needed. In *XGBoost*, it first sorts the dataset according to the value of the jth feature, then enumerates the splitting criterion which partitions the dataset into two subsets. Let $\mathcal{D}'_j$ denote the sorted dataset according to the value of the jth feature. Assume the size of $\mathcal{D}'_j$ is n. There are $n-1$ possible splitting criteria which result in $n-1$ kinds of different partitions of $\mathcal{D}'_j$. For the sth partitioning, we use $\mathcal{D}'_{js1}$ and $\mathcal{D}'_{js2}$ to denote the two subsets, where $1 \leq s \leq n-1$. $\mathcal{D}'_{js1}$ is assigned to l_e's left child, and $\mathcal{D}'_{js2}$ is assigned to l_e's right child after splitting. And we use I_{L_s} and I_{R_s} to denote the corresponding indices of the data points in $\mathcal{D}'_{js1}$ and $\mathcal{D}'_{js2}$, respectively. The gain of splitting the node l_e on the jth feature with the sth partitioning on the

j^{th} feature is computed according to the Equation 5.9. Then we can compute $gain_j$ by $gain_j = \max\{gain(j,s) \mid 1 \leq s \leq n-1\}$.

We choose the jth feature to split l_e if $j = \arg\max_j\{gain_j \mid 1 \leq j \leq d\}$.

$$gain(j,s) = \frac{1}{2}\left[\frac{(\sum_{i\in I_{L_s}} g_i)^2}{\sum_{i\in I_{L_s}} h_i + \lambda} + \frac{(\sum_{i\in I_{R_s}} g_i)^2}{\sum_{i\in I_{R_s}} h_i + \lambda} - \frac{(\sum_{i\in I} g_i)^2}{\sum_{i\in I} h_i + \lambda}\right] - \gamma \tag{5.9}$$

where I is the set of indices of $\mathcal{D}'_j$, and the relation among I_{L_s}, I_{R_s} and I is $I = I_{L_s} \cup I_{R_s}$.

2. Prune the tree f'_t.

   The pruning is done in a bottom up way. More specifically, given a non-leaf node, if the maximum gain obtained by splitting that node in step 1 is negative, we get rid of that splitting. Then that non-leaf node becomes a leaf node. The pruned tree is denoted by f_t.

Then we use f_t to update the predicted targets $\hat{\boldsymbol{y}}$ (line 5). At last, the tree f_t will be added to the tree space $\mathcal{F}$ (line 6).

Algorithm 6 Gradient Boosting Model Learning Algorithm

Input: $\mathcal{D}$: dataset, K: maximum depth, λ, γ: regularization parameters, η: step size, l: loss function, g: first order derivative of l, h: second order derivative of l

Output: $\mathcal{F}$: the set of decision trees

1: $\mathcal{F} \leftarrow \emptyset$
2: **for** t in 1 to T **do**
3: Greedily grow a decision tree f'_t to the maximum depth K that maximize the gain for each split by Equation 5.9
4: Calculate the pruned tree f_t by eliminating splits in f'_t with negative gain
5: $\hat{\boldsymbol{y}}_t \leftarrow \hat{\boldsymbol{y}}_t + \eta \cdot f_t(\mathcal{D}_{train})$
6: $\mathcal{F} \leftarrow \mathcal{F} \cup f_t$
7: **end for**
8: **return** $\mathcal{F}$

5.4.2 Model Parameters

In this model, there are several key parameters which are listed in Table 5.1.

Table 5.1: Parameters of the Gradient Boosting Tree Regression Model

Parameter	Interpretation
η	step size shrinkage used in update to prevents overfitting.
K	maximum depth of a tree.
λ	L2 regularization term on weights.
γ	minimum loss reduction required to make a further partition on a leaf node of the tree.
T	number of trees

5.4.3 Time Complexity

Assume the size of the training data is n, the number of features is d, and the maximum depth of a tree is K. We compute the time complexity of growing a tree in Algorithm 6 as follows. Consider the current leaf level k of a tree, where $0 \leq k \leq K-1$. Since the tree is a full binary tree, the number of leaves in level k is 2^k. Let m_i denote the number of data points assigned to the ith leaf in level k. According to Algorithm 6, the time cost for splitting each leaf node will be $d \cdot m_i \cdot \log m_i$, which is $O(d \cdot m_i \log n)$ because $n > m_i$. And the time cost for the splitting of all the 2^k nodes in level k is $d \sum_{i=1}^{2^k} m_i \log m_i$, which is $O(nd \log n)$. This tree has K levels, thus the time complexity of growing a tree is $O(ndK \log n)$.

Chapter 6

Experiments

In this chapter, we report the experimental results of (1) the efficiency of our proposed pattern growth based frequent neighborhood pattern mining algorithm and, (2) the effectiveness of frequent neighborhood pattern based features on edge weight estimation.

6.1 Environments and Datasets

The pattern mining algorithm is implemented in C#. The regression model for edge weight estimation is implemented in R. All experiments are conducted on a PC computer with Intel Core i7-3770 3.40 GHz CPU and 16GB main memory running 64-bit Microsoft Windows 7.

In this thesis, we conduct the experiments on two real datasets. They are the DBLP bibliography dataset and the Tencent weibo dataset. Details about these datasets are shown in Table 6.1 and Table 6.2, respectively.

For the DBLP dataset, the node label alphabet is $\Sigma_V = \{Author, Paper, Conference\}$. When estimating edge weights, we only consider the edges formed by two nodes with node label $Author$. The reason is that such edges convey meaningful edge weights. For example, the weight of the edge that connects two authors represents how many papers these two authors have co-authored. However, other edges such as an edge formed by a $Paper$ node and a $Conference$ node does not have meaningful weights, so we ignore them.

For the Tencent weibo dataset, all nodes are users. The node label alphabet is $\Sigma_V = \{Unknown, Female, Male\}$. The edge weight is defined as the number of interactions between two users. Interactions include re-tweet, comment, and mention. Intuitively, the weight is used to measure how close two users are. Note that in our experiments, we only

consider the bi-followed user pairs, i.e., two users follow each other.

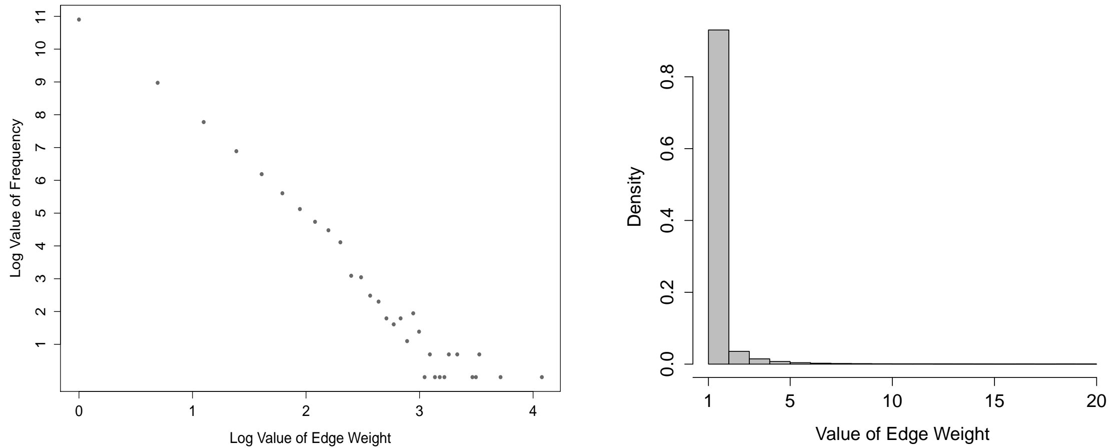
Table 6.1: DBLP Dataset Statistics

ID	#Nodes			#Edges
1	Author	Paper	Conference	103201
	28702	28569	20	

Table 6.2: Tencent Weibo Dataset Statistics

ID	#Nodes			#Edges
2	Male	Female	Unknown	156895
	14576	8444	286	

The values of edge weights are very skewed. In the DBLP dataset, most (81.2%) edge weights are 1. Figure 6.1a shows the weight distribution in a log-log scale, which indicates the weight distribution has a long tail. And Figure 6.1b shows the histogram of the edge weights of the DBLP dataset. Note that, for better visualization, we plot Figure 6.1b by using the edge weights which are not greater than 20 (99.9% of the whole DBLP dataset).



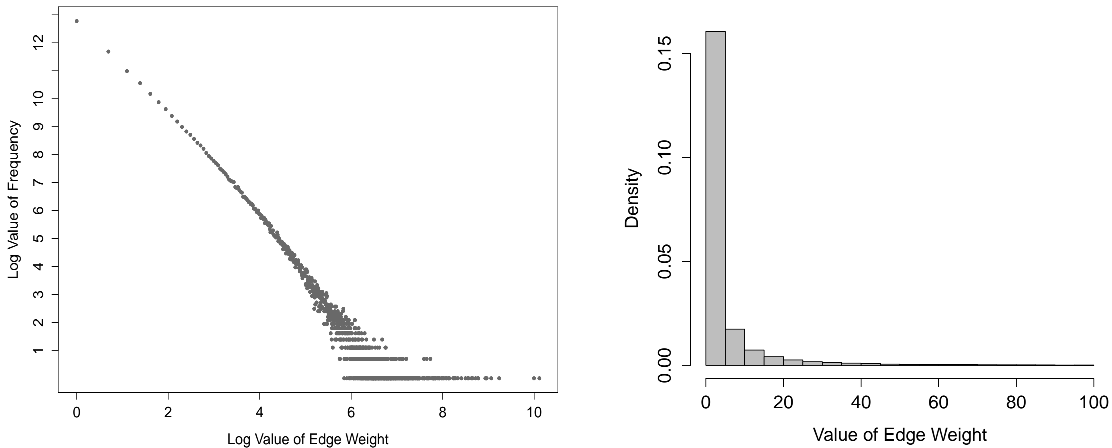
(a) Log-log scale of edge weight and its frequency.

(b) The histogram of edge weights.

Figure 6.1: Edge weight distribution of the DBLP dataset.

The edge weights are also skewed in the Tencent weibo dataset. Figure 6.2a shows the log-log scale edge weight distribution, which indicates a long tail distribution as well. Figure

6.2b shows the histogram of the edge weights of the Tencent weibo dataset. Note that, for better visualization, we plot Figure 6.2b by using the edge weights which are not greater than 100 (99.1% of the whole Tencent weibo dataset).



(a) Log-log scale of edge weight and its frequency. (b) The histogram of edge weights

Figure 6.2: Edge weight distribution of the Tencent weibo dataset.

6.2 Baseline Algorithms

In order to measure the efficiency of our proposed frequent neighborhood pattern mining algorithm, we compare our running time with that of the Apriori based pattern mining algorithm [11].

The problem of continuous edge weight estimation in heterogeneous network scenario has never been studied by literature, so there is no existing baseline algorithms. In this thesis, considering the extremely skewed edge weights, we propose three baseline algorithms listed below. In order to measure the effectiveness of frequent neighborhood pattern based features on edge weight estimation, we compare the RMSE of our model with the RMSEs of these baseline estimators. Let $\boldsymbol{y}$ denote the target ground truth, i.e., the edge weight vector in our regression problem. And $\hat{y}_1$, $\hat{y}_2$ and $\hat{y}_3$ in Equations 6.1, 6.2, and 6.3 represent the results of three baseline algorithms, respectively.

- **Mean**: take the mean value from the training target values as the estimated edge

weight on test data.

$$\hat{y}_1 = \frac{1}{n}\sum_{i=1}^{n} y_i \tag{6.1}$$

where $n = |\boldsymbol{y}|$.

- **Median**: take the median from the training target values as the estimated edge weight on test data.

$$\hat{y}_2 = \text{median of } \boldsymbol{y} \tag{6.2}$$

- **Mode**: take the mode from the training target values a as the estimated edge weight on test data.

$$\hat{y}_3 = \arg\max_{y_i}\{\text{frequency of } y_i \mid i = 1, \ldots, n\} \tag{6.3}$$

where $n = |\boldsymbol{y}|$.

Due to the skewness of edge weights, the baseline estimators are actually quite effective. Section 6.4 will show that the model trained from the frequent neighborhood pattern based features can get better performance on edge weight estimation than all these three baseline algorithms. Before we show the effectiveness of neighborhood pattern based features, we show the efficiency of our pattern mining algorithm in Section 6.3 first.

6.3 Efficiency of Our Pattern Mining Algorithm

In this section, we report the performance of our pattern growth based frequent neighborhood pattern mining algorithm compared with the Apriori based algorithm [11] on the DBLP and Tencent weibo datasets. Generally, the running time of the frequent pattern mining algorithm is positively correlated to the size of result set, that is the larger the number of frequent neighborhood patterns, the longer the running time.

In the frequent neighborhood pattern mining problem, we have two parameters: support threshold τ and pattern size r. In our experiments, we choose τ from $\{0.005, 0.01, 0.05, 0.1\}$ and r from $\{1, 2, 3, 4, 5\}$. For instance, for a dataset with n nodes, when we choose $\tau = 0.01$, the support of a frequent pattern should be at least $0.01n$.

Experiments on the running time test show that our pattern growth based algorithm is up to 22 times faster than the Apriori based algorithm [11] on both datasets.

DBLP Dataset. We test the running time of our algorithm with parameters τ from $\{0.005, 0.01, 0.05, 0.1\}$ and r from $\{1, 2, 3, 4, 5\}$.

Figure 6.3 shows the number of patterns under different supports and with different pattern sizes. From this figure we can see that with the increase of pattern size, the number of patterns increases. With a fixed pattern size, the number of patterns becomes larger when the support threshold is smaller.

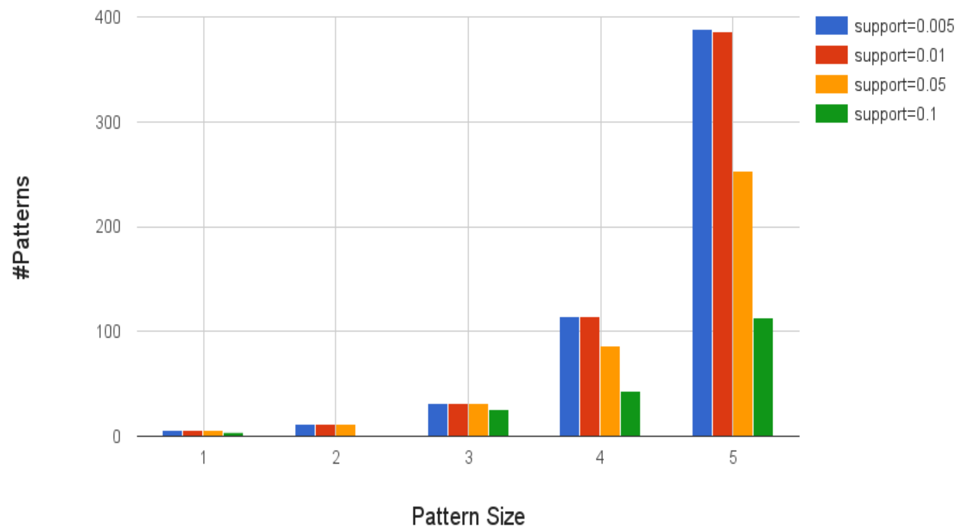
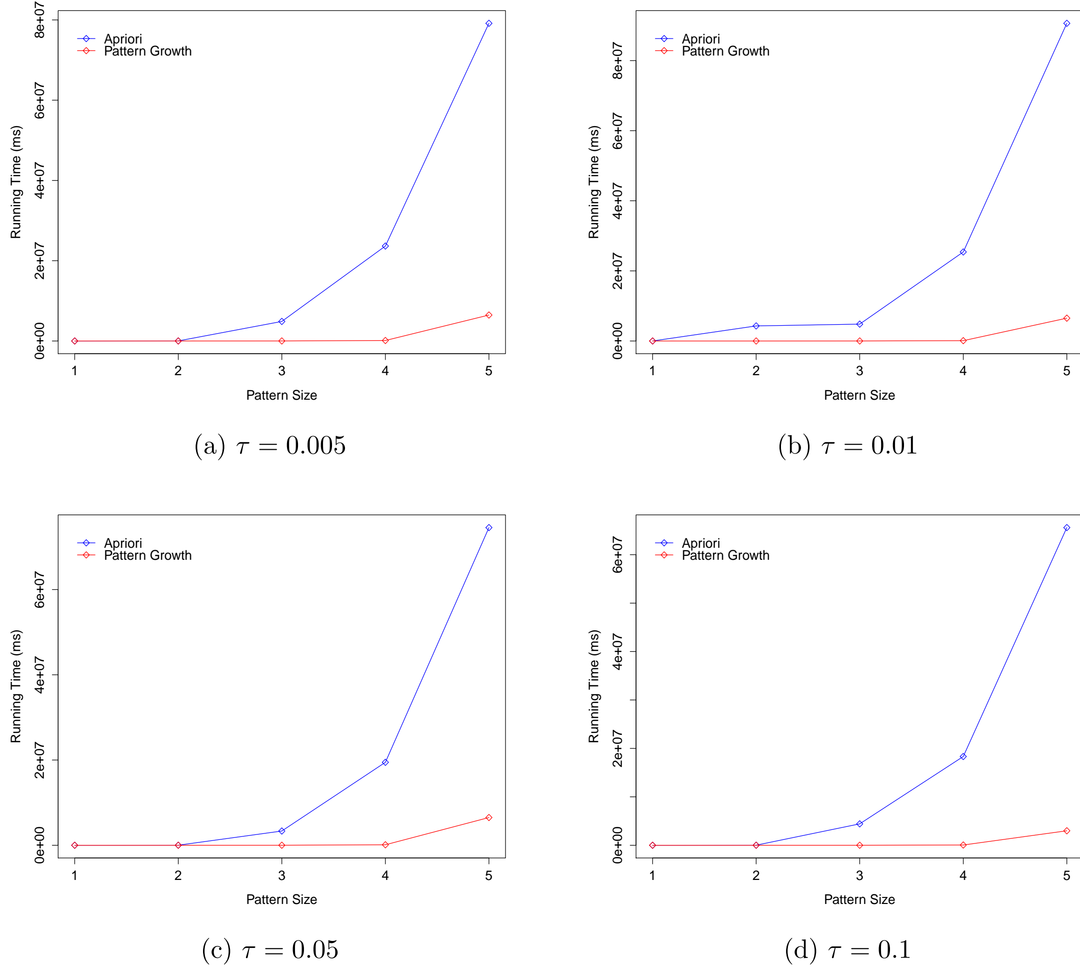


Figure 6.3: The number of patterns under different supports and pattern sizes on the DBLP dataset.

Figure 6.4 shows the efficiency of our pattern growth based mining algorithm compared with the Apriori based mining algorithm [11]. From this figure, we can see that the advantage of our algorithm becomes more pronounced when the pattern size threshold gets larger. That is because when the size increases, the number of frequent neighborhood pattern candidates becomes larger. Therefore, more computation is needed to do expensive pivoted graph isomorphism test and support computing.



(a) $\tau = 0.005$

(b) $\tau = 0.01$

(c) $\tau = 0.05$

(d) $\tau = 0.1$

Figure 6.4: Running time comparison between our algorithm and Apriori base algorithm under 4 support threshold values on the DBLP dataset. In each subfigure, the curves show the running time under different pattern size constraints.

Tencent weibo Dataset. Figure 6.5 shows the number of patterns under different support thresholds and with different pattern size constraints. It has similar trends on the number of patterns to Figure 6.3 of the DBLP dataset. The running time comparison results of frequent neighborhood pattern mining on the Tencent weibo dataset shown in Figure 6.6 also demonstrate that our algorithm outperforms the Apriori based algorithm [11] noticeably.

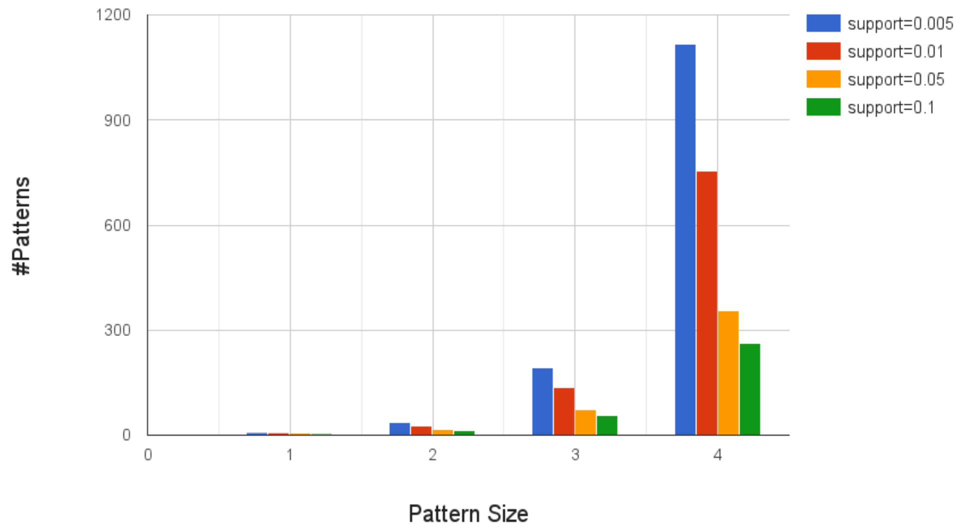
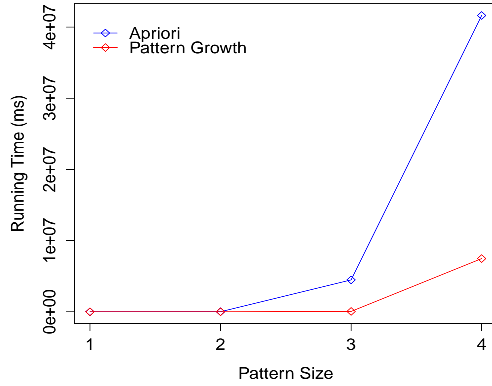
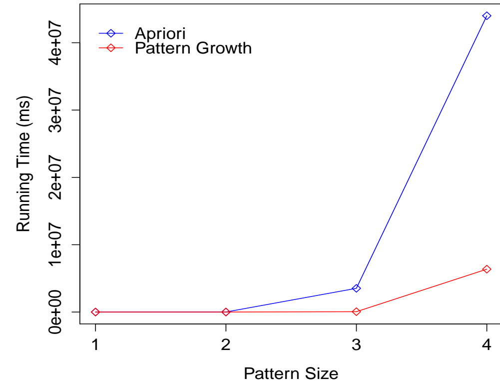


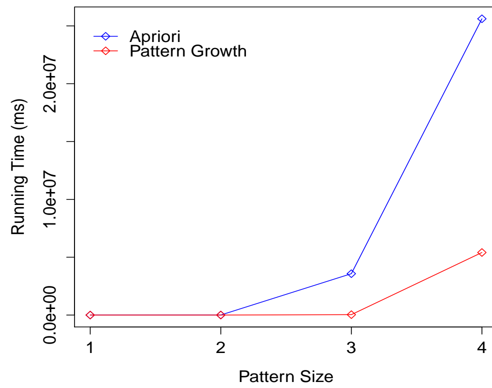
Figure 6.5: The number of patterns under different supports and pattern sizes on the Tencent weibo dataset.



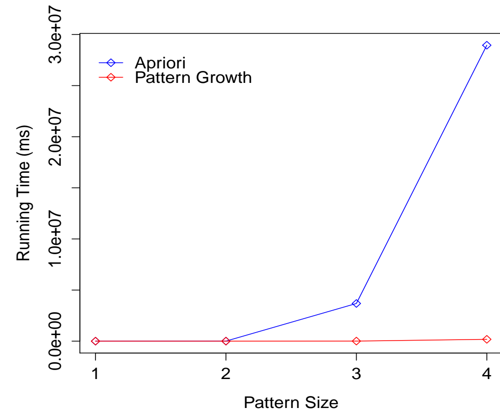
(a) $\tau = 0.005$



(b) $\tau = 0.01$



(c) $\tau = 0.05$



(d) $\tau = 0.1$

Figure 6.6: Running time comparison between our algorithm and Apriori base algorithm under 4 support threshold values on the Tencent weibo dataset. In each subfigure, the curves show the running time under different pattern size constraints.

Analysis. The experimental results on the running time test on both datasets show that our pattern growth based mining algorithm outperforms the Apriori based pattern mining algorithm [11] greatly. We summarize the reasons as follows.

1. Candidate generation method. Our algorithm generates a size k pattern by adding one edge to a size $k-1$ pattern directly. However, the Apriori based algorithm generates a size k pattern by joining two size $k-1$ patterns. The pattern join operation is costly whereas our algorithm avoids that.

2. Pivoted graph isomorphism test. The Apriori based algorithm conducts a pivoted graph isomorphism test by the *direct searching* [31] algorithm. Our algorithm applies the *minimum DFS encoding* in the pivoted graph isomorphism test, which was empirically shown to be more efficient than the *direct searching* by Yan and Han [34].

3. False positive patterns. The Apriori based algorithm generates more false positive pattern candidates than those of our algorithm. Therefore, it needs more computation on checking the support to filter out these false positives. However, we apply the minimum DFS code based pruning techniques described in Chapter 4 in the pattern generation step, which leads to less false positives. So that a lot of computation is saved.

6.4 Effectiveness of Frequent Neighborhood Pattern Features

After mining the frequent neighborhood patterns, we construct edge features based on them and train the regression model to estimate edge weights. In this thesis, we use gradient boosting regression model. We adopt an implementation of the gradient boosting regression in R, called *XGBoost*[1].

First of all, we study the problem: *how do the support threshold τ and the pattern size r affect the effectiveness of edge weight estimation?* Table 6.3 shows the RMSEs of different combinations of τ and r on the DBLP dataset. All the related model parameters in *XGBoost* are set the same as they are shown in Table 6.5 for the DBLP dataset. In Table 6.3, the first row indicates the four different support thresholds, and the first column indicates five different pattern sizes. Every other cell is a tuple of two float numbers. The first float

[1]http://github.com/tqchen/xgboost

number is the RMSE on the training data, and the second one is the RMSE on the test data. Note that we compute the RMSE in the following way. Firstly, we split the dataset into 5 folds, every time we take 4 of them as the training dataset, and the rest one fold as the test dataset. The final RMSE is the average of these 5 RMSEs.

Figure 6.3 shows that with a fixed τ, the larger the pattern size constraint r, the larger the number of frequent neighborhood patterns. And when we fix r, the smaller the support threshold τ, the larger the number of frequent neighborhood patterns. Therefore, in Table 6.3, the upper left cell ($r = 1$, $\tau = 0.1$) represents the RMSEs from the model trained from the smallest amount of features, and the lower right cell ($r = 5$, $\tau = 0.005$) represents the RMSEs from the model trained from the largest amount of features.

From Table 6.3, we get the following three observations.

- In each row from left to right, with the decrease of the support threshold τ, the number of frequent neighborhood patterns increases, and the RMSEs decrease on both training and test data.
- In each column from top to bottom, with the increasing of the pattern size r constraint, the number of frequent neighborhood patterns increases, and the RMSEs decrease on both training and test data.
- The largest RMSE is in the upper left cell which corresponds to the model with the smallest amount of features, and the smallest RMSE is in the lower right cell which corresponds to the model with the largest amount of features.

Table 6.3: The Effect of τ and r on Estimation Performance on the DBLP Dataset

r/τ	0.1	0.05	0.01	0.005
1	(0.4107741, 0.4107398)	(0.4107772, 0.4107646)	(0.4107772, 0.4107646)	(0.4107772, 0.4107646)
2	(0.3589964, 0.3590526)	(0.3519334, 0.3528697)	(0.3519334, 0.3528697)	(0.3519334, 0.3528697)
3	(0.3519448, 0.3528679)	(0.3440261, 0.3486310)	(0.3440261, 0.3486310)	(0.3440261, 0.3486310)
4	(0.3248251, 0.3289706)	(0.3050721, 0.3244362)	(0.2879135, 0.3202999)	(0.2879135, 0.3202999)
5	(0.3137784, 0.3248358)	(0.2687392, 0.3107534)	(0.2495713, 0.3044902)	(0.2503219, 0.3051163)

We get the same observations from the edge weight estimation experiments on the Tencent weibo dataset. The results are shown in Table 6.4. All the related model parameters in *XGBoost* are set the same as they are in Table 6.5 for the Tencent weibo dataset.

We summarize the effects of the support threhold τ and the pattern size r on the performance of edge weight estimation as follows.

Table 6.4: The Effect of τ and r on Estimation Performance on the Tencent Weibo Dataset

r/τ	0.1	0.05	0.01	0.005
1	(1.1220802, 1.1222688)	(1.1214886, 1.1218813)	(1.1184574, 1.1192905)	(1.1172602, 1.1181911)
2	(1.1126835, 1.1199483)	(1.0972538, 1.1112949)	(1.0765263, 1.1070381)	(1.0875048, 1.1047293)
3	(1.1139993, 1.1187700)	(1.0959688, 1.1044453)	(1.0857054, 1.0990376)	(1.0159790, 1.0819079)
4	(1.0254504, 1.0984213)	(0.9773067, 1.0911211)	(0.9383468, 1.0835034)	(0.9321516, 1.0857181)

1. With a fixed support threshold τ, the larger the pattern size constraint r, the larger amount of pattern based features, and the smaller the RMSE.

2. With a fixed pattern size constraint r, the larger the support threshold τ, the larger amount of pattern based features, and the smaller the RMSE.

Secondly, we compare the edge weight estimation performance of our model with that of the three baseline algorithms.

Considering the expensive time cost on the pattern mining algorithm, we make a trade-off between the number of frequent neighborhood patterns and the richness of semantics carried by the patterns. In other words, we set the pattern size constraint to limit the number of frequent neighborhood patterns. For the DBLP dataset, we choose the frequent neighborhood patterns mined with parameters $\tau = 0.005$ and $r = 5$. In total, we get 552 frequent neighborhood patterns. For the Tencent weibo dataset, we choose the frequent neighborhood patterns mined with parameters $\tau = 0.01$ and $r = 4$. In total, we have 925 frequent neighborhood patterns.

Since the Tencent weibo dataset is large, in order to accelerate the regression training process, we sample 20% edges uniformly to do edge weight estimation experiments. The sample dataset has the same skewed edge weight distribution with the original dataset.

For the two datasets in our experiments, we sample 90% uniformly as the training data and the rest 10% as the test data.

In Table 5.1, we list the related parameters in *XGBoost*. We choose the value of each parameter via a 5-fold cross validation on training data for both datasets. The model parameters used in the regression model training for both datasets are shown in Table 6.5. The first column is the related parameters in this regression model, the second column contains the concrete values we choose for the DBLP dataset, and the third column lists the values chosen for the Tencent weibo dataset.

To validate the effectiveness of the frequent neighborhood pattern based features on

Table 6.5: Parameters of Gradient Boosting in Experiments

Parameters	DBLP	Tencent weibo
η	0.02	0.01
K	10	12
λ	0.1	0.1
γ	0	0
T	300	300

edge weight estimation, we compare the RMSE of our model with the RMSEs of the three baseline algorithms.

The RMSEs on both training and test data are shown in Table 6.6 for the DBLP dataset and in Table 6.7 for the Tencent weibo dataset. From these tables, we can see that our model outperforms the other strong baseline algorithms, which shows the effectiveness of the frequent neighborhood pattern based features.

The difference between the RMSE of the training data and the RMSE of the test data of our model is larger than that of every baseline algorithm. The reason is that each baseline algorithm only captures one statistical feature of the dataset, whereas our model captures a lot more features of the dataset than the baselines. Therefore, the variance of that difference of our model is larger and more sensitive to the dataset.

Table 6.6: RMSE Comparison of Edge Weight Estimation on the DBLP Dataset

	Gradient Boosting	Mean	Median	Mode
Training	**0.2503219**	0.4107773	0.4478321	0.4478321
Test	**0.3051163**	0.4107767	0.4478295	0.4478295

Table 6.7: RMSE Comparison of Edge Weight Estimation on the Tencent Weibo Dataset

	Gradient Boosting	Mean	Median	Mode
Training	**0.9383468**	1.1356146	1.1502391	1.4342084
Test	**1.0835034**	1.1355933	1.1501875	1.4341537

Chapter 7

Conclusions

In this thesis, we formulate the problem of exploring the power of frequent neighborhood patterns on edge weight estimation. To tackle this problem, we propose a two-step method. Firstly, we construct features based on frequent neighborhood patterns of nodes. Secondly, we model edge weight estimation as a regression problem with structural features. In the first step, in order to mine frequent neighborhood patterns from a single large labeled graph efficiently, we devise an efficient pattern growth based mining algorithm. Moreover, we apply *minimum DFS encoding* to our pivoted subgraph isomorphism test, which is one of the key components to make our pattern growth based algorithm efficient. We also discuss different ways of feature construction based on frequent neighborhood patterns and explain why we finally choose the way we used in this thesis.

We empirically show that frequent neighborhood patterns are not only correlated to edge weights but also effective features in estimating edge weights. Experiments conducted on two real datasets verify that our pattern growth based algorithm is much more efficient than the Apriori based pattern mining algorithm [11] as well.

As future work, we can consider the following interesting directions.

- *Do edges have structural patterns?* There may exist better methods to mine edge patterns directly from the graph. In this thesis, we first mine node patterns, then construct edge patterns based on node patterns, which is an indirect method.

- *Can we design a new support measure, which can be efficiently computed, and its corresponding substructure can be well interpreted, and as edge features to estimate edge weights?* Frequent neighborhood patterns are one kind of structural patterns on

edges, more effective patterns and pattern support are awaiting to be discovered.

- *Do frequent neighborhood patterns also work in other applications?* Apply frequent neighborhood patterns to other link related tasks, such as binary link prediction and link sign prediction.

- *Can we apply more heuristics to accelerate the pattern mining algorithm?* Under our current pattern mining algorithm framework, we can develop more heuristics to optimize the proposed algorithm. For example, apply the partial count pruning [34] to the candidate generation step.

Bibliography

[1] F. Abel, N. Henze, E. Herder, and D. Krause. Interweaving public user profiles on the web. In *Proceedings of the 18th International Conference on User Modeling, Adaptation, and Personalization*, UMAP'10, pages 16–27, Berlin, Heidelberg, 2010. Springer-Verlag. 1

[2] R. Agrawal, T. Imieliński, and A. Swami. Mining association rules between sets of items in large databases. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, SIGMOD '93, pages 207–216, New York, NY, USA, 1993. ACM. 2, 5, 28

[3] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. *Classification and Regression Trees*. CRC Press, New York, 1999. 33

[4] B. Bringmann and S. Nijssen. What is frequent in a single graph? In *Advances in Knowledge Discovery and Data Mining*, pages 858–863. Springer, 2008. 6

[5] M. Fiedler and C. Borgelt. Subgraph support in a single large graph. In *Data Mining Workshops, 2007. ICDM Workshops 2007. Seventh IEEE International Conference on*, pages 399–404. IEEE, 2007. 6

[6] S. Fortin. The graph isomorphism problem. Technical report, Technical Report 96-20, University of Alberta, Edomonton, Alberta, Canada, 1996. 27

[7] J. H. Friedman. Greedy function approximation: a gradient boosting machine. *Annals of statistics*, pages 1189–1232, 2001. 2, 33

[8] E. Gilbert and K. Karahalios. Predicting tie strength with social media. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 211–220. ACM, 2009. 1, 7

[9] S. A. Golder, D. M. Wilkinson, and B. A. Huberman. Rhythms of social interaction: Messaging within a massive online network. In *Communities and technologies 2007*, pages 41–66. Springer, 2007. 1

[10] R. Guha, R. Kumar, P. Raghavan, and A. Tomkins. Propagation of trust and distrust. In *Proceedings of the 13th international conference on World Wide Web*, pages 403–412. ACM, 2004. 8

[11] J. Han and J.-R. Wen. Mining frequent neighborhood patterns in a large labeled graph. In *Proceedings of the 22Nd ACM International Conference on Conference on Information Knowledge Management*, CIKM '13, pages 259–268, New York, NY, USA, 2013. ACM. 2, 6, 9, 10, 12, 27, 39, 40, 41, 43, 45, 49

[12] F. Heider. Attitudes and cognitive organization. *The Journal of psychology*, 21(1):107–112, 1946. 8

[13] J. Huan, W. Wang, and J. Prins. Efficient mining of frequent subgraphs in the presence of isomorphism. In *Data Mining, 2003. ICDM 2003. Third IEEE International Conference on*, pages 549–552. IEEE, 2003. 4, 5

[14] B. A. Huberman, D. M. Romero, and F. Wu. Social networks that matter: Twitter under the microscope. *Available at SSRN 1313405*, 2008. 1

[15] A. Inokuchi, T. Washio, and H. Motoda. An apriori-based algorithm for mining frequent substructures from graph data. In *Principles of Data Mining and Knowledge Discovery*, pages 13–23. Springer, 2000. 4, 5

[16] G. Jeh and J. Widom. Scaling personalized web search. In *Proceedings of the 12th International Conference on World Wide Web*, WWW '03, pages 271–279, New York, NY, USA, 2003. ACM. 29

[17] I. Kahanda and J. Neville. Using transactional information to predict link strength in online social networks. In *ICWSM*, 2009. 1, 7

[18] M. Kuramochi and G. Karypis. An efficient algorithm for discovering frequent subgraphs. *Knowledge and Data Engineering, IEEE Transactions on*, 16(9):1038–1051, 2004. 4, 5

[19] M. Kuramochi and G. Karypis. Finding frequent patterns in a large sparse graph*. *Data mining and knowledge discovery*, 11(3):243–271, 2005. 6

[20] J. Leskovec, D. Huttenlocher, and J. Kleinberg. Predicting positive and negative links in online social networks. In *Proceedings of the 19th International Conference on World Wide Web*, WWW '10, pages 641–650, New York, NY, USA, 2010. ACM. 8

[21] J. Leskovec, D. Huttenlocher, and J. Kleinberg. Signed networks in social media. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 1361–1370. ACM, 2010. 8

[22] D. Liben-Nowell and J. Kleinberg. The link-prediction problem for social networks. *Journal of the American society for information science and technology*, 58(7):1019–1031, 2007. 8

[23] K. Mardia, J. Kent, and J. Bibby. *Multivariate analysis*. Probability and mathematical statistics. Academic Press, 1979. 2, 33

[24] B. D. McKay et al. *Practical graph isomorphism*. Department of Computer Science, Vanderbilt University, 1981. 17

[25] M. McPherson, L. Smith-Lovin, and J. M. Cook. Birds of a feather: Homophily in social networks. *Annual review of sociology*, pages 415–444, 2001. 2, 7, 29

[26] S. Nijssen and J. N. Kok. A quickstart in frequent structure mining can make a difference. In *Proceedings of the Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '04, pages 647–652, New York, NY, USA, 2004. ACM. 4, 5

[27] D. F. Specht. A general regression neural network. *Neural Networks, IEEE Transactions on*, 2(6):568–576, 1991. 3, 33

[28] Y. Sun, R. Barber, M. Gupta, C. C. Aggarwal, and J. Han. Co-author relationship prediction in heterogeneous bibliographic networks. In *Advances in Social Networks Analysis and Mining (ASONAM), 2011 International Conference on*, pages 121–128. IEEE, 2011. 2, 7, 8

[29] J. A. Suykens and J. Vandewalle. Least squares support vector machine classifiers. *Neural processing letters*, 9(3):293–300, 1999. 2, 33

[30] B. Taskar, M.-F. Wong, P. Abbeel, and D. Koller. Link prediction in relational data. In *Advances in neural information processing systems*, page None, 2003. 2

[31] J. R. Ullmann. An algorithm for subgraph isomorphism. *Journal of the ACM (JACM)*, 23(1):31–42, 1976. 27, 45

[32] N. Vanetik, E. Gudes, and S. E. Shimony. Computing frequent graph patterns from semistructured data. In *Data Mining, 2002. ICDM 2003. Proceedings. 2002 IEEE International Conference on*, pages 458–465. IEEE, 2002. 6

[33] R. Xiang, J. Neville, and M. Rogati. Modeling relationship strength in online social networks. In *Proceedings of the 19th international conference on World wide web*, pages 981–990. ACM, 2010. 1, 7

[34] X. Yan and J. Han. gspan: Graph-based substructure pattern mining. In *Data Mining, 2002. ICDM 2003. Proceedings. 2002 IEEE International Conference on*, pages 721–724. IEEE, 2002. 2, 4, 5, 7, 14, 17, 19, 25, 45, 50

[35] S.-H. Yang, A. J. Smola, B. Long, H. Zha, and Y. Chang. Friend or frenemy?: predicting signed ties in social networks. In *Proceedings of the 35th international ACM SIGIR conference on Research and development in information retrieval*, pages 555–564. ACM, 2012. 8