Incremental Graph Pattern based Node Matching

by

Guohao Sun

Master of Computer Science, Soochow University (Suzhou, China), 2015

Bachelor of Computer Science, Soochow University (Suzhou, China), 2013

A thesis submitted in fulfilment of

the requirements for the award of the degree

Doctor of Philosophy

from

Department of Computing

Faculty of Science and Engineering

MACQUARIE UNIVERSITY

Supervisor: Prof. Yan Wang

Associate Supervisor: Prof. Mehmet A. Orgun

Adjunct Supervisor: Dr. Guanfeng Liu

December 2019

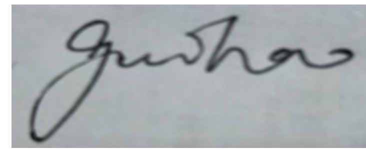
© Copyright by
Guohao Sun
December 2019

Statement of Candidate

I certify that the work in this thesis entitled **"Incremental Graph Pattern based Node Matching"** has not previously been submitted for a degree nor has it been submitted as part of requirements for a degree to any other university or institution other than Macquarie University.

I also certify that the thesis is an original piece of research and it has been written by me. Any help and assistance that I have received in my research work and the preparation of the thesis itself have been appropriately acknowledged.

In addition, I certify that all information sources and literature used are indicated in the thesis.



Guohao Sun

12 December 2019

To my mother, my sister and my wife,

who make me understand the true meaning of love.

In memory of my father,

Daqing Sun.

Abstract

Graph Pattern based Subgraph Matching (GPSM) is used to identify all the matching subgraphs of a pattern graph G_P in a data graph G_D. The existing GPSM solutions are based on subgraph isomorphism or Bounded Graph Simulation (BGS), which aims to find the entire matching subgraphs in G_D. However, in some real applications, such as group finding and expert recommendation, people are more interested in identifying some nodes based on a specified structure between them, leading to the Graph Pattern based Node Matching (GPNM) problem. GPNM aims to find all the matches of the nodes in a data graph G_D based on a given pattern graph G_P.

Firstly, in real scenarios, both G_P and G_D are updated frequently. However, the existing GPNM methods must perform a new GPNM procedure from scratch to deliver the node matching results based on the updated G_P and updated G_D, which consumes significant time. Therefore, there is a pressing need for a method to efficiently deliver the node matching results on the updated graphs. To address this problem, in this thesis, we propose a novel incremental GPNM method called INC-GPNM, where we first build an index to incrementally record the shortest path length range between different label types in G_D, and then identify the affected parts of G_D in GPNM including nodes and edges with respect to the updates of G_P and G_D. Based on the index structure and our novel search strategies, INC-GPNM can efficiently deliver node matching results taking the updates of G_P and G_D as inputs, and can greatly reduce the query processing time with improved time complexity.

Secondly, in some real applications (e.g., social graph searches on Facebook), many typical pattern graphs frequently and repeatedly appear in users' queries in a short period of time. In this situation, it is still time-consuming to apply the incremental GPNM procedure for each of the updates in data graph. To address this problem,

in this thesis, we first analyze the updates in the data graph and find that not all the updates in the data graph essentially affect the GPNM results. Then, we propose the notion of elimination relationships and analyze the elimination relationships between multiple updates in the data graph. In addition, if one update U_a can eliminate the other update U_b, and U_b can eliminate update U_c, there exists the hierarchical structure between these elimination relationships. We further generate an Elimination Hierarchy Tree (EH-Tree) to index the elimination relationships. Based on the EH-Tree, we propose a GPNM method called EH-GPNM, that considers the elimination relationships between multiple updates in the data graph. EH-GPNM can efficiently deliver node matching results when facing frequent and repeated pattern graphs with multiple updates in the data graph.

Thirdly, inspired by EH-GPNM, we noted that the elimination relationships not only exist among the updates in the data graph, but are also present among the updates in the pattern graph and even in the cross updates from G_P and G_D. To further improve the GPNM efficiency when both G_P and G_D are updated frequently, in this thesis, we propose a more efficient GPNM method, called UA-GPNM. UA-GPNM first detects the elimination relationships between multiple independent updates in G_P and G_D, and also the cross elimination relationships between the updates from G_P and G_D, then UA-GPNM generates an EH-Tree to index all the elimination relationships. In addition, we also propose a graph partition strategy in UA-GPNM to accelerate the G-PNM procedure. The experiments show that UA-GPNM can achieve better efficiency compared with INC-GPNM and EH-GPNM when facing the updates of G_P and G_D.

All the methods proposed above in this thesis have been validated and evaluated through sufficient experiments and theoretical analysis. The results have demonstrated that the proposed methods significantly outperform the existing work of GPNM.

Acknowledgments

The thesis would not have been accomplished without the efforts of many kind people who unselfishly contributed to my research in one way or another.

First of all, I would like to express my sincere thanks to my supervisors, Prof. Yan Wang, Prof. Mehmet A. Orgun and Dr Guanfeng Liu for their insightful and patient supervision during my PhD journey. Over the past few years, their professional suggestions kept me researching in the correct direction. Working with them let me truly understand the dedication spirit and the rigorous work attitude. It is my great honour to have them as my supervisors at Macquarie University. My academic journey would not have been so rewarding without their kindness and wisdom.

Second, I wish to express my thanks to my colleagues, Dr Bin Ye, Feng Zhu, Qianli Xing, Qi Wang, Jin He, Yan Zhao, Nan Wang, Wenzhuo Song and Dr Shoujing Wang, for their valuable suggestions in my research and their friendly support in my life. I would like to thank Melina Chan, Sylvian Chow, Jackie Walsh, Jo Aboud, Karen Leung and Grace Zhao who have provided the most professional administrative support for all PhD students in the Department of Computing.

Most importantly, I would like to thank my mother and my sister, Shiyin Jing and Lele Sun. Their unconditional love, support and encouragement make me brave enough to pursue my dream. Many thanks to my lovely wife, Qiqi Jiang, for her understanding and support. All their love and inspiration have supported me to accomplish this work.

Publications

This thesis is based on the research work I have performed with the help of my supervisors and other colleagues during my PhD program at the Department of Computing, Macquarie University between 2016 and 2019. Some parts of my research have been published/submitted in the following papers:

[1] **Guohao Sun**, Guanfeng Liu, Yan Wang, Mehmet A. Orgun, Quan Z. Sheng and Xiaofang Zhou: Incremental Graph Pattern based Node Matching with Multiple Updates, IEEE Transactions on Knowledge and Data Engineering (TKDE) (accepted in September 2019) (**CORE**[1] **Rank A**)

[2] **Guohao Sun**, Guanfeng Liu, Yan Wang, Mehmet A. Orgun, and Xiaofang Zhou: Incremental Graph Pattern based Node Matching, 34th International Conference on Data Engineering (IEEE ICDE 2018), pages 281-292 (**research track, acceptance rate 23%, CORE2018 Rank A***)

[3] **Guohao Sun**, Guanfeng Liu, Yan Wang, and Xiaofang Zhou: Updates-Aware Graph Pattern based Node Matching, 36th International Conference on Data Engineering (IEEE ICDE 2020), accepted in February 2020 (**research track, CORE2018 Rank A***)

[1] CORE stands for the Computing Research and Education Association of Australasia (http://www.core.edu.au).

Contents

Abstract **iv**

Acknowledgments **vi**

Publications **vii**

1 Introduction **1**

1.1 Challenges in GPNM 4

1.1.1 Frequently updated graphs 4

1.1.2 Elimination relationships 5

1.1.3 Graph partition 8

1.2 Contributions of the Thesis 8

1.3 Roadmap of the Thesis 10

2 Literature Review **12**

2.1 Graph Pattern based Subgraph Matching (GPSM) 12

2.1.1 Static GPSM 13

2.1.2 Incremental GPSM 24

2.2 Graph Pattern based Node Matching (GPNM) 29

2.2.1 Static GPNM 30

2.3 Conclusion 39

3 Incremental Graph Pattern based Node Matching with Multiple Updates **41**

3.1 Problem Definition 44

3.1.1 Data Graph, Pattern Graph and BGS 44

3.1.2 Incremental GPNM Problem 45

3.2 INC-GPNM Algorithm . . . 46
3.2.1 INC-GPNM Overview . . . 46
3.2.2 Index Generation and INC-GPNM for $\triangle G_P$ . . . 47
3.2.3 Index Generation and INC-GPNM for $\triangle G_D$ . . . 55
3.3 Experiments on INC-GPNM . . . 61
3.3.1 Experimental Setting . . . 61
3.3.2 Experimental Results and Analysis . . . 64
3.4 Conclusions . . . 67

4 Incremental Graph Pattern based Node Matching Considering the Elimination Relationships Among Updates in Data Graphs 69

4.1 Elimination Relationships in Data Graphs . . . 73
4.1.1 Elimination Relationship Types in Data Graphs . . . 73
4.1.2 Detecting Elimination Relationships in Data Graphs . . . 76
4.2 Elimination Hierarchy Tree (EH-Tree) for the Updates in Data Graphs 80
4.2.1 Identifying the Affected Nodes . . . 80
4.2.2 EH-Tree For the Updates in Data Graphs Establishment . . . 82
4.2.3 EH-Tree For the Updates in Data Graphs Maintenance . . . 83
4.3 EH-GPNM Algorithm . . . 87
4.3.1 Algorithm Overview . . . 87
4.3.2 The Process of EH-GPNM . . . 87
4.4 Experiments on EH-GPNM . . . 89
4.4.1 Experimental Setting . . . 89
4.4.2 Experimental Results and Analysis . . . 91
4.5 Conclusion . . . 95

5 Incremental Graph Pattern based Node Matching Considering the Elimination Relationships Among Updates in Both Pattern Graphs and Data Graphs 96

5.1 Elimination Relationships in Both Pattern Graphs and Data Graphs . . 99

5.1.1 Elimination Relationship Types in Both Pattern Graphs and Data Graphs . . . 99
5.1.2 Detecting Elimination Relationships in Both Pattern Graphs and Data Graphs . . . 101
5.1.3 EH-Tree for the Updates in Both Pattern Graphs and Data Graphs 107
5.2 Graph Partition . . . 108
5.2.1 Label-based Partition . . . 108
5.2.2 Graph Partition based Shortest Path Length Computation . . . 111
5.3 UA-GPNM Algorithm . . . 115
5.4 Experiments on UA-GPNM . . . 115
5.4.1 Experimental Setting . . . 115
5.4.2 Experimental Results and Analysis . . . 118
5.5 Conclusion . . . 121

6 Conclusions and Future Work **122**
6.1 Conclusions . . . 122
6.2 Future Work . . . 125

A The Notations in the Thesis **127**

B The Acronyms in the Thesis **130**

List of Figures

1.1 Subgraph Isomorphism Problem 2
1.2 BGS Problem 3
1.3 The elimination relationships among the updates 6

2.1 An example pattern graph G_P and data graph G_D of Ullmann 14
2.2 A partial search-tree for Ullmann's algorithm. 14

3.1 Incremental GPNM 43
3.2 The average query processing time in CollegeMsg on INC-GPNM . . 64
3.3 The average query processing time in email-Eu-core on INC-GPNM . 64
3.4 The average query processing time in Math Overflow on INC-GPNM 64
3.5 The average query processing time in Ask Ubuntu on INC-GPNM . . 65
3.6 The average query processing time in Super User on INC-GPNM . . . 65
3.7 The average query processing time in Wiki Talk on INC-GPNM . . . 66
3.8 The average query processing time in LiveJournal on INC-GPNM . . 66

4.1 GPNM with multiple updates in data graphs 71
4.2 Elimination Relationship Type I in Data Graphs 74
4.3 Elimination Relationship Type II in Data Graphs 74
4.4 The EH-Tree of Example 4.6 86
4.5 The average query processing time in Ask Ubuntu on EH-GPNM . . . 92
4.6 The average query processing time in Facebook on EH-GPNM 92
4.7 The average query processing time in Super User on EH-GPNM . . . 92
4.8 The average query processing time in Wiki Talk on EH-GPNM 92
4.9 The average query processing time in LiveJournal on EH-GPNM . . . 92

5.1 The elimination relationships among the updates in both G_P and G_D . 97
5.2 The EH-Tree of Example 5.5 108
5.3 Label-based Partition 110
5.4 The average query processing time in email-EU-core on UA-GPNM . 119
5.5 The average query processing time in DBLP on UA-GPNM 119
5.6 The average query processing time in Amazon on UA-GPNM 119
5.7 The average query processing time in Youtube on UA-GPNM 119
5.8 The average query processing time in LiveJournal on UA-GPNM . . . 119

List of Tables

1.1 The GPNM results of Example 1.2 3

1.2 The original GPNM results of Example 1.4 7

3.1 $SLen$ of G_D in Fig. 3.1(c). 48

3.2 R_{SLen} of G_D in Fig. 3.1(c). 49

3.3 The sizes of datasets for INC-GPNM 62

3.4 The average query processing time based on different scales of datasets 65

3.5 The average query processing time based on different scale of G_P . . 66

3.6 The average query processing time based on different scale of $\triangle G_P$. 67

3.7 The average query processing time based on different scale of $\triangle G_D$. 67

4.1 The matching results of the initial query and the subsequent query in Example 4.1 72

4.2 $SLen$ of G_D in Fig. 4.1(c) 78

4.3 $SLen_{new}$ with U_a in Fig. 4.1(a) 79

4.4 $SLen_{new}$ with U_b in Fig. 4.1(b) 79

4.5 $SLen_{new}$ with U_a in Example 4.5 81

4.6 $SLen_{new}$ with U_b in Example 4.5 81

4.7 $SLen_{new}$ with U_c in Example 4.5 81

4.8 $SLen_{new}$ with U_d in Example 4.5 82

4.9 The affected nodes of the updates in Example 4.5 82

4.10 The sizes of datasets on EH-GPNM 89

4.11 The average query processing time based on different datasets on EH-GPNM 93

4.12 Comparison with TopKDAG, INC-GPNM and NEH-GPNM based on different datasets on EH-GPNM 93
4.13 The average query processing time based on different scales of $\triangle G_D$ on EH-GPNM 94
4.14 Comparison with TopKDAG, INC-GPNM and NEH-GPNM based on different scales of $\triangle G_D$ on EH-GPNM 94

5.1 The node matching results of Example 5.1 97
5.2 $SLen$ of G_D in Fig. 5.1(c). 103
5.3 The set of candidate nodes of U_{Pi} 103
5.4 $SLen_{new}$ with U_{D1}. 104
5.5 $SLen_{new}$ with U_{D2}. 105
5.6 The affected nodes of U_{D1} and U_{D2} 105
5.7 The inner bridge nodes and outer bridge nodes of T_{SE} 111
5.8 The shortest path length matrix of P_{SE} 114
5.9 The shortest path length matrix from P_{SE} to P_{TE} 114
5.10 The sizes of datasets for UA-GPNM 117
5.11 The average query processing time based on different datasets for UA-GPNM 118
5.12 Comparison with INC-GPNM, EH-GPNM and UA-GPNM-NoPar based on different datasets for UA-GPNM 120
5.13 The average query processing time based on different scales of $\triangle G$ for UA-GPNM 120
5.14 Comparison with INC-GPNM, EH-GPNM and UA-GPNM-NoPar based on different scales of $\triangle G$ for UA-GPNM 120

A.1 The Notations in Chapter 3 127
A.2 The Notations in Chapter 3 (continued) 128
A.3 The Notations in Chapter 4 128
A.4 The Notations in Chapter 5 129

B.1 The Acronyms in All the Chapters 130

Chapter 1

Introduction

As a popular data model for representing the relations of different data, graphs have been widely used in various fields, such as social networks, social securities, and biology [30, 21, 108, 102, 93]. Graph Pattern based Subgraph Matching (GPSM) is a fundamental problem in graph analysis. GPSM aims to identify all the matching subgraphs of a pattern graph G_P in a data graph G_D. It has been increasingly used in knowledge discovery, traffic network analysis, intelligence analysis, and social network analysis, among other applications [17, 23, 27, 110, 35]. Conventional subgraph matching solutions are based on the *subgraph isomorphism* problem [114, 32], in which matches are strictly based on graph structure. The following example illustrates the *subgraph isomorphism* problem.

Example 1.1 (Subgraph Isomorphism Problem): Fig. 1.1(a) depicts a data graph G_D, where each node denotes a person, labeled with their job title, e.g., *Project Manager* (PM), *Database Developer* (DB), *Software Engineer* (SE), *Test Engineer* (TE), or *Secretary* (S). Each edge indicates a collaboration relationship. A pattern graph G_P is given in Fig. 1.1(b), where a database project requires three types of people, namely: PM, DB and S respectively. A PM needs to connect with an S and a DB, while a DB needs to connect with an S. Based on *subgraph isomorphism*, the matching subgraph G_M is shown in Fig. 1.1(c) because G_M includes the same types of nodes and same structure as G_P.

However, the *subgraph isomorphism* problem is an NP-Complete problem [46],

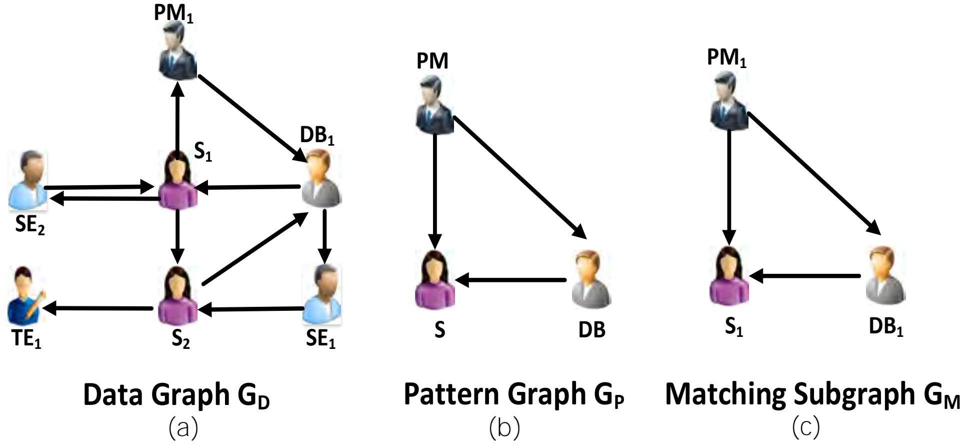


Figure 1.1: Subgraph Isomorphism Problem

which makes it computationally expensive to identify the exact matching subgraphs e-specially in large graphs. To address this problem, Fan et al., proposed *Bounded Graph Simulation* (BGS) [39], which has fewer restrictions but more capacity to extract more useful subgraphs with better efficiency because it supports simulation relations instead of an exact match of edges and nodes. In BGS, each edge in G_P is labeled with either a positive integer k or a symbol "*". k is the constraint of the maximal shortest path length of a match in G_D and "*" indicates that there are no path length constraints. Then, the match of an edge could be a path if the start node and the end node of the path in the data graph have the same labels as the corresponding nodes of the edge in the pattern graph. In social networks, according to the theory of "six degrees of separation" [88], on average, any two people can be connected in about six hops. Therefore, k is usually set as a small integer in social networks [39]. The following example illustrates the BGS problem.

Example 1.2 (BGS Problem): Fig. 1.2(a) depicts a data graph G_D, which has the same meaning as that in Fig. 1.1(a). A pattern graph G_P is given in Fig. 1.2(b), where an IT project requires three types of people: namely, PM, SE and S. Compared with *subgraph isomorphism*, in BGS, an edge in pattern graph can be associated with an integer to show the constraint of the maximum path length between two nodes. For

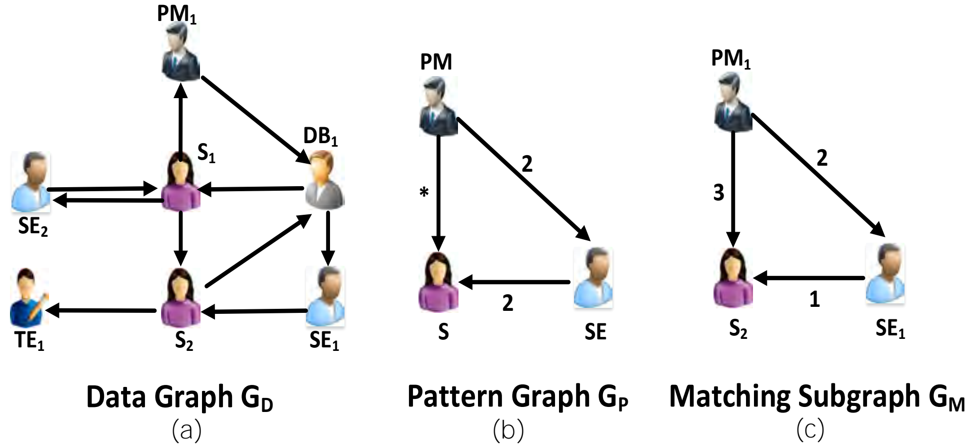


Figure 1.2: BGS Problem

example, in Fig. 1.2(b), a PM must connect with an SE within three hops, and an SE needs to connect with an S within two hops. Based on BGS, the matching subgraph G_M is shown in Fig. 1.2(c).

The subgraph isomorphism-based and BGS-based subgraph matching methods discussed above aim to find the entire subgraphs in G_D. However, in some applications, such as group finding [70] and expert recommendation [89, 18], people are more interested in finding some nodes based on a specified structure between them, leading to the *Graph Pattern based Node Matching* (GPNM) problem [81]. An example of this is discussed below.

Example 1.3 (GPNM Problem): Recall the data graph and pattern graph shown in Fig. 1.2(a) and Fig. 1.2(b) respectively. Instead of finding the matching subgraphs, GPNM aims to identify the matching nodes for each node in pattern graph. The GPNM results of Example 1.2 are shown in Table 1.1.

Table 1.1: The GPNM results of Example 1.2

Nodes in G_P	Matching nodes in G_D
PM	PM_1
SE	SE_1
S	S_2

The existing subgraph matching methods can be applied to solve the GPNM problem. However, they must deliver the entire matching subgraphs, rather than matching nodes only, which incurs a high time complexity [39, 38]. Therefore, Fan et al., [40] proposed a method to find matching nodes only based on a given pattern graph. Although their method can reduce query processing time, it does not consider the updates of G_P and G_D that commonly exist in real scenarios [10]. In addition, not all the updates in both pattern graph and data graph essentially affect the GPNM matching results, there may exists *elimination relationships* among the updates. For example, if one edge (node) is firstly removed from (or inserted into) G_D (G_P) and then inserted back into (or removed from) G_D (G_P), then the effects of the two updates can eliminate each other. By analyzing the elimination relationships among the updates, the efficiency of delivering the GPNM results can be improved. Further, in the GPNM procedure, we must inspect whether the shortest path length between two nodes can satisfy the path length constraints on the pattern graph. Since the computation of the shortest path length between any two nodes is highly time-consuming, especially in big data graphs (e.g., social networks and traffic networks), if we have a strategy to partition the graph into subgraph to overcome this bottleneck, the efficiency of GPNM can be improved further.

This thesis will focus on the three significant challenges of *frequently updated graphs*, *elimination relationships* and *graph partition*.

1.1 Challenges in GPNM

1.1.1 Frequently updated graphs

The first challenge of this thesis is: *when facing frequently updated pattern graph and data graph, how to efficiently deliver the node matching results rather than performing a whole GPNM procedure from scratch that consumes much more query processing time.*

In real scenarios, nodes and edges in both G_P and G_D are usually frequently updated over time [10]. For example, on Facebook, within each minute, on average of 400 new users join in, 510,000 comments are posted, 317,000 statuses are updated, and 147,000 photos are uploaded[1]. In the application of group finding in social graphs [70], the change of requirements (e.g., constraints or structure) leads to updates of G_P and the joining of new users in online social networks leads to the updates of G_D.

However, when facing any update, the existing GPNM methods [82, 40] must perform a new GPNM procedure from scratch, leading to much more query processing time. Although certain existing BGS-based GPSM methods [39, 41] can incrementally deliver GPSM results taking the updates of G_D into account, they still need to deliver the entire matching subgraphs when taking the updates of G_P as input, rather than delivering the matching nodes directly.

In addition, in real applications, although G_P and G_D are updated frequently, these updates usually account for a small proportion of the entire graph. For example, Facebook had more than two billion active monthly users in June 2017, and about six hundred thousand new users joined every month[2], which accounted for only 0.03% of all the users.

1.1.2 Elimination relationships

The second challenge of this thesis is: *how to effectively detect the elimination relationships of the updates.*

Although both the pattern graphs and data graphs are updated frequently, not all the updates in a pattern graph G_P or a data graph G_D essentially affect the GPNM matching results. The following example illustrates this.

Example 1.4 (The elimination relationships among the updates): Based on the data graph and pattern graph shown in Fig. 1.3(a) and Fig. 1.3(c) respectively, the original

[1] https://sproutsocial.com/insights/facebook-stats-for-marketers/
[2] https://newsroom.fb.com/company-info/

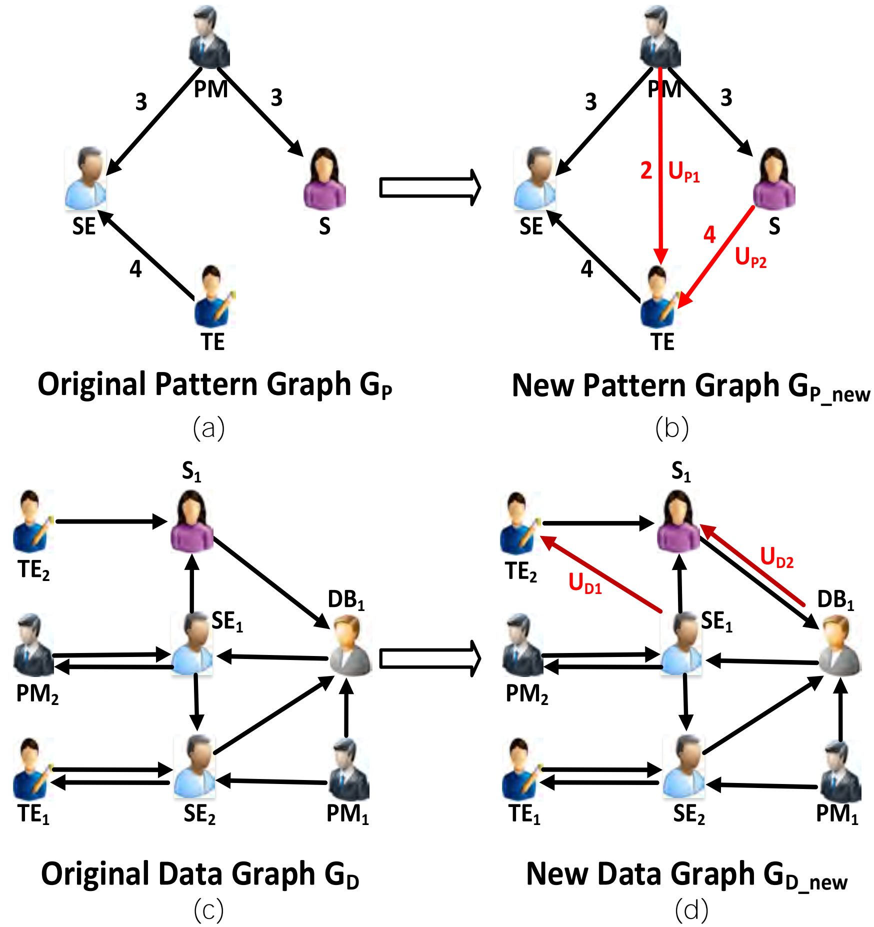


Figure 1.3: The elimination relationships among the updates

GPNM matching results are shown in Table 1.2. Suppose that there are two updates in the pattern graph, where PM needs to be associated with a TE within two hops (denoted as U_{P1} in Fig. 1.3(b)), and an S needs to be associated with a TE within four hops (denoted as U_{P2} in Fig. 1.3(b)). In addition, there are also two updates in the data graph, where SE_1 establishes the collaboration relationship with TE_2 (denoted as U_{D1} in Fig. 1.3(d)) and DB_1 establishes the collaboration relationship with S_1 (denoted as U_{D2} in Fig. 1.3(d)). The new pattern graph G_{P_new} and new data graph G_{D_new} are shown in Fig. 1.3(b) and Fig. 1.3(d), respectively.

Based on these two updated graphs, the state-of-the-art GPNM methods [82, 40] need to perform a new GPNM procedure from scratch, leading to much more query processing time. However, in practice, one update can be eliminated by another up-

Table 1.2: The original GPNM results of Example 1.4

Nodes in G_P	Matching nodes in G_D
PM	PM_1
SE	SE_1, SE_2
S	S_1
TE	TE_1, TE_2

date. It is easy to understand that in each single graph (G_P or G_D), if one edge (node) is firstly removed from (or inserted into) G_D (G_P) and then inserted back to (or removed from) G_D (G_P), the effects of the two updates can eliminate each other. Therefore, there may be elimination relationships among the updates in a single graph of G_P or G_D, and we term this kind of elimination relationships of a single graph as *single-graph elimination relationships*. More importantly, one update in a graph may eliminate an update in another graph, we term this kind of elimination relationships as *cross-graph elimination relationships*. In Example 1.4, although in update U_{P1}, a PM needs to be associated with a TE within two hops, it indeed leads to no change in the GPNM results. This is because in another update U_{D1}, SE_1 happens to establish the collaboration with TE_2, making all the PMs in the data graph be connected with a TE within 2 hops. Therefore, the effects of U_{P1} and U_{D1} eliminate each other.

It is non-trivial to identify the elimination relationships among the updates because there exist both single-graph elimination relationships and cross-graph elimination relationships. In addition, if update U_a eliminates update U_b, and update U_b eliminates update U_c, there exists a hierarchical structure of them, which applies to all the elimination relationships. As it is computationally expensive to deliver GPNM results by investigating each of the elimination relationships among the updates, it is beneficial to build up an index to record the hierarchical structure of all the elimination relationships. Therefore, *how to build up an index structure to record the hierarchical structure of all the elimination relationships, including both single-graph elimination relationships and cross-graph elimination relationships, that supports the development of an efficient algorithm to deliver the GPNM results by making use of the index.* is another

challenging problem of elimination relationships.

1.1.3 Graph partition

The third challenge of this thesis is *how to efficiently compute the shortest path length between any two nodes to accelerate the GPNM procedure without destroying the connectivity of the graphs.*

In the GPNM procedure, we need to consider whether the shortest path length between two nodes can satisfy the path length constraints on the pattern graph. The computation of the shortest path length between any two nodes is very time-consuming especially in big data graphs (e.g., social networks and traffic networks). In addition, as we mentioned above, the graphs are updated frequently and it is also time-consuming to update the shortest path length between any two nodes. In order to overcome this bottleneck, we aim to propose a strategy to partition the graph into subgraphs to accelerate the GPNM procedure.

In the partition strategy, we must ensure that the connectivity of the data graph is not destroyed and that the shortest path length between any two nodes can be efficiently updated when the graphs are modified.

1.2 Contributions of the Thesis

Targeting the above significant and challenging problems in the GPNM, this thesis has the following three major contributions:

1. The first contribution of the thesis is to propose an incremental GPNM method, called INC-GPNM, which aims to deliver the GPNM results by considering the updates of both pattern graph and data graph.

   (a) In INC-GPNM, we first propose a new index to incrementally record the shortest path length range between label types in G_D, and then propose a

novel method to incrementally investigate the affected parts of G_D based on the updates of G_D and G_P.

(b) Based on the index structure and our novel search strategies, INC-GPNM can efficiently deliver node matching results taking the updates of G_P and G_D as input, and can greatly reduce the query processing time with improved time complexity.

(c) The experiments on seven real-world social graphs demonstrate that our INC-GPNM can significantly outperform the most promising state-of-the-art static GPNM method [40], and reduces the query processing time by an average of 40.24%.

2. The second contribution of the thesis is to propose an efficient GPNM method, called EH-GPNM, to answer repeating GPNM queries with multiple updates only in data graphs. To the best of our knowledge, EH-GPNM is the first GPNM solution that takes the elimination relationships between multiple updates in a data graph G_D and the hierarchical structure of the elimination relationships into consideration.

   (a) We first propose an effective method to identify the single-graph elimination relationships existing among updates in a data graph G_D by comparing the affected nodes for each pair of updates.

   (b) We then generate an Elimination Hierarchy Tree (EH-Tree) to record the hierarchical structure of the single-graph elimination relationships. By using an EH-Tree, our method can efficiently investigate each of the elimination relationships between the updates.

   (c) The experiments conducted on five real-world social graphs demonstrate that our EH-GPNM method significantly outperforms the state-of-the-art GPNM methods [40, 106], by reducing the query processing time by averages of 51.23% and 22.59% respectively.

3. The third contribution of the thesis is to propose a more efficient GPNM method, UA-GPNM, to answer GPNM queries with multiple updates in both pattern graph and data graph. To the best of our knowledge, UA-GPNM is the first GPNM solution that considers both the single-graph elimination relationships and cross-graph elimination relationships. In addition, UA-GPNM also propose a graph partition method to accelerate the GPNM procedure.

   (a) We propose effective methods to detect the single-graph elimination relationships in a data graph and in a pattern graph, and cross-graph elimination relationships between a data graph and a pattern graph.

   (b) We build up an Elimination Hierarchy Tree (EH-Tree) to index the hierarchical structure of all the different types of elimination relationships, which enhances query processing efficiency.

   (c) We propose a graph partition method and, based on the method, the efficiency of GPNM can be further improved.

   (d) The experiments conducted on five real-world social graphs demonstrate that our UA-GPNM with graph partition strategy significantly outperforms the state-of-the-art GPNM methods [106, 105] by reducing the the query processing time by an average of 58.60% and 35.29% respectively.

1.3 Roadmap of the Thesis

The thesis is structured as follows:

Chapter 2 starts with a comprehensive literature review on GPSM and GPNM.

Chapter 3 presents an incremental GPNM method, called INC-GPNM, which aims to deliver the GPNM results considering the updates of both pattern graph and data graph. This chapter includes our paper published at IEEE ICDE 2018 [106].

Chapter 4 presents an efficient GPNM method, called EH-GPNM, to answer GP-NM queries with multiple updates in data graphs. It considers the single-graph elimi-

nation relationships between multiple updates in a data graph G_D and the hierarchical structure of these elimination relationships. This chapter includes our paper published by IEEE TKDE 2019 [105], which is available online.

Chapter 5 presents a more efficient GPNM method to answer GPNM queries with multiple updates in both pattern graph and data graph, called UA-GPNM, which considers both the single-graph elimination relationships and cross-graph elimination relationships. In addition, UA-GPNM also proposes a graph partition method to accelerate the GPNM procedure. This chapter includes our paper submitted to IEEE ICDE 2019.

Finally, Chapter 6 concludes the work in this thesis and presents directions for future research opportunities.

Chapter 2

Literature Review

The existing related methods can be classified into two categories based on the matching results they deliver: i.e., (1) *Graph Pattern based Subgraph Matching (GPSM)*, and (2) *Graph Pattern based Node Matching (GPNM)*. In this section, we review these two categories respectively. In GPSM, we focus on the existing static and incremental methods based on the exact and inexact matching results, respectively. To the best of our knowledge, there is no existing incremental GPNM method. Therefore, Section 2.2 only reviews the existing static GPNM methods based on the exact and inexact matching results.

In particular, this chapter is organized as follows:

- Section 2.1 introduces the existing static and incremental GPSM methods based on the exact and inexact matching results, respectively.
- Section 2.2 introduces the existing static GPNM methods based on the exact and inexact matching results.
- Section 2.3 presents a summary of the existing studies.

2.1 Graph Pattern based Subgraph Matching (GPSM)

GPSM aims to identify all the matching subgraphs of a pattern graph G_P in a data graph G_D. It has been increasingly applied to knowledge discovery, traffic network analysis, intelligence analysis, and social network analysis, among other areas [17, 23,

27, 110, 35]. Conventional subgraph matching solutions are based on the *subgraph isomorphism* (SI) problem [114, 32], in which the matches are based strictly on graph structure.

However, the *subgraph isomorphism* problem is an NP-Complete problem [46], which makes it computationally expensive to find the exact matching subgraphs, e-specially in large graphs. In light of the intractability of the problem, approximate solutions have been studied to find inexact matches (see [45, 100] for surveys), that have fewer restrictions but more capacity to extract more useful subgraphs with better efficiency because it supports simulation relations instead of an exact match of edges and nodes.

In real scenarios, nodes and edges in both G_P and G_D are usually frequently up-dated over time [10]. In order to efficiently deliver the GPSM results when graphs are updated, the incremental GPSM methods have been proposed.

In this section, we will review the existing static and incremental GPSM methods based on the exact and inexact matching results, respectively.

2.1.1 Static GPSM

Exact static GPSM: One of the earliest and most-cited approaches of subgraph isomorphism-based static GPSM algorithms was proposed by Ullmann [115]. This algorithm op-erates on single untyped graphs with directed or undirected edges. If the user wishes to find matches to the pattern graph G_P in the data graph G_D (shown in Fig. 2.1), Ullmann's basic approach is to enumerate all possible mappings of vertices in G_P to those in G_D using a depth-first tree-search algorithm. Each node at level i of the search tree maps vertex V_{Pi} in G_P to some vertex in G_D (shown in Fig. 2.2). Each path from root to leaf in the search-tree represents a complete mapping of the vertices in G_P to those in G_D. Any such mapping that preserves adjacency in P and G (i.e., vertices that are neighbors in G_P map to vertices that are neighbors in G_D) represents an isomor-phism from G_P to a subgraph of G_D. If no such mapping preserves adjacency, then

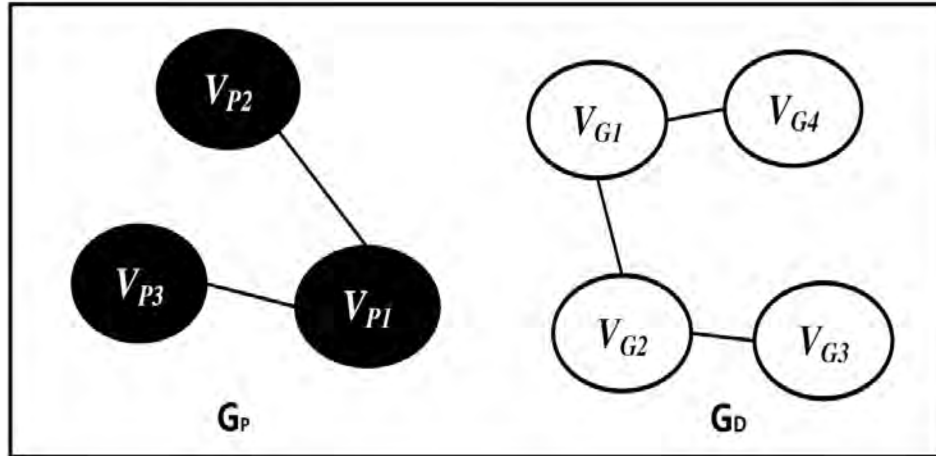


Figure 2.1: An example pattern graph G_P and data graph G_D of Ullmann

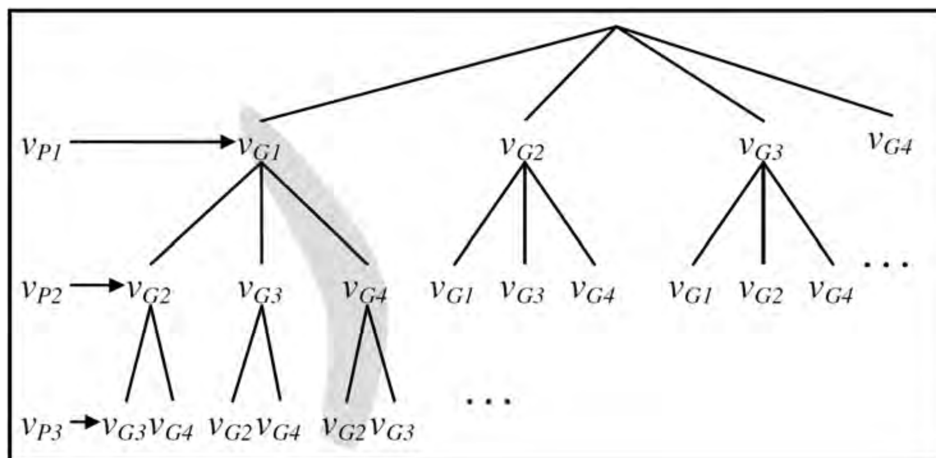


Figure 2.2: A partial search-tree for Ullmann's algorithm.

no isomorphism exists. Since the search-space considered by this approach increases exponentially with the size of the input graphs, Ullmann suggested a refinement procedure to prune unpromising sub-trees, eliminating the need to search them. This procedure eliminates vertex mappings from consideration based on three criteria:

- **Vertex degree:** If the degree of vertex V_{Pi} (i.e., the number of edges adjacent to V_{Pi}) is greater than the degree of V_{Gj} then V_{Pi} cannot map to V_{Gj}. For example, in Fig. 2.1, V_{P1} cannot map to V_{G4} since degree(V_{P1})=2 and degree(V_{G4})=1;

- **One-to-one mapping of vertices:** Once decided to map V_{Pi} to V_{Gj}, along a particular path through the tree, V_{Pi} cannot be mapped with any other vertex in G_D and we cannot map any other vertex in G_P to V_{Gj};

- **Forward checking:** Working the way down the tree, for any possible vertex mapping that remains, the mapping can be eliminated if it cannot preserve adjacency between G_P and G_D. For example, suppose that V_{P1} can be mapped to V_{G1} and we are considering the possible mapping from V_{P2} to V_{G3}. Regardless of what we do further down the tree, mapping V_{P2} to V_{G3} cannot possibly preserve adjacency since V_{P1} and V_{P2} are neighbors in G_P, but V_{G1} and V_{G3} are not neighbors in G_D. So, we can eliminate the mapping from V_{P2} to V_{G1} from further consideration.

As Ullmann's algorithm expands a particular path in the search-tree, one of two things happens:

- The algorithm eliminates all possible mappings for some vertex in G_P. In this case, the path cannot yield a match. The matching process can be safely stopped, without expanding additional nodes along the current path, and backtrack;

- The algorithm reaches a leaf in the tree, having mapped each vertex in G_P to a vertex in G_D. In this case, the path represents a match for G_P in G_D (shown in Fig. 2.2).

As noted by Messmer and Bunke, despite the refinement procedure, Ullmann's algorithm has exponential worst-case time-complexity [87]. Messmer and Bunke proposed an alternative method for exact subgraph isomorphism that has only quadratic worst-case time complexity. Their algorithm also operates on multiple untyped graphs with directed or undirected edges. The approach is to pre-process the graph data set to generate all possible permutations of the graph adjacency matrices offline and use them to build a decision tree. At run time the decision tree is used to classify the adjacency matrix of the pattern graph. The drawback to this approach is that the size of the decision tree grows exponentially with respect to the size of the data graph. To address this issue, the authors present pruning techniques, which are effective in reducing decision tree size. However, the pruned decision trees can no longer guarantee polynomial run times.

Another backtracking algorithm is the one presented in [97] by Schmidt and Druffel. It uses the information contained in the distance matrix representation of a graph to establish an initial partition of the graph nodes. This distance matrix information is then used in a backtracking procedure to reduce the search tree of possible mappings. A more recent algorithm, known as VF, is based on a depth-first search strategy, with a set of rules to efficiently prune the search tree. Such rules in case of isomorphism are shown in [31].

With regards to the graph isomorphism problem, it is also necessary to mention the McKay's nauty algorithm [85], which detects isomorphism between untyped graphs that may be directed or undirected. Nauty uses transformations to reduce graphs to a canonical form that may be checked relatively quickly for isomorphism [119]. Specifically, the algorithm computes invariants for each vertex in a graph (e.g., degree and counts of adjacent vertices of various degrees) that are used for candidate selection. Nauty partitions a graph into non-overlapping sets of vertices based on invariant values. Sets having the same invariant values can then be compared between graphs. If all sets are isomorphic between two graphs, then the two graphs must be isomorphic. Alternatively, if two graphs contain sets with differing invariants, there is no need to

test isomorphism between the sets directly.

Another possible approach to the isomorphism problem is the one presented in [19]. Instead of reducing the complexity of matching two graphs, the authors attempt to reduce the overall computational cost when matching a sample graph against a large set of prototypes. The method performs the matching in quadratic time with the size of the input graph and independently on the number of prototypes. It is obviously convenient in applications requiring the matching of a graph against a database, but the memory required to store the pre-processed database grows exponentially with the size of the graphs, making the method suitable only for small graphs. So one of the authors concludes in [86] that in cases of one-to-one matching other algorithms are more suitable.

GraphGrep [48] is another popular SI method as well. In GraphGrep, it operates in the graph-transaction setting on undirected graphs with typed vertices. The algorithm makes implicit use of vertex type information to perform matching. Matching in GraphGrep relies on the concept of a label path, which is a sequence of type labels along a path in a graph (e.g., actor-movie director-movie-actor). During index construction, the algorithm computes a "fingerprint" for each graph in the data set. The fingerprint of a graph is a set of pairs h(labelPath), count , one for each unique label path in the graph. Here h is a hash function and count is the number of instances of the specified label path in the graph. During candidate selection, the data set is filtered based on the fingerprint of the pattern graph G_P. Specifically, if a graph G_D has a lower count value than G_P for any labelPath, then G_D cannot contain an exact match for G_P and G_D is eliminated from consideration. During the subgraph matching phase, G_P is divided into a set of overlapping label paths, which are compared against the candidate graphs. The label paths of the candidate graphs that match G_P's label paths may be combined into matching subgraphs.

Yuan et al., [129] realized that, in real applications, the graph data are often noisy, incomplete and inaccurate. In other words, there are many uncertain graphs. Therefore, in [129], Yuan et al., studied pattern matching in a large uncertain graph. Specif-

ically, they aimed to retrieve all qualified matches of a query pattern in the uncertain graph. Although pattern matching over an uncertain graph is NP-Complete, they employed a filtering-and-verification framework to speed up the search. In the filtering phase, they proposed a probabilistic matching tree, a PM-tree, based on match cuts obtained by a cut selection process. Based on the PM-tree, they devised a collective pruning strategy to prune a large number of unqualified matches. During the verification phase, they developed an efficient sampling algorithm to validate the remaining candidates. Furthermore, Yuan et al., [130] proposed a tree index structure that contains the best upper bounds and collective pruning techniques to reduce the search space for retrieving matches from large uncertain graphs.

In recent years, due to the high time complexity of subgraph isomorphism, some parallel algorithms have been proposed to improve the matching efficiency. In [20], Carletti et al., proposed VF3P, a parallel algorithm to solve subgraph isomorphism. The effectiveness of the proposed algorithm was proved using very large and dense graphs considering three performance measures: the speed, efficiency and memory usage. On the base of the achieved results they demonstrated that the proposed algorithm is highly efficient and able to scale with respect to the number of used CPUs. Nevertheless, a deeper analysis can be performed to explore other aspects impacting the performance and further improvements to the efficiency can be achieved by adopting different communication schemas and agglomerations. In [118], Wang et al., proposed the StarMR star-decomposition-based query processor for efficiently answering subgraph matching queries in large RDF graph data using MapReduce. Moreover, they also developed two optimization strategies, including RDF property filtering and postponing Cartesian product operations, to improve the basic StarMR algorithm. In their method, query graphs are decomposed into a set of stars that utilize the semantic and structural information embedded RDF graphs as heuristics. Two optimization techniques are proposed to further improve the efficiency of their algorithms. One technique, called RDF property filtering, filters out invalid input data to reduce intermediate results; the other is to improve the query performance by postponing the Carte-

sian product operations. In [107], Sun et al., develop an efficient parallel subgraph enumeration algorithm for a single machine, named LIGHT. Their algorithm reduces redundant computation in DFS by delaying the materialization of pattern vertices until necessary and converting the candidate set computation into finding a minimum set cover. Moreover, they parallelize their algorithm with both SIMD (Single-Instruction-Multiple-Data) instructions and SMT (Simultaneous Multi-Threading) technologies in modern CPUs. Lai et al., [69] pointed out that the existing sequential algorithms for subgraph enumeration fall short in handling large graphs due to the involvement of computationally intensive subgraph isomorphism operations. Thus, some recent researches focus on solving the problem using MapReduce. Nevertheless, exiting MapReduce approaches are not scalable to handle very large graphs since they either produce a huge number of partial results or consume a large amount of memory. Motivated by this, Lai et al., further proposed a new algorithm TwinTwigJoin based on a left-deep-join framework in MapReduce, in which the basic join unit is a TwinTwig (an edge or two incident edges of a node). They showed that in the random graph model, TwinTwigJoin is instance optimal in the left-deep-join framework under reasonable assumptions, and they devised an algorithm to compute the optimal join plan. They further discussed how their approach can be adapted to handle the power-law random graph model. Three optimization strategies were explored to improve their algorithm. Ultimately, by aggregating equivalent nodes into a compressed node, the authors constructed the compressed graph, by which the subgraph enumeration was further improved.

Inexact static GPSM: Early approaches to inexact static GPSM were proposed by Tsai and Fu [112] and Shapiro and Haralick [99]. Both approaches are based on the idea of measuring similarity between graphs as the probability that one graph could result from a random alteration of the other. Both approaches match based on graph structure and on the attributes of individual graph elements. They are implemented using search-based algorithms with various pruning strategies. Tsai and Fu require an

exact structural match, but allow for attribute value differences. They proposed calculating empirical probabilities of attribute deformations based on observations from data. In addition, they proposed the weighted distance and weighted-square-error distance measures for cases where such data is unavailable. Shapiro and Haralick support inexact structural matching by considering graphs to match only if the amount of non-matching structure falls within some threshold. More important structural elements are given more weight and the presence or absence of these elements more heavily influences the determination of a match. Likewise, graphs only match if differences between attribute values of corresponding graph elements fall within some threshold.

SUBDUE was proposed by Cook and Holder [37], which operates in a single-graph setting with typed vertices and typed, directed edges. SUBDUE is a graph mining system, but performs pattern matching as a supporting step in the mining process. The general approach is similar to Ullmann's. They constructed a search-tree, where the nodes at the ith level map the ith vertex from G_P to some vertex in G_D. A path through the tree represents a complete mapping of vertices. Since SUBDUE performs inexact matching, each node in the search-tree has an associated cost that captures how well G_P matches G_D. If G_P and G_D are exactly isomorphic, there will be a mapping between them with cost zero. The less similar G_P and G_D are, the higher the cost will be. These costs are based on graph edit distance [90]. The edit distance between two graphs is the minimum cost of edit operations required to transform one graph into another. Edit operations include deletion, insertion, and substitution of vertices and edges. For inexact matching, the goal state is the final state (i.e., leaf) with the lowest cost of all final states. Since the search-space is again exponentially large, SUBDUE applies a branch-and-bound search to the tree. Because branch-and-bound is guaranteed to find an optimal solution and thus, the search terminates once any complete mapping is found. The algorithm also allows an upper limit to be placed on the number of search nodes considered, which can lead to a significant savings in search time at the expense of solution quality.

LAW [122] performs inexact pattern matching on typed, directed graphs. Patterns are represented as graphs with typed vertices and edges. The pattern language also supports the construction of more sophisticated pattern queries through constraints between vertices, hierarchy (i.e., sub-patterns), disjunction, and cardinality (i.e., the number occurrences of a vertex or edge). Like SUDBUE, LAW uses graph edit distance to measure similarity between potential matches. LAW's graph edit operations include deletion and replacement of vertices and edges. LAW uses ontological distance to measure differences between types. The LAW search algorithm is based on A* and selects tree nodes for expansion based on the minimum worst-case cost. This cost is calculated as the true cost of the mappings so far plus the cost of deleting all unexplored vertices and edges in the pattern. Although the worst-case cost heuristic is not admissible (in fact, it is an upper bound on the actual cost), LAW does find the lowest-cost matches because, unlike pure A*, LAW uses the heuristic only as a selection rule and not as its termination condition.

TMODS [29, 52] uses genetic algorithms to find exact and inexact pattern matches in directed, attributed graphs. Patterns may specify both structural and attribute characteristics. TMODS searches for patterns from the bottom-up, finding sub-patterns first and then composing them into more complex higher-level patterns. Coffman et al., did not describe the TMODS pattern matching algorithm in further detail.

TRAKS [2] performs inexact pattern matching in typed, directed graphs. Matches are ranked by similarity to the original pattern, taking into account ontological distance between types. Entities in a pattern are processed in ascending order according to the frequency of their type to rapidly eliminate non-matches. The algorithm searches for matches in a depth-first fashion by expanding partial matches by one vertex or edge at a time.

In 2010, Fan et al., proposed *Bounded Graph Simulation* (BGS) [39], which has fewer restrictions but more capacity to extract more useful subgraphs with better efficiency because it supports simulation relations instead of an exact match of edges and nodes. In BGS, each edge in G_P is labeled with either a positive integer k or a symbol

"*". k is the constraint of the maximal shortest path length of a match in G_D and "*" indicates that there are no path length constraints. Then, the match of an edge could be a path if the start node and the end node of the path in the data graph have the same labels as the corresponding nodes of the edge in the pattern graph respectively. In social networks, according to the theory of "six degrees of separation" [88], on average, any two people can be connected in about six hops. Therefore, k is usually set as a small integer in social networks [39].

BGS has the disadvantage of capturing the topology of data graphs, i.e., graphs may have a structure that is drastically different from the pattern graphs they match, and the matches found are often too large to understand and analyze. As an extension work of [39], in [84], to rectify these problems, Ma et al., further proposed a notion of *Strong Simulation*, a revision of graph simulation, for graph pattern matching. In [84], Ma et al., identified a set of criteria for preserving the topology of graphs matched. Strong simulation preserves the topology of data graphs and identifies a bounded number of matches. Their work showed that strong simulation retains the same complexity as earlier extensions of graph simulation, by providing a cubic-time algorithm for computing strong simulation. They also presented the locality property of strong simulation, which allowed us to develop an effective distributed algorithm to conduct graph pattern matching on distributed graphs.

In recent years, simulation-based methods have been proposed to support many emerging applications. For instance, in [80], Liu et al., first conceptually extended Bounded Simulation to Multi-Constrained Simulation (MCS), and proposed a novel NP-Complete Multi-Constrained Graph Pattern Matching (MC-GPM) problem. Then, to address the efficiency issue in large-scale MC-GPM, they proposed a new concept called Strong Social Component (SSC), consisting of participants with strong social connections. They also proposed an approach to identify SSCs, and propose a novel index method and a graph compression method for SSC. Moreover, they devised a heuristic algorithm to identify MC-GPM results effectively and efficiently without decompressing graphs.

In [131], Yuan et al., discussed the problem of subgraph similarity matching, where given a pattern graph and a data graph, subgraph similarity matching is to retrieve all matches of the pattern graph in data graph with the number of missing edges bounded by a given threshold. A data graph can be extremely large, (e.g., a web-scale graph containing hundreds of millions of vertices and billions of edges). to address this problem, the authors investigated subgraph similarity matching for a web-scale graph deployed in a distributed environment. They proposed distributed algorithms and optimization techniques that exploit the properties of subgraph similarity matching, so that they can well utilize the parallel computing power and lower the communication cost among the distributed data centers for query processing. Specifically, they first relaxed and decomposed the pattern graph into a minimum number of sub-queries. Next, they sent each sub-query to conduct the exact matching in parallel. Finally, they scheduled and joined the exact matches to obtain final query answers. Moreover, their workload-balance strategy further accelerated the query processing.

Lyu et al., [83] highlighted that, most of the existing solutions for subgraph searches follow the pruning-and-verification framework, which prunes false answers based on features in the pruning phase and performs subgraph isomorphism testings on the remaining graphs in the verification phase. However, they are not scalable to handle large-sized data-graphs and query-graphs due to three drawbacks. First, they rely on a frequent subgraph mining algorithm to select features which is expensive and cannot generate large features. Second, they require a costly verification phase. Third, they process features in a fixed order without considering their relationship to the query-graph. In this paper, Lyu et al., addressed the three drawbacks and proposed new indexing and query processing algorithms. In indexing, they selected features directly from the data-graphs without expensive frequent subgraph mining. The features form a feature-tree that contains all-sized features and both the cost sharing and pruning power of the features are considered. In query processing, they proposed a verification-free algorithm, where the order to process features is query-dependent by considering both the cost sharing and the pruning power. They explored two optimization strategies to

further improve the algorithm efficiency. The first strategy applies a lightweight graph compression technique and the second strategy optimizes the inclusion of answers.

In [127], Yang et al., studied the problem of diversified subgraph querying in a large graph, which is to find k subgraphs that are isomorphic to a given query graph with the maximum coverage. They proposed a novel level-based algorithm called DSQL with an approximation guarantee. DSQL proceeds from low to high levels and the level number refers to the number of common vertices of a newly selected subgraph with the collected subgraphs.

2.1.2 Incremental GPSM

Social graphs are frequently updated, and it is computationally expensive to perform a new procedure from scratch to find matching subgraphs when facing any updates. Therefore, the incremental GPSM methods have been proposed.

Exact incremental GPSM: An early approach to exact incremental GPSM was proposed in [96]. In this paper, Rudolf et al., proposed a way to represent and solve the graph matching problem as a CSP. The main benefit of this approach is that they gain direct access to the rich research findings in the CSP area; instead of inventing new algorithms for graph matching from scratch, they can now apply well-elaborated CSP solution algorithms right “out of the box”. Another important advantage is that the actual solution algorithm becomes independent of the concrete graph model, allowing to change the model without affecting the algorithm. Users need only to adapt the translation step from the graph model into the CSP representation.

In [14], Borgwardt et al., extended frequent subgraph mining algorithms to time series of graphs. In particular, they were looking for subgraphs that are topologically frequent within a large graph and that show insertions and deletions of edges in the same temporal order. They first searched for edges whose edge type is frequent within the dynamic graph (neglecting edge existence strings). Among these, they first ensure

that embeddings of candidates do not overlap by applying a greedy maximal independent set algorithm. Then they count non-overlapping occurrences of each candidate. Finally, they verify whether these embeddings constitute a frequent dynamic subgraph (FDS), and if so, they mark these embeddings and rewrite the graph such that each marked embedding of this FDS is represented by one super-node.

Rather than repeating the search of the template graph at each instance of change, in [104], Stotz et al., proposed an incremental subgraph isomorphism approach. The algorithm draws strongly from their previous state space search approach, TruST. They have shown that the incremental search procedure generates solutions of the same (or similar) quality in significantly lower computational times.

For the important application of graph pattern matching, community finding, in [53], Greene et al., have described both a general model for tracking communities in dynamic networks, and a fast, effective method based on the model that readily scales to graphs. They have described an approach for bench-marking dynamic community finding using synthetic graphs with embedded community events. Evaluations on these synthetic networks show that the proposed method performs at least as well if not better than static community finding. Additionally, Greene et al., performed a preliminary evaluation on a real-world mobile call network. Their method uncovered a large number of dynamic communities with different evolutionary characteristics in this network, while requiring relatively little computational overhead. Their experiments on the network suggest that the choice of the time step window size is important.

Recently, in [47], Gillani et al., studied the problem of GPSM over graph streams using event-based and incremental evaluation models. Gillani et al., proposed a query-based graph pruning technique to enable join-ahead pruning of unnecessary triples. They used a bidirectional multi-map data structure to materialise a set of pruned tables. These tables are then processed using fast hash-join, thus further removing the unnecessary triples. The final pruned set of triples is explored using a graph structure. They named this technique as join-and-explore and it is index-free. Gillani et al., used an automata-based model to guide the join and exploration process and extended these

techniques to enable the incremental evaluation of graph streams, i.e., by joining the new updates only with the matched triples within a window.

In order to give accurate and timely in manner recommendations for cold-start users, in [132], Zhang et al., proposed an incremental graph pattern matching based dynamic cold-start recommendation method (IGPMDCR), which updates similar users for cold-start users according to the latest social relationship, and provides recommendations based on the latest similar users' records.

Given the history of a node-labeled graph in the form of graph snapshots, corresponding to the state of the graph at different time instants, Semertzidis et al., in [98] focused on the problem of efficiently finding the most durable patterns, that is, patterns that persist over time, either continuously or collectively. They have proposed an approach termed *Durable Pattern* that is able to identify durable patterns by traversing a compact representation of the graph snapshots. They also introduced time and neighborhood indexes on labels and nodes that boost the candidate patterns reduction.

In [78], Li et al., studied subgraph isomorphism issues with the timing order constraint over high-speed streaming graphs. They proposed an expansion list to efficiently answer subgraph searches and proposed MS-tree to greatly reduce the space cost. More importantly, they designed effectively concurrency management in their computation to improve system's throughput. For this, they first studied concurrency management on subgraph matching over streaming graphs, then, evaluated their solution on both real and synthetic benchmark datasets. Their extensive experimental results confirm the superiority of their approach compared with the state-of-the-arts subgraph match algorithms on streaming graphs.

Inexact incremental GPSM: An inexact algorithms have been studied for incremental subgraph searching in [117]. In this paper, Wang et al., investigated a new problem in continuous subgraph pattern searching in the situation where multiple target graphs are constantly changing in a stream style, namely the subgraph pattern search over graph streams. The proposed problem is clearly a continuous join between query patterns

and graph streams where the join predicate is the existence of subgraph isomorphism. Due to the NP-Complete of subgraph isomorphism checking, to achieve the real-time monitoring of the existence of certain subgraph patterns, they avoided using subgraph isomorphism verification to find the exact query stream subgraph isomorphic pairs but aimed to offer an approximate answer that could report all probable pairs without excluding any of the actual answer pairs. They proposed a light-weight yet effective feature structure called Node-Neighbor Tree to filter false candidate query-stream pairs. To reduce the computational cost, they further projected the feature structures into a numerical vector space and conducted dominant relationship checking in the projected space. They proposed two methods to efficiently check dominant relationships and substantiate their methods with extensive experiments.

Based on BGS, Fan et al., [41] proposed an incremental approximate method to identify the matching subgraphs. They proposed a revision of graph pattern matching based on a notion of BGS. This yielded a cubic-time method for finding matches, as opposed to the intractability of its counterpart via subgraph isomorphism. Moreover, the method is able to capture more sensible matches in emerging applications. Fan et al., also investigated the incremental pattern matching problem for matching defined in terms of subgraph isomorphism, graph simulation, and BGS, from complexity (boundedness) analysis to incremental algorithms. They showed that the incremental matching problem is unbounded for matching based on all the three notions. Nonetheless, for graph simulation and BGS, they showed that their incremental matching problems are semi-bounded, and developed efficient incremental algorithms for (possibly cyclic) patterns and batch updates. They have also developed incremental algorithms for maintaining auxiliary data structures, that is, landmark and distance vectors. These allow users to efficiently identify matches when data graphs are updated, minimizing unnecessary re-computation. Their experimental results have verified the scalability and effectiveness of their batch and incremental methods, using real-life and synthetic data. They further experimented with real-life datasets in various domains, to identify areas in which the revised matching is most effective. They also investigated opti-

mization techniques, as well as lower bounds for incremental matching by exploring the usage patterns of real-life networks [120, 92, 68]. Finally, Fan et al., extended their incremental matching methods to querying distributed graphs, using MapReduce. The complexity of this method is more accurately characterized in terms of the size of the area affected by the updates of data graphs, rather than the size of the entire input.

Targeting the labeled graph, in [126], Yang et al., considered the individual needs of users and proposed a dynamic Top-K interesting subgraph query. This method establishes a novel graph topology feature index (GTSF index) including a node topology feature index (NTF index) and an edge feature index (EF index), which can effectively prune and filter the invalid nodes and edges that do not meet the restricted condition. The multi-factor candidate set filtering strategy was proposed based on the GTSF index, which can be further pruned to obtain fewer candidate sets. Yang et al., then proposed a dynamic Top-K interesting subgraph query method based on the idea of the sliding window. This allowed them to realize the dynamic modification of the matching results of the subgraph in the dynamic evolution of the label graph, to ensure real-time and accurate query results. In addition, considering the factors, including frequent Input/Output (I/O) and network communication overheads, the optimization mechanism of the graph changes and an incremental maintenance strategy for the index are proposed to reduce the significant cost of redundant operation and global updates. The experimental results showed that the proposed method can effectively manage a dynamic Top-K interesting subgraph query on a large-scale labeled graph. Simultaneously, the optimization mechanism of graph changes and the incremental maintenance strategy of the index can effectively reduce the maintenance overheads.

Kim et al., [66] proposed a fast and continuous subgraph matching system called TurboFlux, which provides a high throughput over a fast graph update stream. Their work shows that TurboFlux solved the problems of existing methods and efficiently processed continuous subgraph matching for each update operation. They first proposed the novel notion of a data-centric graph, which is an efficiently updatable graph for storing partial solutions. Then, they proposed the edge transition model, which

efficiently identifies which update operation can affect the current partial solutions and/or contributes to generating positive/negative matches. They next presented the detailed algorithms of TurboFlux under the edge transition model and explained how the re-computation of subgraph matching for each update operation can be minimized.

In the real application of community finding, in [76], Li et al., studied the problem of finding persistent communities in a temporal network, in which every edge is associated with a timestamp. They aimed to identify the communities that are persistent over time and thus, proposed a novel persistent community model to capture the persistence of a community. To solve this problem, they first proposed a near-linear temporal graph reduction algorithm to prune the original temporal graph substantially, without loss of accuracy. Then, in the reduced temporal graph, they presented a novel branch-and-bound algorithm with several carefully designed pruning rules to efficiently find the maximum persistent communities efficiently. In addition, in [74], Li et al., further proposed a method to seek cohesive subgraphs in a signed network, in which each edge can be positive or negative, denoting friendship or conflict, respectively.

2.2 Graph Pattern based Node Matching (GPNM)

Applying the existing GPSM methods to solve the GPNM problem incurs a high time complexity as they need to deliver the entire matching subgraphs in G_D [39, 38]. Therefore, several GPNM methods have been proposed and aim to find some nodes based on a specified structure between those nodes, such as in the applications of group finding [70] and expert recommendation [89]. To the best of our knowledge, there is no existing incremental GPNM method. Therefore, this section only reviews the existing static GPNM methods based on the exact and inexact matching results respectively.

2.2.1 Static GPNM

Exact static GPNM: In [111], Tong et al., focused on large graphs where nodes have attributes, such as a social network where the nodes are labeled with each person's job title. In such a setting, they aimed to find subgraphs that match a user query pattern. For example, an query would be, "find a CEO who has strong interactions with a Manager, a Lawyer, and an Accountant, or another structure as close to that as possible". Similarly, a loop query could help detect a money laundering ring. Traditional SQL-based methods, as well as more recent graph indexing methods, will return no answer when an exact match does not exist. Their method can find exact matching nodes, and it will present them to the user in their proposed order.

In [51], Gou et al., proposed two efficient algorithms, DP-B and DP-P, for retrieving top-ranked matching nodes from large graphs. Their first algorithm, DP-B, is able to retrieve exact top-ranked answer matching nodes from a potentially exponentially number of matches in time and space linear in the size of the data inputs even in the worst case. Further, beyond the linear-cost result of DP-B, their second algorithm, DP-P, could require far less than linear time and space cost in practice. Their algorithms are the first to have these performance properties.

In [134], Zou et al., proposed a novel pattern match problem over a large graph. They transformed vertices in graphs into points in a vector space via *subgraph isomorphism* methods, converting a pattern match query into a distance-based multi-way join problem over vector space. Several pruning techniques are developed to reduce the search space significantly, such as neighbor area pruning, triangle inequality pruning and hash join. They also designed a cost estimation technique to identify a cheap query plan (i.e., join order).

Team formation is one of the typical applications of GPNM [135, 8, 25, 121]. Lappas et al. [70] introduced the problem of discovering a team of experts from a social network, that satisfies all attributed skills required for a given task with low communication cost. Kargar and An [64] studied the team formation problem with a

team leader who communicates with each team member to monitor and coordinate the project. Most of the team formation studies focus on a tree substructure, as opposed to the densely connected subgraph required by community searches. Gajewar and Sarma [44] extended the team formation problem to allow for potentially more than one member possessing each required skill, and use maximum density measure or minimum diameter as the objective.

Social circle discovery is one special kind of GPNM applications in social networks [61], where, for a query user, social circles are communities formed only by their friends. The induced subgraph of an entire network producted only by their friends and themselves is called "ego network". The authors in [73] proposed an unsupervised community model to automatically detect circles in ego networks. The discovered circles are disjointed, overlapping and hierarchically nested. Social circles can affect the process of information diffusion in social contagion [113]. A social circle represents the distinct social context of a user, and the multiplicity of social contexts is termed "structural diversity" [113]. Taking one social contagion process in Facebook as an example, a user is much more likely to join Facebook and become engaged if he or she has a larger structural diversity. The authors in [56, 57] studied the problem of finding k users with the highest structural diversity in graphs, which can be beneficial to political campaigns, promotion of health practices, marketing, and so on.

Keyword search over a graph aims to find a substructure of the graph containing all or some of the input keywords. Most of previous methods in this area find connected minimal trees that cover all the query keywords. In [11], a backward search algorithm for producing Steiner trees was presented. A dynamic programming approach for finding Steiner trees in graphs was presented in [36]. Although the dynamic programming approach has exponential time complexity, it is feasible for input queries with small number of keywords. In [50], the authors proposed algorithms that produce Steiner trees with polynomial delay. The algorithms follows the Lawlers procedure [71]. Due to the NP-Complete of the Steiner tree problem, producing trees with distinct roots was

introduced in [63]. BLINKS improved the work of [63] by using an efficient indexing structure [55]. Recently, it was shown that finding GPNM results rather than trees can be more useful and informative for the users. However, the current tree or graph based methods may produce answers in which some content nodes (i.e., nodes that contain input keywords) are not very close to each other. In addition, when searching for answers, these methods may explore the whole graph rather than only the content nodes. This may lead to poor performance in execution time. To address the above problems, in [65], Kargar et al., proposed the problem of finding r-cliques in graphs. An r-clique is a group of content nodes that cover all the input keywords and the distance between each two nodes is less than or equal to r. An exact algorithm was proposed that finds all r-cliques in the input graph. In addition, an approximation algorithm that produces r-cliques with 2-approximation in polynomial delay was proposed.

Community search is an important problem in GPNM. Given a graph and a set of query nodes, the community search problem aims to identify a cohesive subgraph containing the query nodes. Community search has recently attracted significant attention, fueled by applications such as advertising and viral marketing, content recommendation, and team formation [34]. A number of criteria to assess the goodness of a community have been proposed, such as random-walk-based measures [109, 67], or density-based measures [33, 58, 75, 123, 72, 49, 22]. Sozio and Gionis [103] introduced the term "community-search problem", which they formalized from a combinatorial-optimization perspective as the problem of finding a connected subgraph that contains all query vertices and maximizes the minimum degree. Minimum degree has in fact been shown to be an effective way of measuring the goodness of a community [13, 34].

In [95], Ruan et al., discussed a very simple approach of combining content and link information in graph structures for the purpose of community discovery, a fundamental task in network analysis. Their approach hinges on the basic intuition that many networks contain noise in the link structure and that content information can help strengthen the community signal. This enables elimination of the effect of noise (false positives and false negatives), which is particularly prevalent in online social

networks and web-scale information networks. Specifically, they introduce a measure of signal strength between two nodes in the network by fusing their link strength with content similarity. Link strength is estimated based on whether the link is likely (with high probability) to reside within a community. Content similarity is estimated through cosine similarity or the Jaccard coefficient. They discussed a simple mechanism for fusing content and link similarity, then presented a biased edge sampling procedure that retains edges that are locally relevant for each graph node. The resulting backbone graph can be clustered using standard community discovery algorithms such as Metis and Markov clustering.

In recent years, local search for communities in real graphs is attracting ever-increasing research interests [28, 5, 116, 33]. Aaron [28] first proposed the problem of finding a community with size constraint k for a certain vertex. He used "local modularity" as the community goodness measure, which characterized the relative density within the community to outside of the community. According to the new measure, he proposed a heuristic algorithm with quadratic time complexity regarding to k. Bagrow [5] further improved the performance by selecting the vertex with largest "outwardness", where outwardness of vertex v is the number of v' neighbors outside the community minus the number inside. Local search of community is widely used in existing Sybil defense schemes [116], where the local community around a trusted node is also considered trustworthy [116]. Cui [33] studied the overlapping structure of local search. These local search methods [28, 5] find communities with size constraint to limit the search space. As a result, they are not guaranteed to find the best community under a corresponding community goodness measure. Another weakness is that the size constraint as an input parameter is difficult to select. In general, each vertex has its own the most appropriate community size. Predefining a global parameter or testing different parameters blindly can hardly find meaningful results. To deal with the limitation of the existing work, in [34], Cui et al., investigated the problem of finding the best community containing a given query vertex in its neighborhood. They proposed a local search method for this purpose. Local search is more efficient

than global search since global search needs to visit all vertices in the network for community detection. They addressed the local search challenge that arises from the non-monotonicity of community goodness measure and proposed the CST and the CSM algorithms to solve a variety of community search problems.

In [7], Barbieri et al., focused on the min-degree-based formulation of community search and propose a novel method that overcomes the limitations of existing approaches. They pointed out that the community-search problem based on min-degree can be solved in linear time in the size of the input graph. Specifically, the algorithm by Sozio and Gionis [103] is a global search (GS) method that needs to visit the entire input graph, thus being computationally expensive on large graphs. A more efficient solution has been recently proposed by Cui et al [34]. Their local search (LS) method however works only for single-vertex queries. Then, in their work [7], Barbieri et al., proposed an approach that improves upon the efficiency of those methods, including the method by Cui et al [34]. on the special case of a single query vertex. Barbieri et al., did so while still keeping generality, as their method is able to handle multiple query vertices. In addition, in the approach they proposed in this work they do not explicitly constrain the size of the communities, rather they aim at finding the smallest-sized solution among all the optimal ones. Thus, they are able to produce communities that are on average orders of magnitude more effective, (i.e., smaller and denser) than global and local search methods. Moreover, unlike a constrained global search, their approach achieves high efficiency, identifies optimal communities and requires no parameters.

Community search in attributed graphs aims to identify all densely connected components with homogeneous attributes [133, 26, 95]. Zhou et al. [133] modelled the community detection problem as graph clustering, and combined structural and attribute similarities through a unified distance measure. When high-dimensional attributed communities are difficult to interpret or discover, [59, 54] considered subspace clustering on high-dimensional attributed graphs. A survey of clustering on attributed graphs can be found in [15]. Community detection in attributed graphs is to find all

communities of the entire graph, which is clearly different from the goal of query-based community search. Moreover, it is practically hard and inefficient to adapt the above community detection approaches for online attributed community search: community detection is inherently global and much of the work involved may be irrelevant to the community being searched.

In [75], Li et al., noted that many previous studies on community search do not consider the influence of a community; thus, in [75], Li et al., introduced a novel community model called k-influential community based on the concept of k-core, which can capture the influence of a community. Based on the new community model, they proposed a linear-time online search algorithm to find the top-k k-influential communities in a network. To further speed up the influential community searching algorithm, they devised a linear-space index structure that supports efficient search of the top-k k-influential communities in optimal time. They also proposed an efficient algorithm to maintain the index when the network is frequently updated.

Fang et al., [43] investigated the attributed community query (or ACQ), which returns an attributed community (AC) for an attributed graph. Due to the recent developments of gigantic social networks (e.g., Flickr, Facebook, and Twitter), the topic of attributed graphs has attracted attention from industry and research communities [124, 11, 36, 55, 63, 128, 65]. An attributed graph is essentially a graph associated with text strings or keywords. The AC is a subgraph of data graph, which satisfies both structure cohesiveness (i.e., its vertices are tightly connected) and keyword cohesiveness (i.e., its vertices share common keywords). The AC enables a better understanding of how and why a community is formed (e.g., members of an AC have a common interest in music, because they all have the same keyword “music”). An AC can be “personalized”; for example, an ACQ user may specify that an AC returned should be related to some specific keywords like “research” and “sports”. To enable efficient AC search, they develop the CL-tree structure and three algorithms based on it. In contrast, the model [60] is based on k-truss in structure with a relaxed attribute function. They formulated their problem of finding attributed truss commu-

nities (ATC), as finding all connected and close k-truss subgraphs containing node v, that are locally maximal and have the largest attribute relevance score among such subgraphs. They designed a novel attribute relevance score function and establish its desirable properties. The problem is shown to be NP-hard. However, they developed an efficient greedy algorithmic framework, which finds a maximal k-truss containing node v, and then iteratively removes the nodes with the least popular attributes and shrinks the graph so as to satisfy community constraints. They also built an elegant index to maintain the known k-truss structure and attribute information, and propose efficient query processing algorithms.

Recent studies proposed the computation of top-k influential communities, where each reported community not only is a cohesive subgraph but also has a high influence value. The existing approaches to the problem of top-k influential community search can be categorized as index-based algorithms [75] and online search algorithms without indexes [24]. The index-based algorithms, although being very efficient in conducting community searches, need to pre-compute a special purpose index and only work for one built-in vertex weight vector. In [12], Bi et al., investigated online search approaches and propose an instance-optimal algorithm LocalSearch whose time complexity is linearly proportional to the size of the smallest subgraph that a correct algorithm needs to access without indexes. In addition, they also propose techniques to make LocalSearch progressively compute and report the communities in decreasing influence value order such that k does not need to be specified. Moreover, they extended their framework to the general case of top-k influential community search regarding other cohesiveness measures.

With the rapid development of location-aware mobile devices, many users share their locations in social networks [4, 77]. The problem of querying geo-social groups seeks a group of users densely and closely connected in terms of both social and spatial proximity. Yang et al., [125] proposed a new family of k-core based geo-social group queries with minimum acquaintance constraint. Li et al., [77] studied a minimum user group query, in which each user had k neighbors and the users' joint regions covered

all query points. Variants of R-tree index structure integrating the social information are designed for different geo-social query processing. A general framework that offers flexible data management and algorithmic design for geo-social network queries was proposed in [4, 3].

Inexact static GPNM: To improve efficiency, Tong et al., [111] proposed a "Seed-Finder" method that identifies approximate matches for certain pattern nodes. This method only requires cubic time.

In [64], Kargar et al., studied the problem of discovering a team of experts from a social network. Given a project whose completion requires a set of skills, their goal is to find a set of experts that together have all of the required skills and also have the minimal communication cost among them. Kargar et al., proposed two communication cost functions designed for two types of communication structures and show that the problem of finding the team of experts that minimizes one of the proposed cost functions is NP-hard. Thus, an approximation algorithm with an approximation ratio of two was designed. In addition, they introduced the problem of finding a team of experts with a leader. The leader is responsible for monitoring and coordinating the project, and thus a different communication cost function is used in this problem. To solve this problem, an inexact polynomial algorithm was proposed. They showed that the total number of teams may be exponential with respect to the number of required skills. Thus, two procedures that produce top-k teams of experts with or without a leader in polynomial delay were also proposed.

Recently, there has been significant interest in the study of the community search problem in social and information networks: given one or more query nodes, users may aim to find densely connected communities containing the query nodes. However, most existing studies do not address the "free rider" issue, that is, nodes far away from query nodes and irrelevant to them are included in the detected community. Some models have attempted to address this issue, but not only are their formulated problems NP-hard, but they also do not admit any approximations without restrictive

assumptions, which may not always hold in practice. In [62], given an undirected graph G_D and a set of query nodes Q, Huang et al., studied community search using the k-truss based community model. They formulated their problem of finding a closest truss community (CTC), as finding a connected k-truss subgraph with the largest k that contains Q, and has the minimum diameter among such subgraphs. They proved this problem is NP-hard. Furthermore, it is NP-hard to approximate the problem within a factor. However, they developed a greedy algorithmic framework, which first finds a CTC containing Q, and then iteratively removes the furthest nodes from Q, from the graph. The method achieves 2-approximation to the optimal solution. To further improve the efficiency, they used of a compact truss index and developed efficient algorithms for k-truss identification and maintenance as nodes are eliminated. In addition, using bulk deletion optimization and local exploration strategies, they proposed two more efficient algorithms. One of them trades some approximation quality for efficiency while the other is a highly efficient heuristic.

Based on BGS, Fan et al., [40] revised graph patterns to support a specific output node. Given data graph and patten graph, they aimed to find those nodes in the matching graphs that match a specific node v, instead of the subgraphs. They studied two classes of functions for ranking the matches: relevance functions and distance functions. They develop two algorithms for computing top-k matches of node v based on relevance functions, with the early termination property (i.e., they identified top-k matches without computing the entire graphs). They also studied diversified top-k matching, a bi-criteria optimization problem based on both relevance functions and distance functions. Finally, they provided an approximation algorithm with performance guarantees and a heuristic algorithm with the early termination property. Motivated by network analysis applications, Fan et al., further [42] proposed quantified matching for a specific pattern node, in which they extend traditional graph patterns with counting quantifiers.

In [91], Namaki et al., studied the problem of event pattern discovery by keywords in graph streams. Specifically, they proposed a class of event patterns to capture events

relevant to user-specified keywords, by integrating (approximate) topological and value bindings from keywords. They also introduced an activeness measure, to balance the pattern expressiveness and the cost of pattern discovery. In addition, they also developed both from-scratch algorithm to discover and maintain active events in graph streams.

In [101], Shi et al., noted that the existing GPNM methods for matching the designated node v do not consider the multiple constraints of the attributes associated with each vertex and each edge which commonly exist in real applications (e.g., the constraints of social contexts for the experts recommendation in contextual social). In their work, they first proposed the Multi-Constrained Top-K Graph Pattern Matching problem (MC-Top-K-GPM), which involves the NP-Complete Multiple Constrained GPNM problem. To address the efficiency and effectiveness issues of MC-TopK-GPM in large-scale social graphs, they proposed a novel index, called HB-Tree, which indexes the label and degree of nodes in graphs and can get candidates of v efficiently. Furthermore, they developed a Multi-Constrained Top-K GPNM method, called MTK, which can identify Top-K matches of v effectively and efficiently.

In [79], Liu et al., analysed a new type of context-aware graph pattern based node selection problem. Then they proposed two index structures, MA-Tree, and SSC index to save the important information, such as the categories of nodes, the shortest path between nodes, the aggregated social contexts, and the predecessors and ancestors of each node in SSCs, which can greatly improve the efficiency of search. Finally, they have proposed a probabilistic algorithm based on the Monte Carlo algorithm, called MC-TAG-K, with the index structures and the proposed early termination strategies.

2.3 Conclusion

The existing methods in the above two categories face the efficiency issue when answering GPNM queries with the updates in both pattern graphs and data graphs. The existing GPNM methods do not consider any update of a pattern graph or a data graph.

Therefore, with the updates of a pattern graph and/or a data graph, they have to perform a new GPNM procedure from scratch to deliver the node matching results, which consumes much more query processing time. In contrast, the existing incremental GPSM methods consider the updates of a data graph but return the matching subgraphs, rather than the matching nodes. In addition, when the pattern graph is updated, these existing incremental GPSM methods also have to perform a new GPSM procedure from scratch to find the matching subgraphs consuming much more query processing time.

Chapter 3

Incremental Graph Pattern based Node Matching with Multiple Updates

In real scenarios, nodes and edges in both G_P and G_D are usually frequently updated over time. For example, in the application of group finding in social graphs [70], the change of requirements (e.g., constraints or the structure) leads to the updates of G_P, and the joining of new users in OSNs leads to the updates of G_D. However, when facing with any update, the existing GPNM methods [81, 40] need to perform a new GPNM procedure from scratch, leading to much more query processing time. Although some existing BGS based GPSM methods [39, 41] can incrementally deliver GPSM results taking the updates of G_D into account, they still need to deliver the entire matching subgraphs when taking the updates of G_P as input, rather than delivering the matching nodes directly.

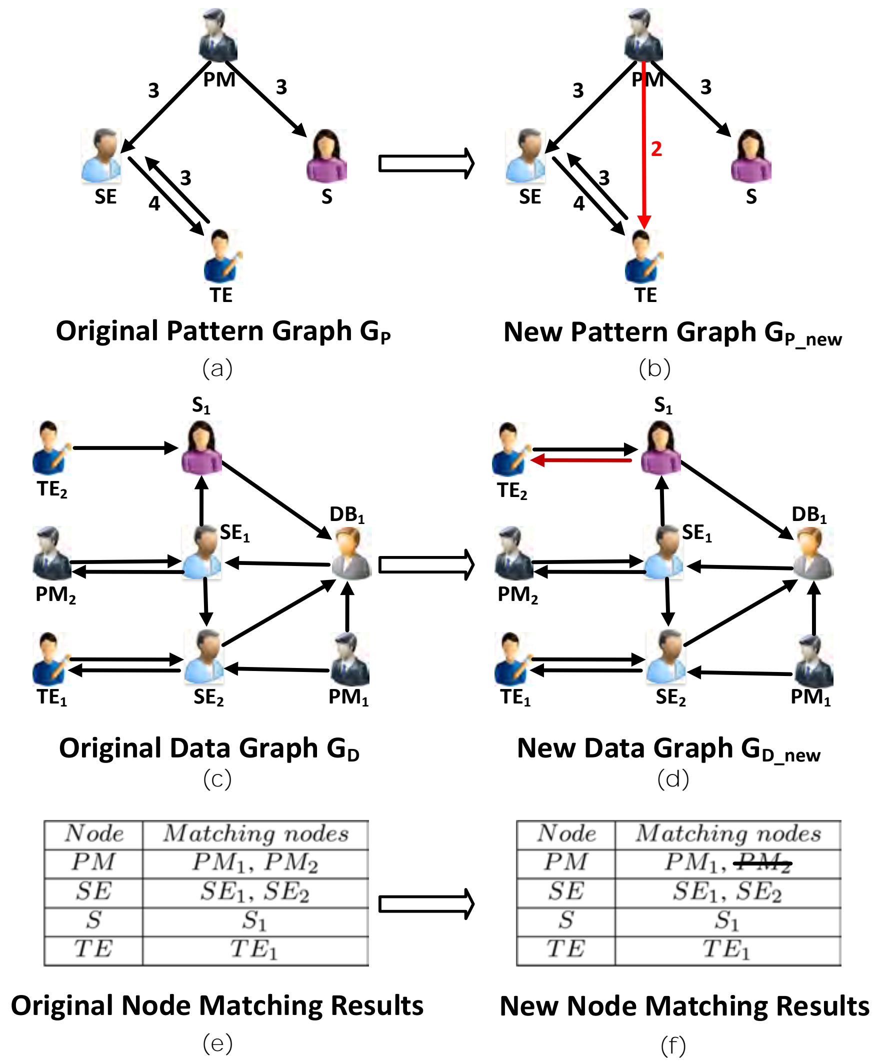
In addition, in real applications, although G_P and G_D are updated frequently, such updates usually account for a small proportion of the entire graphs. For example, Facebook had more than two billion monthly active users in June 2017, and about six hundred thousand new users joined every month[1], which accounted for only 0.03% of all the users. This characteristic motivates us to develop an incremental GPNM method to investigate the affected parts of G_D and efficiently deliver the node matching result-s, rather than performing a whole GPNM procedure from scratch that consumes much more query processing time. The following example illustrates the incremental GPN-

[1] https://newsroom.fb.com/company-info/

M procedure.

Example 3.1 (Incremental GPNM): Given a pattern graph G_P and a data graph G_D shown in Fig. 3.1(a) and Fig. 3.1(b)) respectively. A new directed edge $e(PM, TE)$ from a PM to an TE is added into G_P, and the bounded path length (length constraint) on this edge is 2 (depicted as the red arrow line in Fig. 3.1(b)). In addition, a new edge $e(S_1, TE_2)$ is added into G_D in Fig. 3.1(c) (depicted as the red arrow line in Fig. 3.1(d)). The new pattern graph G_{P_new} and new data graph G_{D_new} are shown in Fig. 3.1(b) and Fig. 3.1(d) respectively. In this example, with the updates of G_P and G_D, PM_2 in G_{D_new} is not the matching node of PM in G_{P_new} any more because PM_2 cannot be connected with TE within 2 hops. The new node matching results are shown in Fig. 3.1(f). Compared to the original node matching results, we find that the new node matching results exclude PM_2 as the matching node of PM based on G_{P_new} and G_{D_new}.

Based on the above explanation, in order to deliver the node matching results, after $e(PM, TE)$ and $e(S_1, TE_2)$ are added into G_P and G_D respectively, we only need to investigate whether PM_2 in G_D can still be the matching node of PM and whether TE_2 in G_D can be the new matching node of TE based on G_{P_new} and G_{D_new}. More specifically, we need to investigate whether the shortest path lengths between PM_2 and SE_1, and between PM_2 and SE_2 can satisfy the requirements of the corresponding bounded path length on $e(PM, SE)$. In addition, we also need to investigate whether the shortest path lengths between SE_1 and TE_2, and between SE_2 and TE_2 can satisfy the requirements of the corresponding bounded path length on $e(SE, TE)$. By considering the affected parts of G_D, we only need to investigate the shortest path length of *4 pairs of nodes*. By contrast, the existing GPNM methods [81], [40] need to perform a new GPNM procedure from scratch, and thus have to investigate the shortest path length between all the *18 pairs of nodes in total*. Therefore, a new efficient GPNM method is in demand.



Node	Matching nodes
PM	PM_1, PM_2
SE	SE_1, SE_2
S	S_1
TE	TE_1

Node	Matching nodes
PM	PM_1, ~~PM_2~~
SE	SE_1, SE_2
S	S_1
TE	TE_1

Figure 3.1: Incremental GPNM

3.1 Problem Definition

Before introducing the detailed steps of the algorithm, we first introduce the related definitions.

3.1.1 Data Graph, Pattern Graph and BGS

Data Graph. A data graph is a directed graph $G_D = (V_D, E_D, f_a)$, where

- V_D is a finite set of nodes;
- $E_D \subseteq V_D \times V_D$, in which $(u, u')\in E$ denotes a directed edge from node u to u';
- $f_a(u)$ is a function such that for each node $u \in V_D$, $f_a(u)$ is a set of labels. Intuitively, f_a consists of the attributes of a node, e.g., name, age, job title [70].

Example 3.2: G_D in Fig. 3.1(c) depicts a data graph. In G_D, each node denotes a person, together with the labels of a person, e.g., PM means this person is a project manager. Each edge denotes a relationship between the two connected nodes, e.g., $e(TE_2, S_1)$ means TE_2 has a collaboration relationship with S_1.

Pattern Graph. A pattern graph is defined as $G_P = (V_P, E_P, f_v, f_e)$, where

- V_P and E_P are a set of nodes and a set of directed edges, respectively;
- f_v is a function defined on V_P such that for each node $u \in V_P$, $f_v(u)$ is the label of a node u, e.g., Project Manager;
- f_e is a function defined on E_P such that for each edge (u, u'), $f_e(u, u')$ is the bounded path length of (u, u') that is either a positive integer k or a symbol "*".

Example 3.3: G_P in Fig.3.1 (a) depicts a pattern graph. In addition to the labels, each edge in G_P has an integer as the bounded path length.

Bounded Graph Simulation. Consider a data graph $G_D = (V_D, E_D, f_a)$ and a pattern $G_P = (V_P, E_P, f_v, f_e)$. The data graph G_D *matches* the pattern graph G_P based on

bounded graph simulation, denoted by $G_P \trianglelefteq G_D$, if there exists a binary relation $M \subseteq V_P \times V_D$ such that

- for any $u \in V_P$, there exists $v \in V_D$, such that $(u, v) \in M$;
- $f_a(v)$ of v includes $f_v(u)$ of u;
- for each edge (u, u') in E_P, there exists a $path\ \rho = v/.../v'$ in G_D such that $(u', v') \in M$, and $len(\rho) \leq k$ if $f_e(u, u') = k$.

Note that there exists a path ρ from u to u' with $len(\rho) \leq k$ if the shortest path length from u to u' is no longer than k. If $G_P \trianglelefteq G_D$, the graph pattern matching results are denoted as $M(G_P, G_D)$ [39].

Example 3.4: Recall G_P and G_D given in Fig.3.1 (a) and Fig.3.1 (c) respectively, and $G_P \trianglelefteq G_D$. A match M_1 is obtained in G_D w.r.t. G_P by mapping PM to PM_1, SE to SE_1, S to S_1, and TE to TE_1, respectively. Another match M_2 is obtained by mapping PM to PM_2, SE to SE_2, S to S_1, and TE to TE_1, respectively. Here TE_2 cannot be mapped to TE because the shortest path length from SE_1 to TE_2 and the shortest path length from SE_2 to TE_2 are both longer than the bounded path length.

3.1.2 Incremental GPNM Problem

GPNM. Given a pattern graph G_P, a data graph G_D, for each node u_i in G_P, we define the matching node of u_i in G_D to be $N_{u_i} = \{v_i | (v_i, u_i) \in M(G_P, G_D)\}$. *GPNM* is to find N_{u_i} for each u_i of G_P in G_D. If G_D has no match of G_P based on BGS, then $N_{u_i} = \emptyset$.

Example 3.5: Recall G_P and G_D given in Fig.3.1 (a) and Fig.3.1 (c) respectively. Instead of finding the subgraphs which can match G_P based on BGS, the GPNM only aims to find the matching nodes in G_D for each node of G_P. Taking PM as an example, since PM_1 and PM_2 are in the subgraphs which can match G_P based on BGS, they are the matching nodes of PM. However, TE_2 is not the matching node of TE

as it is not in any subgraph which can match G_P. The node matching results of G_P in G_D are shown in Fig.3.1 (e).

Incremental GPNM:

- **Input:** a pattern graph G_P, a data graph G_D, the original node matching results N_{u_i}, a list $\triangle G_D$ of *updates* to G_D, a list $\triangle G_P$ of *updates* to G_P.
- **Output:** the updated node matching results N'_{u_i} of G_{P_new} in G_{D_new} (G_{P_new} and G_{D_new} denote the updated G_P and updated G_D respectively).

Remark. $\triangle G_D$ includes the insertion of edges, insertion of nodes, deletion of edges and deletion of nodes, denoted by $\triangle G^+_{D_E}$, $\triangle G^+_{D_N}$, $\triangle G^-_{D_E}$ and $\triangle G^-_{D_N}$ respectively. $\triangle G_P$ includes the insertion of edges, insertion of nodes, deletion of edges and deletion of nodes, denoted by $\triangle G^+_{P_E}$, $\triangle G^+_{P_N}$, $\triangle G^-_{P_E}$ and $\triangle G^-_{P_N}$, respectively.

Example 3.6: Recall G_P and G_D given in Fig.3.1 (a) and Fig.3.1 (c). Suppose $\triangle G^+_{P_E} = \{e(PM, TE)\}$ with the bounded path length on $e(PM, TE)$ is 2, and $\triangle G^+_{D_E} = \{e(S_1, TE_2)\}$. The G_{P_new} and G_{D_new} are shown in Fig.3.1 (d) and Fig.3.1 (e) respectively. PM_2 is not the matching node of PM any more because PM_2 cannot be connected to TE within 2 hops. The new node matching results are shown in Fig.3.1 (f).

3.2 INC-GPNM Algorithm

In this section, we introduce our INCremental Graph Pattern based Node Matching method called INC-GPNM in detail.

3.2.1 INC-GPNM Overview

When a pattern graph G_P is updated, INC-GPNM builds up *the shortest path length matrix*, called $SLen$, to record the length of the shortest path between each pair of nodes in G_D, and builds up *the shortest path length range matrix*, called R_{SLen}, to

record the shortest path length range between each category of nodes in G_D. Based on $SLen$ and R_{SLen}, INC-GPNM can investigate whether the updates of G_P satisfy the following two conditions: (1) $N^{'}_{u_i}$ = $\emptyset$, and (2) $N^{'}_{u_i}$ = N_{u_i}. If both of them are not satisfied, two procedures $PMatch^+$ and $PMatch^-$ are devised to identify the affected parts of G_D and incrementally deliver $N^{'}_{u_i}$ respectively. $PMatch^+$ is performed when new edges and/or nodes are added into G_P, denoted by $\triangle G^+_{P_E}$ and $\triangle G^+_{P_N}$ respectively. $PMatch^-$ is performed when edges and/or nodes are removed from G_P, denoted by $\triangle G^-_{P_E}$ and $\triangle G^-_{P_N}$ respectively. The details of the index establishment, $\triangle G_P$ checking process, and the details of $PMatch^+$ and $PMatch^-$ are discussed in *Section 3.2.2*.

When a data graph G_D is updated, INC-GPNM builds up *the set of affected pairs of nodes*, called AFF, in which the length of the shortest path between each pair of nodes is changed due to $\triangle G_D$. Based on AFF, INC-GPNM can investigate whether the updates of G_D have an influence on N_{u_i}. If there is no influence, then $N^{'}_{u_i}$ = N_{u_i}. Otherwise, two methods $DMatch^+$ and $DMatch^-$ are devised to identify the affected parts of G_D and deliver $N^{'}_{u_i}$ respectively. $DMatch^+$ is performed when edges and/or nodes are added into G_D, denoted by $\triangle G^+_{D_E}$ and $\triangle G^+_{D_N}$ respectively. $DMatch^-$ is performed when edges and/or nodes are removed from G_D, denoted by $\triangle G^-_{D_E}$ and $\triangle G^-_{D_N}$ respectively. The details of AFF, $\triangle G_D$ checking process, and the details of $DMatch^+$ and $DMatch^-$ are discussed in *Section 3.2.3*.

3.2.2 Index Generation and INC-GPNM for $\triangle G_P$

$SLen$ and R_{SLen}: In GPNM, in addition to the matching of labels and the satisfaction of the requirements of reachability in G_P, we need to investigate if the shortest path length between each pair of nodes in G_D can satisfy the requirements of the bounded path length in G_P. Therefore, we build the shortest path length matrix, $SLen$, to record the shortest path length between each pair of nodes in G_D. In addition, based on $SLen$, we build the shortest path length range matrix, R_{SLen}, to record the shortest

path length range between different label types in G_D. When an edge $e(u, v)$ is added into or removed from G_P, based on R_{SLen}, INC-GPNM can investigate if the nodes in G_D that have the same label as u and v can satisfy the bounded path length on $e(u, v)$. As the updates of G_D lead to the change of $SLen$, we adopt the method proposed in [94] to incrementally update $SLen$, and then update R_{SLen} accordingly. The method proposed in [94] maintains the all-pairs's shortest path when edges are deleted from graphs or edges are inserted into graphs, and it is also adopted in some BGS based GPSM and GPNM methods [2, 7, 9]. In [94], for certain edge-modifications (insertion of edges or deletion of edges), it is first used to detect the nodes where the shortest path lengths between them are unchanged. *Dijkstra*'s algorithm is applied for updating the shortest path lengths between the affected nodes. After adopting this method, it is still necessary to check if any change of the shortest path lengths in the data graph affects the matching results, and if any update in the pattern graph leads to the change of GPNM results. Otherwise the processing efficiency would be bad. For example, the most promising state-of-the-art GPNM approach in [7] also adopts the approach in [26], however, in this approach, any of such updates leads to a new procedure of finding matching results from scratch consuming much more query processing time (see details in experiments).

Example 3.7: $SLen$ and R_{SLen} of G_D in Fig.3.1 (c) are shown in Table 3.1 and Table 3.2 respectively.

Table 3.1: $SLen$ of G_D in Fig. 3.1(c).

	PM_1	PM_2	SE_1	SE_2	S_1	TE_1	TE_2	DB_1
PM_1	0	3	2	1	3	2	∞	1
PM_2	∞	0	1	2	2	3	∞	3
SE_1	∞	1	0	1	1	2	∞	2
SE_2	∞	3	2	0	3	1	∞	1
S_1	∞	3	2	3	0	4	∞	1
TE_1	∞	4	3	1	4	0	∞	2
TE_2	∞	4	3	4	1	5	0	2
DB_1	∞	2	1	2	2	3	∞	0

Table 3.2: R_{SLen} of G_D in Fig. 3.1(c).

	PM	SE	S	TE	DB
PM	[0, 0]	[1, 2]	[2, 3]	[2, ∞]	[1,3]
SE	[1, ∞]	[0, 0]	[1, 3]	[1, ∞]	[1, 2]
S	[3, ∞]	[2, 3]	[0, 0]	[4, ∞]	[1, 1]
TE	[4, ∞]	[1, 4]	[1, 4]	[0, 0]	[2, 2]
DB	[2, ∞]	[1, 2]	[2, 2]	[3, ∞]	[0, 0]

INC-GPNM for $\triangle$G$_\mathbf{P}$: When a pattern graph is updated, INC-GPNM has different processes based on the following situations.

(1) **For each edge $\boldsymbol{e(u,v)} \in \triangle \boldsymbol{G^{+}_{P_E}}$** with $R_{SLen}(u,v) = [a,b]$,

- **Situation 1:** if $a > f_e(u,v)$, then set $N^{'}_{u_i}$ = $\emptyset$;
- **Situation 2:** if $b \leq f_e(u,v)$, then set $N^{'}_{u_i}$ = N_{u_i};
- **Situation 3:** if $a \leq f_e(u,v) < b$, then $N^{'}_{u_i} \subseteq N_{u_i}$, and INC-GPNM performs $PMatch^{+}$ procedure to incrementally deliver the GPNM results.

(2) **For each node $\boldsymbol{e(u,v)} \in \triangle \boldsymbol{G^{+}_{P_N}}$**, if v is isolated from G_P, then $N^{'}_{u_i}$ = $N_{u_i} \cup \{v_j\}$, where v_j is the node in G_D that has the same label as v. If the newly added node leads to one or several new edges in G_P, the process of these newly added edges is similar to the above mentioned method for the edges in $\triangle G^{+}_{P_E}$. The only difference is that, when facing *Situation 2*, INC-GPNM needs to add the node in G_D that has the same label as v into N_{u_i}, i.e., $N^{'}_{u_i}$ = $N_{u_i} \cup \{v_j\}$.

(3) **For each edge $\boldsymbol{e(u,v)} \in \triangle \boldsymbol{G^{-}_{P_E}}$** with $R_{SLen}(u,v) = [a,b]$,

- **Situation 4:** if $b \leq f_e(u,v)$, then set $N^{'}_{u_i}$ = N_{u_i};
- **Situation 5:** if $b > f_e(u,v)$, then $N_{u_i} \subseteq N^{'}_{u_i}$, and INC-GPNM performs $PMatch^{-}$ procedure to incrementally deliver the GPNM results.

(4) **For each node $\boldsymbol{e(u,v)} \in \triangle \boldsymbol{G^{-}_{P_N}}$**, the corresponding edges are removed, where v is the start node or the end node of the edges. The process of these deleted edges is similar to the above mentioned method for $\triangle G^{-}_{P_E}$. The only difference is that, when

facing *Situation 4*, INC-GPNM needs to remove the node from N_{u_i} that has the same label as v, i.e., $N'_{u_i} = N_{u_i} \setminus \{v_j\}$.

PMatch⁺: Given $\triangle G^+_{P_E}$ and $\triangle G^+_{P_N}$, the matching nodes in G_D need to satisfy the new constraint of reachability, the matching of labels and bounded path lengths on these newly added edges and/or nodes. $PMatch^+$ investigates the affected nodes in N_{u_i} and removes the nodes that do not satisfy these new constraints from N_{u_i}. The pseudo-code of $PMatch^+$ is shown in *Algorithm 1* and explained below.

For each edge $\boldsymbol{e(u,v) \in \triangle G^+_{P_E}}$,

- **Step 1:** For each pair of nodes (u_i, v_j) that have the same labels as u and v respectively in N_{u_i}, if $SLen(u_i, v_j) > f_e(u, v)$, then (a) when there is no other node v_n in N_{u_i}, such that $SLen(u_i, v_n) \leq f_e(u, v)$, we add u_i into the *candidate set of deleted nodes*, denoted as $DeleteSet()$; (b) when there is no other node u_n in N_{u_i}, such that $SLen(u_n, v_j) \leq f_e(u, v)$, we add v_j into $DeleteSet()$ (*lines 2-9 in Algorithm 1*);

- **Step 2:** For each node v_i in $DeleteSet()$, if there is a pattern edge $e(v, u)$ or $e(u, v)$ in G_P, and v_i is the only node such that $SLen(v_i, u_j) \leq f_e(v, u)$ or $SLen(u_j, v_i) \leq f_e(u, v)$, we add u_j into $DeleteSet()$ (*lines 10-14 in Algorithm 1*);

- **Step 3:** *PMatch*$^+$ recursively performs *Step 2* to identify the nodes that need to be removed, and terminates when no new node can be added into $DeleteSet()$. Then, INC-GPNM returns the new GPNM result N'_{u_i} (*lines 15-17 in Algorithm 1*).

For each node $\boldsymbol{v \in \triangle G^+_{P_N}}$, INC-GPNM first adds the nodes that have the same label as v into N_{u_i}, and then performs $PMatch^+$ procedure to deliver the node matching results. The correctness of $PMatch^+$ is proven in *Theorem 3.1*.

Algorithm 1: PMatch⁺

```
    Input: G_P, G_D, N_{u_i}, e(u,v) ∈ △G^+_{P_E}, f_e(u,v), SLen
    Output: N'_{u_i}
 1  Set DeleteSet() = ∅;
 2  for each pair of nodes (u_i, v_j) ∈ N_{u_i} do
 3      if SLen(u_i, v_j) > f_e(u,v) then
 4          for each v_n ∈ N_{u_i}(n ≠ j) do
 5              if There is no v_n such that SLen(u_i, v_n) ≤ f_e(u,v) then
 6                  Add u_i into DeleteSet();
 7          for each u_n ∈ N_{u_i}(n ≠ i) do
 8              if There is no u_n such that SLen(u_n, v_j) ≤ f_e(u,v) then
 9                  Add v_j into DeleteSet();
10  for each node v_i ∈ DeleteSet() do
11      if (v,u) ∈ G_P and v_i is the only node such that SLen(v_i, u_j) ≤ f_e(v,u)
        then
12          Add u_j into DeleteSet();
13      if (u,v) ∈ G_P and v_i is the only node such that SLen(u_j, v_i) ≤ f_e(u,v)
        then
14          Add u_j into DeleteSet();
15      if There is no newly added node in DeleteSet() then
16          Break;
17  return N'_{u_i} = N_{u_i} \ DeleteSet();
```

Theorem 3.1: *Taking $\triangle G^{+}_{P_E}$ and $\triangle G^{+}_{P_N}$ as input, $PMatch^{+}$ can deliver correct $N^{'}_{u_i}$ based on G_{P_new} and G_{D_new}.*

The Proof of Theorem 3.1: Let $N^{''}_{u_i}$ denote the correct node matching results based on G_{P_new} and G_{D_new}. Suppose $N^{'}_{u_i} \neq N^{''}_{u_i}$, then there is at least one node v such that (1) $v \in N^{''}_{u_i}$ and $v \notin N^{'}_{u_i}$; or (2) $v \notin N^{''}_{u_i}$ and $v \in N^{'}_{u_i}$. If $v \in N^{''}_{u_i}$, since $N^{''}_{u_i} \subseteq N_{u_i}$, then $v \in N_{u_i}$. Since $N^{'}_{u_i} = N_{u_i} \setminus DeleteSet()$, if $v \notin N^{'}_{u_i}$, then $v \in DeleteSet()$, which means v is not a matching node due to $\triangle G^{+}_{P_E}$ and/or $\triangle G^{+}_{P_N}$. This contradicts $v \in N^{''}_{u_i}$. If $v \notin N^{''}_{u_i}$, since $N^{''}_{u_i} \subseteq N_{u_i}$, then $v \notin N_{u_i}$ or $v \in N_{u_i} \setminus N^{''}_{u_i}$. If $v \notin N_{u_i}$, since $N^{'}_{u_i} = N_{u_i} \setminus DeleteSet()$, then $v \notin N^{'}_{u_i}$. This contradicts $v \in N^{'}_{u_i}$. If $v \in N_{u_i} \setminus N^{''}_{u_i}$, which means v is not a matching node due to $\triangle G^{+}_{P_E}$ and/or $\triangle G^{+}_{P_N}$, then $v \in DeleteSet()$,

since $N^{'}_{u_i}$ = $N_{u_i} \setminus DeleteSet()$, thus, $v \notin N^{'}_{u_i}$. This contradicts $v \in N^{'}_{u_i}$. Therefore, *Theorem 3.1* is proven. □

Example 3.8: Recall G_P and G_D in Fig. 3.1 (a) and Fig. 3.1 (c). Suppose $\triangle G^{+}_{P_E}$ = $\{e(SE, S)\}$ with $f_e(SE, S) = 2$. (1) For each matching node of SE and S in N_{u_i}, by checking $SLen$, we find that $SLen(SE_1, S_1) = 1$ and $SLen(SE_2, S_1) = 3$. Then S_1 can be kept in N_{u_i} and SE_2 should be added into $DeleteSet()$. (2) After that, as $SLen(PM_1, SE_1)$, $SLen(PM_2, SE_1)$, $SLen(TE_1, SE_1)$ and $SLen(TE_1, SE_1)$ all can satisfy the bounded path length in G_D, they are not added into $DeleteSet()$. Then the results of $N^{'}_{u_i}$ is by matching PM_1, PM_2 to PM; SE_1 to SE; S_1 to S and TE_1 to TE.

Complexity: The time complexity of the establishment and updates of the index is $\mathcal{O}(|N_D|(|N_D| + |E_D|)$ [6]. In the worst case, for each edge $e \in \triangle G^{+}_{P_E}$, $PMatch^{+}$ needs to check $SLen$ for each pair of nodes in N_{u_i}. Since the number of the nodes in N_{u_i} is bounded in $|N_D|$, the time complexity of $PMatch^{+}$ is $\mathcal{O}(|N_D|(|N_D| + |E_D|) + |\triangle G^{+}_{P_E}||N_D|^2)$.

PMatch⁻: Given $\triangle G^{-}_{P_E}$ and $\triangle G^{-}_{P_N}$, the constraint of reachability, the matching of labels and the bounded path length on these removed nodes and edges can be neglected. Therefore, the node $v \in N_{u_i}$ is still a matching node. $PMatch^{-}$ investigates the affected parts in G_D and adds the nodes that match G_{P_new} into N_{u_i}. The pseudo-code of $PMatch^{-}$ is shown in *Algorithm 2* and explained below.

For each edge $e(u, v) \in \triangle G^{-}_{P_E}$,

- **Step 1:** In contrast to **PMatch⁺**, if $SLen(u_i, v_j) > f_e(u, v)$, we add u_i and v_j into the *candidate set of newly added nodes*, denoted as $AddSet()$ (*lines 2-4 in Algorithm 2*);
- **Step 2:** If there is a pattern edge $e(v, u)$ in G_P, then for each pair of nodes

in $AddSet()$, if $SLen(v_j, u_i) \leq f_e(v, u)$, keep them in $AddSet()$; otherwise, remove them from $AddSet()$ (***lines 5-7 in Algorithm 2***);

- **Step 3:** For each node v_i in $AddSet()$, if there is a pattern edge $e(v, u)$ or $e(u, v)$ in G_P, and there is a node $u_j \notin N_{u_i}$ such that $SLen(v_i, u_j) \leq f_e(v, u)$ or $SLen(u_j, v_i) \leq f_e(u, v)$, we add u_j into $AddSet()$ (***lines 8-16 in Algorithm 2***);

- **Step 4:** *PMatch*$^-$ recursively performs *Step 3* to identify the new matching nodes that need to be added into $AddSet()$, and terminates when no new node can be added into $AddSet()$. Then, INC-GPNM returns the new GPNM result N'_{u_i} (***lines 17-19 in Algorithm 2***).

For each node $v \in \triangle G^-_{P_N}$, INC-GPNM first removes the nodes that have the same label as v from N_{u_i}, and then performs $PMatch^-$ procedure to deliver the node matching results. The correctness of $PMatch^-$ is proven in *Theorem 3.2*.

Theorem 3.2: *Taking $\triangle G^-_{P_E}$ and $\triangle G^-_{P_N}$ as input, $PMatch^-$ can deliver correct N'_{u_i} based on G_{P_new} and G_{D_new}.*

The Proof of Theorem 3.2: Let N''_{u_i} denote the correct node matching results based on G_{P_new} and G_{D_new}. Suppose $N'_{u_i} \neq N''_{u_i}$, then there is at least one node v such that (1) $v \in N''_{u_i}$ and $v \notin N'_{u_i}$; or (2) $v \notin N''_{u_i}$ and $v \in N'_{u_i}$. If $v \notin N'_{u_i}$, since $N_{u_i} \subseteq N'_{u_i}$, then $v \notin N_{u_i}$. If $v \in N''_{u_i}$ with $v \notin N_{u_i}$, since $N_{u_i} \subseteq N''_{u_i}$, then v is a new matching node due to $\triangle G^-_{P_E}$ and/or $\triangle G^-_{P_N}$. Thus, $v \in AddSet()$. Since $N'_{u_i} = N_{u_i} \cup Addset()$, then $v \in N'_{u_i}$. This contradicts $v \notin N'_{u_i}$. If $v \notin N''_{u_i}$, since $N_{u_i} \subseteq N''_{u_i}$, then $v \notin N_{u_i}$. As $N'_{u_i} = N_{u_i} \cup Addset()$, if $v \in N'_{u_i}$, then $v \in AddSet()$. Thus v is a new matching node due to $\triangle G^-_{P_E}$ and/or $\triangle G^-_{P_N}$. This contradicts $v \notin N''_{u_i}$. Therefore, *Theorem 3.2* is proven. □

Example 3.9: Recall G_P and G_D in Fig.3.1 (a) and Fig.3.1 (c). Suppose $\triangle G^-_{P_E} = \{e(SE, TE)\}$. (1) For each pair of nodes labeled with SE and TE respectively in G_D, we can get $SLen(SE_1, TE_1) = 2$, $SLen(SE_1, TE_2) = \infty$, $SLen(SE_2, TE_1) = 1$

Algorithm 2: PMatch$^-$

Input: G_P, G_D, N_{u_i}, $e(u,v) \in \triangle G^-_{P_E}$, $SLen$
Output: N'_{u_i}

```
1  Set AddSet() = ∅ ;
2  for each pair of nodes (u_i, v_j) ∈ G_D do
3      if SLen(u_i, v_j) > f_e(u, v) then
4          Add u_i, v_j into AddSet();
5      if (v, u) ∈ G_P then
6          if SLen(v_j, u_i) > f_e(v, u) then
7              remove u_i, v_j from AddSet();
8  for each node v_i ∈ AddSet() do
9      if (v, u)∈ G_P then
10         for each v_i ∉ N_{u_i} do
11             if SLen(v_i, u_j) ≤ f_e(v, u) then
12                 Add the u_j into AddSet();
13     if (u, v)∈ G_P then
14         for each u_j ∉ N_{u_i} do
15             if SLen(u_j, v_i) ≤ f_e(u, v) then
16                 Add the u_j into AddSet();
17     if There is no newly added node in AddSet() then
18         Break;
19 return N'_{u_i} = N_{u_i} ∪ Addset();
```

and $SLen(SE_2, TE_2) = \infty$. Their values are all greater than $f_e(SE, TE)$. Thus, we add SE_1, TE_2 and SE_2 into $AddSet()$. (2) Since there is a pattern edge $e(TE, SE)$ in G_P, and $SLen(TE_2, SE_1) = 3$, $SLen(TE_2, SE_2) = 4$. Then, we add TE_2 and SE_2 into $AddSet()$. (3) Since PM_1, PM_2 and TE_1 are already in N_{u_i}, no new node can be added into $AddSet()$. Then the results of N'_{u_i} is by matching PM_1, PM_2 to PM; SE_1, SE_2 to SE; S_1 to S and TE_1, TE_2 to TE.

Complexity: In the worst case, for each edge $e \in \triangle G^-_{P_E}$, $PMatch^-$ needs to check $SLen$ for each pair of nodes in G_D. Since $PMatch^-$ and $PMatch^+$ have the same indices, the time complexity of $PMatch^-$ is $\mathcal{O}(|N_D|(|N_D|+|E_D|)+|\triangle G^-_{P_E}||N_D|^2)$.

3.2.3 Index Generation and INC-GPNM for $\triangle G_D$

When a data graph is updated, INC-GPNM has different processes based on the following situations.

(1) **For each edge $e(u,v) \in \triangle G^{+}_{D_E}$,**

- **Situation 6:** for each pair of nodes $(u_i, v_j) \in AFF$, if there is no pattern edge $e(u,v) \in G_P$, then set $N^{'}_{u_i} = N_{u_i}$;
- **Situation 7:** if there is one pattern edge $e(u,v) \in G_P$, then INC-GPNM performs *DMatch*$^+$ procedure to incrementally deliver the GPNM results.

(2) **For each node $v \in \triangle G^{+}_{D_N}$**, if v has no links with other nodes in G_D, then v cannot be the matching node if there is no isolated node in G_P. Therefore, when the newly added node v does not affect N_{u_i}, then $N^{'}_{u_i} = N_{u_i}$. If v leads to one or several new edges in G_D, the process of these newly added edges is the same as the above mentioned method for $\triangle G^{+}_{D_E}$.

(3) **For each edge $e(u,v) \in \triangle G^{-}_{D_E}$,**

- **Situation 8:** for each pair of nodes (u_i, v_j) in AFF, if there is no pattern edge $e(u,v) \in G_P$, then set $N^{'}_{u_i} = N_{u_i}$;
- **Situation 9:** if there is one pattern edge $e(u,v) \in G_P$, then INC-GPNM performs *DMatch*$^+$ procedure to incrementally deliver the GPNM results.

(4) **For each node $v \in \triangle G^{-}_{D_N}$**, the corresponding edges are removed, where v is the start node or the end node of the edges. The process of these deleted edges is the same as the above mentioned method for $\triangle G^{-}_{D_E}$.

DMatch$^+$: If $v \in N_{u_i}$, which means v satisfies the constraints before G_D is updated, given $\triangle G^{-}_{D_E}$ and/or $\triangle G^{-}_{D_N}$, the shortest path lengths between v and other nodes in G_D keep unchanged or decrease, thus v still satisfies the constraints. Therefore, $DMatch^+$ investigates the affected nodes in G_{D_new} and adds the nodes that match the pattern

graph into N_{u_i}. The pseudo-code of *DMatch*$^+$ is shown in *Algorithm 3* and explained below.

For each edge $e(u,v) \in \triangle G_{D_E}^+$,

- **Step 1:** For each pair of nodes (u_i, v_j) in AFF with $AFF[u_i, v_j] = [a, b]$, if $a > f_e(u,v)$ and $b \leq f_e(u,v)$, we add u_i and v_j into the *candidate set of newly added nodes*, denoted as $AddSet()$ (*lines 2-4 in Algorithm 3*);

- **Step 2:** If there is a pattern edge $e(v,u)$ in G_P, then for each pair of nodes in $AddSet()$, if $SLen_{new}(v_j, u_i) \leq f_e(v,u)$, keep them in $AddSet()$; otherwise, remove them from $AddSet()$ (*lines 5-7 in Algorithm 3*);

- **Step 3:** For each node v_i in $AddSet()$, if there is a pattern edge $e(v,u)$ or $e(u,v)$ in G_P, and there is a node $u_j \notin N_{u_i}$ such that $SLen_{new}(v_i, u_j) \leq f_e(u,v)$ or $SLen_{new}(u_j, v_j) \leq f_e(u,v)$, we add u_j into $AddSet()$ (*lines 8-16 in Algorithm 3*);

- **Step 4:** *DMatch*$^+$ recursively performs *Step 3* to identify the new matching nodes that need to be added into $AddSet()$, and terminates when no new node can be added into $AddSet()$. Then, INC-GPNM returns the new GPNM result $N_{u_i}^{'}$ (*lines 17-19 in Algorithm 3*).

For each node $v \in \triangle G_{D_N}^+$, if v has no links with other nodes in G_D, then v cannot be the matching node if there is no isolated node in G_P. Therefore, the newly added node v does not affect N_{u_i}, then $N_{u_i}^{'}$ = N_{u_i}. If v leads to one or several new edges in G_D, the procedure of these newly added edges is the same as the above mentioned method for $\triangle G_{D_E}^+$. The correctness of $DMatch^+$ is proven in *Theorem 3.3*.

Theorem 3.3: *Taking $\triangle G_{D_E}^+$ and $\triangle G_{D_N}^+$ as input, $DMatch^+$ can deliver correct $N_{u_i}^{'}$ based on G_{P_new} and G_{D_new}.*

Algorithm 3: DMatch+

Input: G_P, G_D, N_{u_i}, $e(u,v) \in \triangle G^+_{D_E}$, $SLen_{new}$, AFF
Output: $N^{'}_{u_i}$

1 Set $AddSet()$ = $\emptyset$;
2 **for** *each pair of node* $(u_i, v_j) \in AFF$ *with* $AFF[u_i, v_j] = [a, b]$ **do**
3   **if** $a > f_e(u,v)$ *and* $a \leq f_e(u,v)$ **then**
4   Add u_i, v_j into $AddSet()$;
5   **if** $(v,u) \in G_P$ **then**
6   **if** $SLen_{new}(v_j, u_i) > f_e(v,u)$ **then**
7   remove u_i, v_j from $AddSet()$;
8 **for** *each node* $v_i \in AddSet()$ **do**
9   **if** (v,u)∈ G_P **then**
10   **for** *each* $u_j \notin N_{u_i}$ **do**
11   **if** $SLen_{new}(v_i, u_j) \leq f_e(v,u)$ **then**
12   Add the u_j into $AddSet()$;
13   **if** (u,v)∈ G_P **then**
14   **for** *each* $u_j \notin N_{u_i}$ **do**
15   **if** $SLen(u_j, v_i) \leq f_e(u,v)$ **then**
16   Add the u_j into $AddSet()$;
17   **if** *There is no newly added node in* $AddSet()$ **then**
18   Break;
19 **return** $N^{'}_{u_i}$ = $N_{u_i} \cup Addset()$;

The Proof of Theorem 3.3: Let $N^{''}_{u_i}$ denote the correct node matching results based on the updated pattern graph G_{P_new} and the updated data graph G_{D_new}. Suppose $N^{'}_{u_i}$ $\neq N^{''}_{u_i}$, then there is at least one node v such that (1) $v \in N^{''}_{u_i}$ and $v \notin N^{'}_{u_i}$; or (2) $v \notin N^{''}_{u_i}$ and $v \in N^{'}_{u_i}$. If $v \notin N^{'}_{u_i}$, since $N_{u_i} \subseteq N^{'}_{u_i}$, then $v \notin N_{u_i}$. If $v \in N^{''}_{u_i}$ with $v \notin N_{u_i}$, since $N_{u_i} \subseteq N^{''}_{u_i}$, then v is the new matching node due to $\triangle G^+_{D_E}$ and/or $\triangle G^+_{D_N}$. Thus, $v \in AddSet()$. Since $N^{'}_{u_i}$ = $N_{u_i} \cup Addset()$, then $v \in N^{'}_{u_i}$. This contradicts $v \notin N^{'}_{u_i}$. If $v \notin N^{''}_{u_i}$, since $N_{u_i} \subseteq N^{''}_{u_i}$, then $v \notin N_{u_i}$. Since $N^{'}_{u_i}$ = $N_{u_i} \cup$ $Addset()$, if $v \in N^{'}_{u_i}$, then $v \in AddSet()$. Thus v is the new matching node due to $\triangle G^+_{D_E}$ and/or $\triangle G^+_{D_N}$. This contradicts $v \notin N^{''}_{u_i}$. Therefore, *Theorem 3.3* is proven. □

Example 3.10: Recall G_P and G_D in Fig.3.1 (a) and Fig.3.1 (c). Suppose $\triangle G^{+}_{D_E}$ = $\{e(S_1, TE_2)\}$. (1) For $(SE_1, TE_2) \in AFF$, we have $AFF[SE_1, TE_2] = [\infty, 2]$. Since $f_e(SE, TE) = 4$, we add SE_1 and TE_2 into $AddSet()$. (2) Since $e(TE, SE) \in G_P$, and we find $SLen_{new}(TE_2, SE_1) \leq f_e(TE, SE)$, then we keep TE_2 and SE_2 in $AddSet()$. (3) Since PM_1, PM_2 and TE_1 are already in N_{u_i}, no new node can be added into $AddSet()$. Then the results of $N^{'}_{u_i}$ is by matching PM_1, PM_2 to PM; SE_1, SE_2 to SE; S_1 to S and TE_1, TE_2 to TE.

Complexity: The time complexity of the establishment and updates of the index is $\mathcal{O}(|N_D|(|N_D| + |E_D|)$ [6]. In the worst case, for each edge $e \in \triangle G^{+}_{D_E}$, $DMatch^{+}$ needs to check $SLen_{new}$ for each pair of nodes in AFF. Since the number of nodes in AFF is bounded in $|N_D|$, the time complexity of $DMatch^{+}$ is $\mathcal{O}(|N_D|(|N_D| + |E_D|) + |\triangle G^{+}_{D_E}||N_D|^2)$.

DMatch^{-}: If $v \notin N_{u_i}$, which means v does not satisfy the constraints before G_D is updated, given $\triangle G^{-}_{D_E}$ and/or $\triangle G^{-}_{D_N}$, the shortest path lengths between v and other nodes in G_D keep unchanged or increase, thus, v still does not satisfy the constraints. Therefore, $DMatch^{-}$ investigates the affected nodes in N_{u_i} and removes the nodes that do not satisfy the constraints of the pattern graph. The pseudo-code ***DMatch^{-}*** is shown in *Algorithm 4* and explained below.

For each edge $\boldsymbol{e(u, v) \in \triangle G^{-}_{D_E}}$,

- **Step 1:** For each pair of node (u_i, v_j) in AFF, if $(u_i, v_j) \in N_{u_i}$ and $SLen_{new}(u_i, v_j) > f_e(u, v)$, then (a) when there is no other node v_n in N_{u_i} such that $SLen_{new}(u_i, v_n) \leq f_e(u, v)$, we add u_i into the *candidate set of newly deleted nodes*, denoted as $DeleteSet()$; (b) when there is no other node u_n in N_{u_i} such that $SLen_{new}(u_n, v_j) \leq f_e(u, v)$, we add u_i into $DeleteSet()$ (*lines 2-9 in Algorithm 4*);

- **Step 2:** For each node v_i in $DeleteSet()$, if there is a pattern edge $e(v, u)$ or $e(u, v)$ in G_P, and v_i is the only node such that $SLen_{new}(v_i, u_j) \leq f_e(v, u)$ or

Algorithm 4: DMatch^{-}

Input: G_P, G_D, N_{u_i}, $e(u,v) \in \triangle G^{-}_{D_E}$, $SLen_{new}$, AFF
Output: $N^{'}_{u_i}$

```
1  Set DeleteSet() = ∅;
2  for each pair of node (u_i, v_j) ∈ AFF do
3      if (u_i, v_j) ∈ N_{u_i} and SLen_new(u_i, v_j) > f_e(u, v) then
4          for each v_n ∈ N_{u_i}(n ≠ j) do
5              if There is no v_n such that SLen_new(u_i, v_n) ≤ f_e(u, v) then
6                  Add u_i into DeleteSet();
7              for each u_n ∈ N_{u_i}(n ≠ i) do
8                  if There is no u_n such that SLen_new(u_n, v_j) ≤ f_e(u, v) then
9                      Add v_j into DeleteSet();
10 for each node v_i ∈ DeleteSet() do
11     if (v, u) ∈ G_P and v_i is the only node such that SLen_new(v_i, u_j) ≤ f_e(v, u)
       then
12         Add u_j into DeleteSet();
13     if (u, v) ∈ G_P and v_i is the only node such that SLen_new(u_j, v_i) ≤ f_e(u, v)
       then
14         Add u_j into DeleteSet();
15     if There is no newly added node in DeleteSet() then
16         Break;
17 return N'_{u_i} = N_{u_i} \ DeleteSet();
```

$SLen_{new}(u_j, v_i) \leq f_e(u,v)$, we add u_j into $DeleteSet()$ (*lines 10-14 in Algorithm 4*);

- **Step 3:** *DMatch^{-}* recursively performs *Step 2* to identify the nodes that need to be removed, and *DMatch^{-}* terminates when no new node can be added into $DeleteSet()$. Then, INC-GPNM returns the new GPNM result $N^{'}_{u_i}$ *(lines 15-17 in Algorithm 4).*

For each node $v \in \triangle G^{-}_{D_N}$, the corresponding edges are removed, where v is the start node or the end node of the edges. The procedure of these deleted edges is the same as the above mentioned method for $\triangle G^{-}_{D_E}$. The correctness of $DMatch^{-}$ is proven in *Theorem 3.4*.

Theorem 3.4: *Taking $\triangle G^-_{D_E}$ and $\triangle G^-_{D_N}$ as input, $DMatch^-$ can deliver correct $N^{'}_{u_i}$ based on G_{P_new} and G_{D_new}.*

The Proof of Theorem 3.4: Let $N^{''}_{u_i}$ denote the correct matching results based on the updated pattern graph G_{P_new} and the updated data graph G_{D_new}. Suppose $N^{'}_{u_i} \neq N^{''}_{u_i}$, then there is at least one node v such that (1) $v \in N^{''}_{u_i}$ and $v \notin N^{'}_{u_i}$; or (2) $v \notin N^{''}_{u_i}$ and $v \in N^{'}_{u_i}$. If $v \in N^{''}_{u_i}$, since $N^{''}_{u_i} \subseteq N_{u_i}$, then $v \in N_{u_i}$. As $N^{'}_{u_i}$ = $N_{u_i} \backslash$ $DeleteSet()$, if $v \notin N^{'}_{u_i}$, then $v \in DeleteSet()$, which means the shortest path length between v and another node u in G_{D_new} cannot satisfy the corresponding bounded path length due to $\triangle G^-_{D_E}$ and/or $\triangle G^-_{D_N}$. This contradicts $v \in N^{''}_{u_i}$. If $v \notin N^{''}_{u_i}$, since $N^{''}_{u_i} \subseteq N_{u_i}$, then $v \notin N_{u_i}$ or $v \in N_{u_i} \setminus N^{''}_{u_i}$. If $v \notin N_{u_i}$, as $N^{'}_{u_i}$ = $N_{u_i} \backslash$ $DeleteSet()$, then $v \notin N^{'}_{u_i}$. This contradicts $v \in N^{'}_{u_i}$. If $v \in N_{u_i} \setminus N^{''}_{u_i}$, which means the shortest path length between v and another node u in G_{D_new} cannot satisfy the corresponding bounded path length due to $\triangle G^-_{D_E}$ and/or $\triangle G^-_{D_N}$, then $v \in DeleteSet()$. Since $N^{'}_{u_i}$ = $N_{u_i} \backslash$ $DeleteSet()$, $v \notin N^{'}_{u_i}$. This contradicts $v \in N^{'}_{u_i}$. Therefore, *Theorem 4* is proven. □

Example 3.11: Recall G_P and G_D given in Fig.3.1 (a) and Fig.3.1 (c). Suppose $\triangle G^-_{D_E}$ = $\{e(PM_2, SE_1)\}$. (1) For $(PM_2, SE_1) \in AFF$, we find $SLen_{new}(PM_2, SE_1) = \infty$ which is greater than $f_e(PM, SE)$, and there is no other node v_j in N_{u_i} that has same label with v_i and $SLen_{new}(PM_2, v_j) \leq 3$. Thus PM_2 is added into $DeleteSet()$. Since $SLen_{new}(PM_1, SE_1) \leq 3$, then SE_1 can be kept in N_{u_i}. (3) After PM_2 is added into $DeleteSet()$, since $SLen_{new}(PM_1, SE_1)$, $SLen_{new}(PM_1, SE_2)$ and $SLen_{new}(PM_1, S_1)$ all can satisfy the bounded path length, no new node can be added into $DeleteSet()$. Then the results of $N^{'}_{u_i}$ is by matching PM_1 to PM; SE_1, SE_2 to SE; S_1 to S and TE_1 to TE.

Complexity: In the worst case, for each edge $e \in \triangle G^-_{D_E}$, $DMatch^-$ needs to check $SLen_{new}$ for each pair of nodes in AFF. Since $DMatch^-$ and $DMatch^+$ have

the same index, the time complexity of $DMatch^{-}$ is $\mathcal{O}(|N_D|(|N_D| + |E_D|) + |\triangle G^{-}_{P_E}||N_D|^2)$.

3.3 Experiments on INC-GPNM

We have conducted experiments on seven real-world social graphs to evaluate the performance of our INC-GPNM in solving the GPNM problem with the updates of both pattern graphs and data graphs.

3.3.1 Experimental Setting

Datasets: The seven real-world social graphs are available at *snap.stanford.edu*. Their details are shown in Table 3.3.

- *CollegeMsg*: This dataset is comprised of private messages sent on an online social network at the University of California, Irvine.
- *email-Eu-core*: The network was build up using email data from a large European research institution.
- *Math Overflow*: This is a temporal network of interactions on the stack exchange web site *Math Overflow*.
- *Ask Ubuntu*: This is a temporal network of interactions on the stack exchange web site *Ask Ubuntu*.
- *Super User*: This is a temporal network of interactions on the stack exchange web site *Super User*.
- *Wiki Talk*: This is a temporal network representing Wikipedia users editing each other's Talk page.
- *LiveJournal*: LiveJournal is a free on-line blogging community where users declare friendship each other.

Table 3.3: The sizes of datasets for INC-GPNM

Name	#Nodes	#Edges
CollegeMsg	1,899	59,835
email-Eu-core	5,613	151,635
Math Overflow	24,818	506,550
Ask Ubuntu	159,316	964,437
Super User	194,085	1,443,339
Wiki Talk	1,140,149	7,833,140
LiveJournal	4,847,571	68,993,773

To the best of our knowledge, there are no existing real-life dynamic data-sets with the labels. As there are no fixed patterns for the labels in a social network, and without loss of generality, the well-known existing works in GPSM [39, 41] and GPNM [40] randomly set the classes of labels in their datasets for experiments. In our experiments, we randomly set the labels of the nodes. We found that different numbers of classes of labels have a negligible impact on the query processing time. Due to page limitations, we report the results with 20 classes of labels.

Pattern Graph Generation and Parameter Setting: We use a graph generator, *socnetv*[2], to generate pattern graphs, controlled by 3 parameters: (1) the number of nodes, (2) the number of edges, and (3) the bounded path length on each edge. Since the numbers of nodes and edges in a pattern graph are usually not large [39], the number of nodes is set between 6 and 10, and the number of edges is set between 6 and 10 as well. Let (N_G, E_G) denote the scale of a graph G, where N_G is the number of nodes and E_G is the number of edges. Since the bounded path length on each edge is usually a small integer [39], we randomly set the bounded path length on each edge from 1 to 3.

Updates of G_P and G_D:

(a) Updates of G_P: At each update, we delete m_P nodes and n_P edges from G_P, and add m_P new nodes and n_P new edges into G_P, where $1 \leq m_P \leq 5$, and $1 \leq n_P \leq 5$.

[2]https://socnetv.org/

(b) Updates of G_D: At each update, we delete m_G nodes and n_G edges from G_D, and add m_G new nodes and n_G new edges into G_D. m_G increases from 20 to 100 with a step of 20 and n_G increases from 200 to 1000 with a step of 200.

Comparison Method: As discussed in *Section 2*, there is no existing incremental graph pattern based nodes matching method in the literature. Therefore, in the experiments, we have implemented the most promising state-of-the-art graph pattern based nodes matching method proposed in [40] as the **BaseLine** method, and then we compared the query processing time of the BaseLine method with that of our proposed INC-GPNM in delivering node matching results with the updates of G_P and G_D.

Implementation: Both the algorithms have been implemented using gcc 4.8.2 running on a server with Intel Xeon-E5 2630 2.60GHz CPU, 256GB RAM, and Red Hat 4.8.2-16 operating system.

Complexity Comparison of BaseLine and INC-GPNM: The time complexity for INC-GPNM is $\mathcal{O}(|N_D|(|N_D| + |E_D|) + (|\triangle G_D| + |\triangle G_P|)|N_D^2|)$. And the time complexity for BaseLine is $\mathcal{O}((|N_P| + |E_P|)(|N_D| + |E_D|) + (|N_D| + |E_D|)^2)$. Because $|\triangle G_D|$ and $|\triangle G_P|$ usually account for a very small proportion of G_D and G_P respectively, thus, INC-GPNM has a complexity of $\mathcal{O}(|N_D|(|N_D| + |E_D|) + |N_D^2|)$, and it is better than that of BaseLine. In terms of the space complexity, we use the matrix $\boldsymbol{SLen}$ to record the shortest path length between each pair of nodes in G_D, and thus it is $\mathcal{O}(|N_D|^2)$, which is the same as that of BaseLine. In addition, in real-life social networks or road networks, many nodes are not connected to other nodes (i.e., no out-degree or in-degree). Therefore, the shortest path lengths between these nodes are infinite, which makes our matrix sparse. Then, we will use some techniques to compress the sparse matrix of a given network. The Hybrid format [9] is a well-known technique that can be adopted here. A storage space of size $2|N_D||K|$ is required in Hybrid format, where $|K|$ is the maximum number of non-infinite values in a row and

$|N_D|$ is the number of nodes in a data graph. Sparse matrices can save the storage space because $|K|$ is usually much smaller than $|N_D|$.

3.3.2 Experimental Results and Analysis

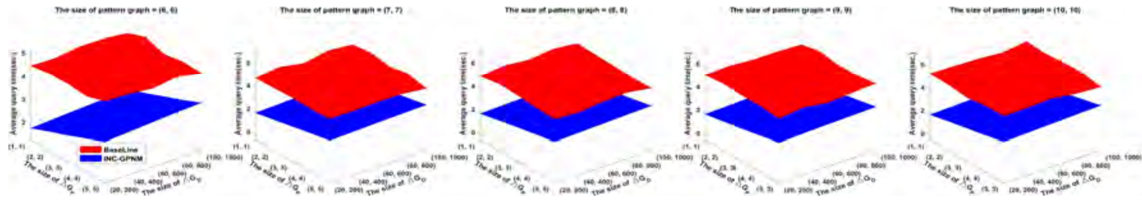


Figure 3.2: The average query processing time in CollegeMsg on INC-GPNM

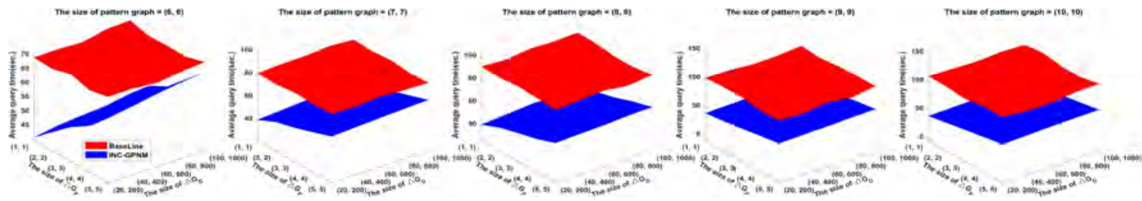


Figure 3.3: The average query processing time in email-Eu-core on INC-GPNM

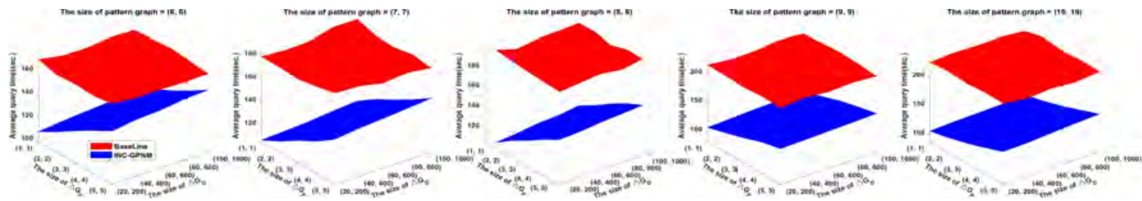


Figure 3.4: The average query processing time in Math Overflow on INC-GPNM

Fig. 3.2 to Fig. 3.8 depict the average query processing time with varying the sizes of $\triangle G_D$ and $\triangle G_P$ on different sizes of G_P. The results and analysis are as follows.

Results-1: With the increase of the size of the datasets, the average processing time of INC-GPNM is always less than that of BaseLine in all the cases of experiments. The detailed results are listed in Table 3.4. On average, INC-GPNM can reduce the query processing time by *40.24%*. The improvement remains consistent when the size

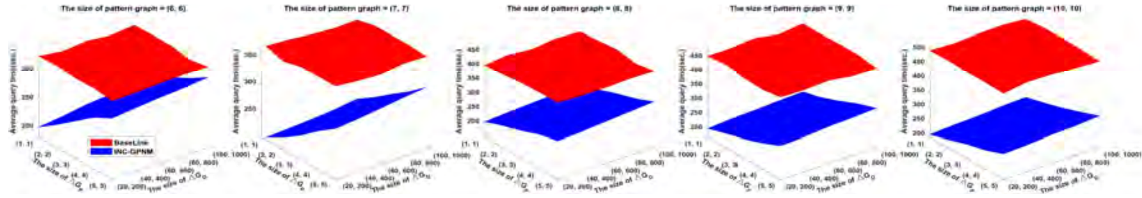


Figure 3.5: The average query processing time in Ask Ubuntu on INC-GPNM

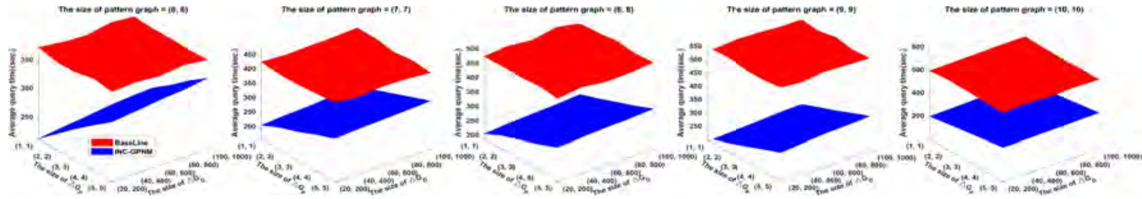


Figure 3.6: The average query processing time in Super User on INC-GPNM

of datasets has significantly increased.

Analysis-1: The results on the average query processing time for BaseLine and INC-GPNM are consistent with the complexity analysis.

Table 3.4: The average query processing time based on different scales of datasets

Dataset	BaseLine	INC-GPNM	Comparison with BaseLine
CollegeMsg	5.11s	2.64s	**48.41% less**
email-Eu-core	91.52s	54.02	**40.97% less**
Math Overview	196.94s	129.18s	**34.41% less**
Ask Ubuntu	411.42s	254.41s	**38.16% less**
Super User	493.06s	275.30s	**44.16% less**
Wiki Talk	1115.71s	713.57s	**36.04% less**
LiveJournal	5004.20s	3005.24s	**40.42% less**

Results-2: With the increase of the scale of the pattern graph from (6, 6) to (10, 10), the processing time of BaseLine increases dramatically while the processing time of INC-GPNM remains stable. The detailed results are listed in Table 3.5.

Analysis-2: With the increase of the scale of G_P, since Baseline needs to perform the whole process of GPNM to find the matching nodes for G_P, the scale of G_P has a significant influence on its query processing time. While INC-GPNM only needs to

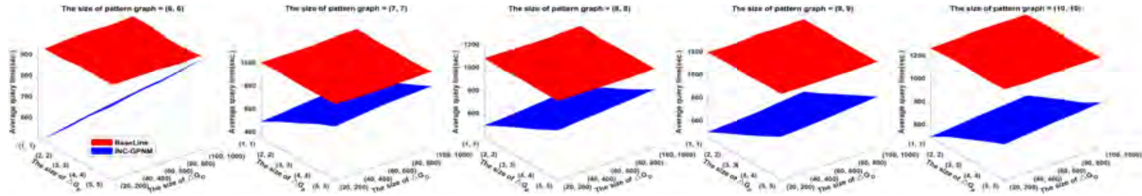


Figure 3.7: The average query processing time in Wiki Talk on INC-GPNM

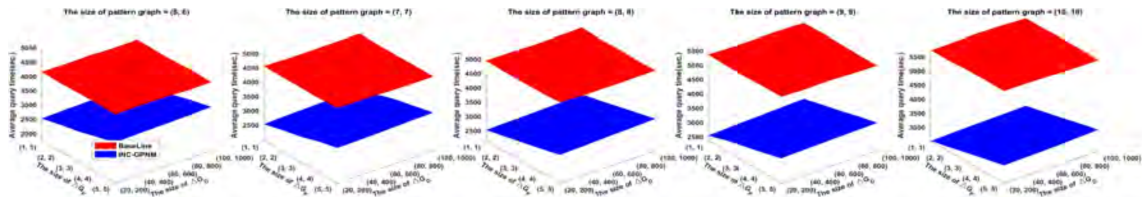


Figure 3.8: The average query processing time in LiveJournal on INC-GPNM

perform GPNM method for the affected parts of G_D. Thus, INC-GPNM has better scalability with the increase of the scale of G_P.

Results-3: With the increase of the scale of $\triangle G_P$ from (1, 1) to (5, 5) and the increase of the scale of $\triangle G_D$ from (20, 200) to (100, 1000), the processing time of INC-GPNM increases while the processing time of BaseLine remains stable. However, the processing time of INC-GPNM is still far less than that of BaseLine. The detailed results are listed in Table 3.6 and Table 3.7.

Analysis-3: When the scales of $\triangle G_P$ and $\triangle G_D$ increase, INC-GPNM needs to spend more time to investigate the affected parts, leading to more query processing time. As discussed in *Analysis-1*, the proportions of $\triangle G_D$ and $\triangle G_P$ are usually small in G_D

Table 3.5: The average query processing time based on different scale of G_P

Scale of G_P	BaseLine	INC-GPNM	Comparison with BaseLine
(6, 6)	875.51s	633.75s	**27.61% less**
(7, 7)	962.91s	633.91s	**34.17% less**
(8, 8)	1047.79s	630.41s	**39.83% less**
(9, 9)	1141.46s	639.32s	**43.99% less**
(10, 10)	1228.15s	630.01s	**48.70% less**

and G_P respectively. Therefore, although the query processing time of INC-GPNM increases, it is still far less than that of BaseLine.

Table 3.6: The average query processing time based on different scale of $\triangle G_P$

Scale of $\triangle G_P$	BaseLine	INC-GPNM	Comparison with BaseLine
(1, 1)	1051.76s	548.08s	**47.89% less**
(2, 2)	1050.57s	591.10s	**43.74% less**
(3, 3)	1051.17s	632.79s	**39.80% less**
(4, 4)	1050.55s	676.26s	**35.63% less**
(5, 5)	1051.19s	719.16s	**31.59% less**

Table 3.7: The average query processing time based on different scale of $\triangle G_D$

Scale of $\triangle G_D$	BaseLine	INC-GPNM	Comparison with BaseLine
(20, 200)	1051.26s	616.53s	**41.35% less**
(40, 400)	1051.09s	625.23s	**40.52% less**
(60, 600)	1051.47s	633.37s	**39.76% less**
(80, 800)	1050.95s	641.73s	**38.94% less**
(100, 1000)	1050.90s	650.53s	**38.09% less**

Summary: The above experimental results have demonstrated that the proposed incremental GPNM method INC-GPNM is an efficient approach to answering GPNM queries with the updates of a pattern graph and a data graph. Compared to the BaseLine method, INC-GPNM can greatly reduce the query processing time by an average of *40.24%*.

3.4 Conclusions

In this chapter, we have proposed an INCremental Graph Pattern node Matching method INC-GPNM to deliver the GPNM results based on the updates of both pattern graphs and data graphs. While keeping the space complexity unchanged, INC-GPNM has

better time complexity than the state-of-the-art GPNM method and can significantly reduce the query processing time.

Chapter 4

Incremental Graph Pattern based Node Matching Considering the Elimination Relationships Among Updates in Data Graphs

In real-world applications, many typical pattern graphs frequently and repeatedly appear in users' queries in a short period of time. For example, on Facebook, some typical queries like "Find somebody's friends, and friends of friends who like the movies Star Wars and Harry Potter" and "Find somebody's friends, and friends of friends who took photos at Sydney National Park and study at the University of Sydney" frequently and repeatedly appear in users' queries[1]. In such a situation, when facing the same subsequent query, some of the prior query answers can be reused. For example, suppose that a user asks a query "find all the users who are one kilometer away from me on Wechat" for the first time. After returning the initial results, when facing the same query ten minutes later, we can answer the query based on the initial result by only considering the changes of the users that occurred in the past ten minutes (i.e., being newly added or removed).

However, to the best of our knowledge, such a situation has not been considered in the existing methods. Even INC-GPNM [106], has to perform an incremental GPNM

[1] http://en.wikipedia.org/wiki/Facebook_Graph_Search

procedure for each of the updates in G_D, which is still computationally expensive in a large-scale social graph with high updating frequency. But, not all the updates in G_D essentially affect the GPNM matching results. We analyze this point further in the following two cases.

Case 1: If one edge (node) is firstly removed from (or inserted into) G_D and then inserted back to (or removed from) G_D, the effects of the two updates eliminate each other.

Case 2: If the set of affected nodes of an update U_a in G_D covers the set of affected nodes of a subsequent update U_b, then U_a eliminates U_b as well.

In both cases, we refer to the relationship between such two updates as an *elimination relationship*.

Following the above analysis, when facing typical queries that are frequently and repeatedly given by users, we can compute the GPNM result for the first incoming query (termed as the *initial query*), and then deliver the GPNM result for a *subsequent query* by analyzing the elimination relationships of all the updates that occur between the initial query and a subsequent query, instead of investigating each of the updates between them separately. Example 4.1 below illustrates the details of our motivations.

Example 4.1 (GPNM with multiple updates): Suppose on Facebook, there is a query "Find the people, who can connect with me within two hops, and who has been taken photos at Sydney National Park and study at the University of Sydney (USYD)" given by *Adam* (a staff in a travel agency) for the travel lines recommendation. Fig. 4.1(a) is the pattern graph corresponding to the initial query. The initial data graph is shown in Fig. 4.1(c), and the matching result for the initial query based on the initial data graph is shown in Table 4.1. The subsequent query given by another travel agency staff *Bella* is shown in Fig. 4.1(b), and three updates occur between the initial query and the subsequent query, i.e., *Update* U_1 in Fig. 4.1(d), *Update* U_2 in Fig. 4.1(e) and *Update* U_3 in Fig. 4.1(f). Fig. 4.1(g) depicts the timeline of this process. In the first update U_a, *Fiona* takes a new photo in the National Park. In the second update

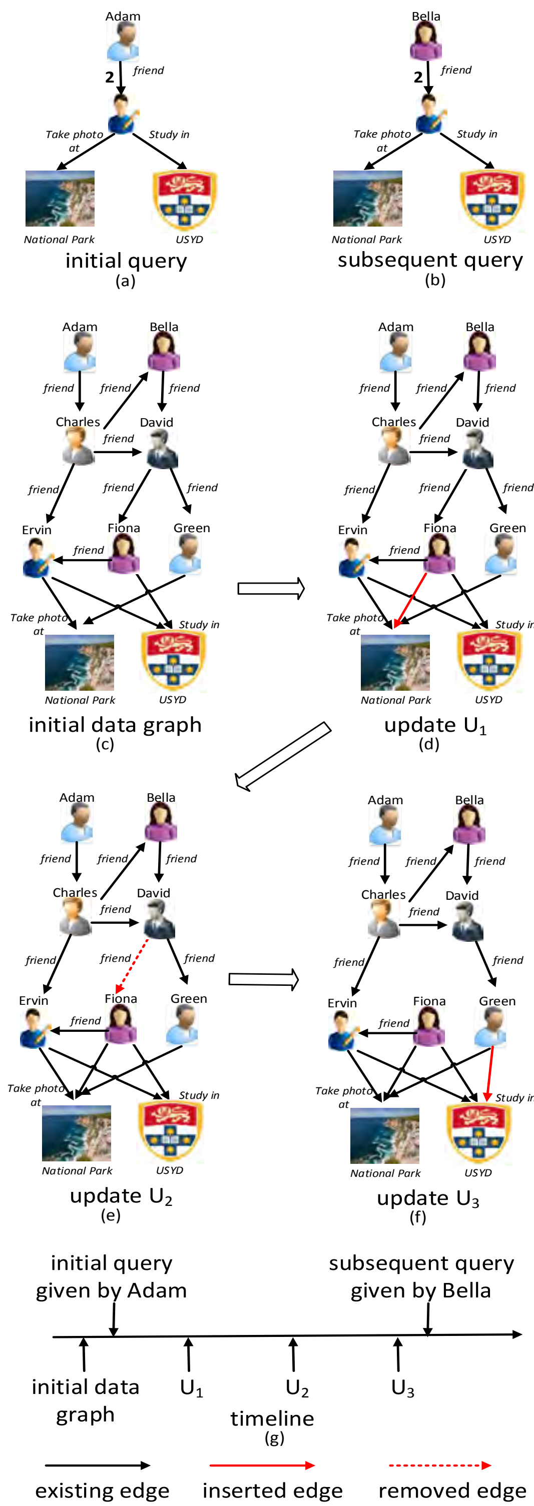


Figure 4.1: GPNM with multiple updates in data graphs

U_2, *David* removes the friend relationship with *Fiona*. In the last update U_3, *Green* is enrolled at USYD.

In order to solve the problem of computing the GPNM result for the subsequent query with multiple updates in data graph, INC-GPNM has to apply the incremental procedure three times for the three updates, leading to low efficiency. However, although *Fiona* takes a new photo in U_1, she still cannot appear in the matching result. This is because the friend relationship between *David* and *Fiona* is removed by *David* in U_2, and then *Fiona* cannot connect with *Bella* within two hops. Thus, the effect of U_1 *is eliminated by* the effect of U_2. The matching result of the subsequent query is shown in Table 4.1.

Table 4.1: The matching results of the initial query and the subsequent query in Example 4.1

Query	Matching nodes
initial query	*Charles, Ervin*
subsequent query	*Charles, Ervin, David, Green*

This example shows the need for a new GPNM solution that considers the multiple updates in G_D that occur between an initial query and a subsequent query to efficiently answer GPNM queries. Such a solution is significant for social graph searches in large-scale and frequently updated social networks, such as Facebook and Twitter. In this new GPNM solution, there are two major challenges.

Firstly, it is non-trivial to identify the elimination relationships between updates because, as we have analyzed in *Case 2*, the elimination relationships are not limited to the insertion and deletion of the same node and the edge. Therefore, the first challenge of this chapter is (1) *how to effectively identify the elimination relationships of multiple updates.* Secondly, if U_a eliminates U_b, and U_b eliminates U_c, there exists a hierarchical structure of the elimination relationships. As it is computationally expensive to deliver GPNM results by investigating each of the elimination relationships between the updates, it is beneficial to generate an index to record the hierarchical structure of the elimination relationships. This index structure can efficiently help identify the elimination relationships between each pair of updates. Therefore, the

second challenge of this chapter is (2) *how to generate an index structure to record the hierarchical structure of the elimination relationships, and then develop an efficient algorithm to deliver the GPNM results by making use of the index.*

4.1 Elimination Relationships in Data Graphs

In this section, we analyze the *elimination relationships* between the updates in G_D and discuss how they can be detected.

4.1.1 Elimination Relationship Types in Data Graphs

If one edge or node is firstly removed from (or inserted into) a G_D and then inserted back to (or removed from) G_D, these two updates eliminate each other (i.e., *Case 1*). Moreover, if the set of affected nodes in G_D of an update U_a covers the set of affected nodes of another update U_b, then U_a eliminates U_b as well (i.e., *Case 2*). Therefore, we classify the elimination relationships into the following two types.

Remark: When given a series of updates, we identify the affected nodes for each of the updates sequentially. For example, if U_a is prior to U_b, we first identify the affected nodes for U_a and update the initial data graph based on U_a. Then, when U_b is applied, we identify the affected nodes for U_b and update the data graph accordingly.

Elimination Relationship Type I in Data Graphs: In GPNM, we need to investigate if the shortest path length between each pair of nodes in G_D can satisfy the requirements of the bounded path length in G_P. The insertion and deletion of edges (or nodes) can have the following two situations: (1) The insertion of edges (or nodes) in G_D will decrease the shortest path length between any two nodes or keep it unchanged, and (2) the deletion of edges (or nodes) in G_D will increase the shortest path length between any two nodes or keep it unchanged. If one insertion (denoted as update U_a) and one deletion (denoted as update U_b) keep the shortest path length between any nodes

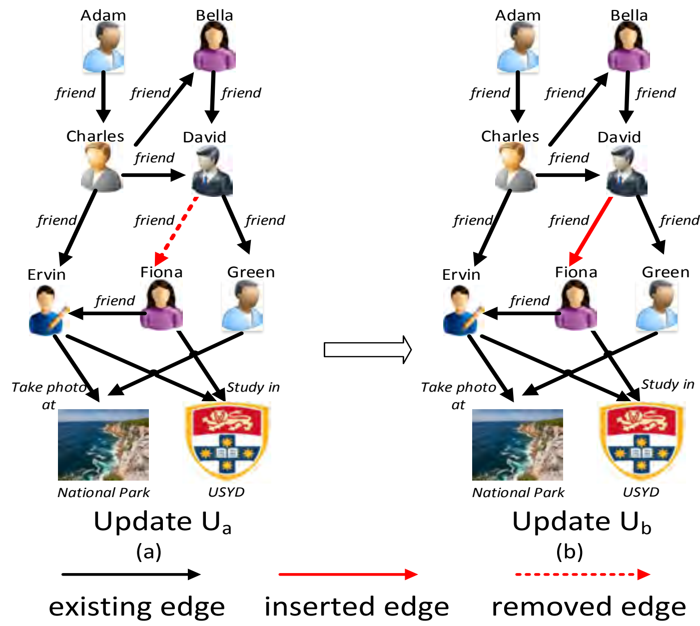


Figure 4.2: Elimination Relationship Type I in Data Graphs

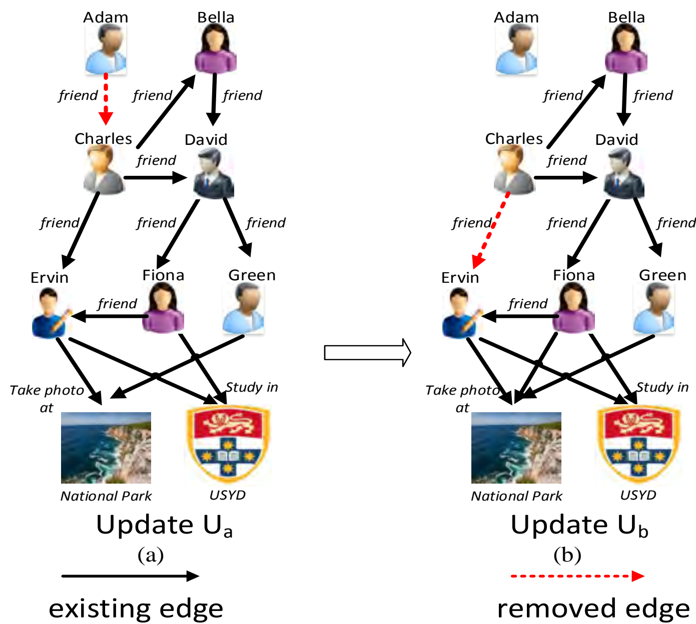


Figure 4.3: Elimination Relationship Type II in Data Graphs

unchanged, we say U_a and U_b *eliminate each other*, denoted as $U_a \Leftrightarrow U_b$.

Remark: If a node is isolated from the data graph, then the insertion and deletion of the node will not affect the shortest path lengths between other nodes. Otherwise, if the node is deleted, the corresponding edges linking this node are removed as well.

Definition 1 (Elimination Relationship Type I in Data Graphs): Given two updates $U_a \in \triangle G_D^+$ and $U_b \in \triangle G_D^-$, with an update U_i (i = a or b), if the shortest path length between two nodes has changed, we put all such nodes into the set of affected nodes, denoted as $\mathit{Aff_N}(U_i)$. $\mathit{Aff_N}(U_a, U_b)$ denotes the set of nodes where the shortest path length between a pair of any two nodes has changed caused by updates U_a and U_b. $U_a \Leftrightarrow U_b$ if and only if $\mathit{Aff_N}(U_a, U_b) = \emptyset$, which means that U_a and U_b eliminate each other if the shortest path length between any pair of nodes in G_D remains unchanged with U_a and U_b.

Example 4.2: Recall the case shown in Fig. 4.1, suppose U_a is to remove edge $e(David, Fiona)$ from G_D and U_b is to insert edge $e(David, Fiona)$ into G_D as depicted in Fig. 4.2(a) and Fig. 4.2(b) respectively. In this example, since $\mathit{Aff_N}(U_a, U_b) = \emptyset$, $U_a \Leftrightarrow U_b$.

Elimination Relationship Type II in Data Graphs: Given two updates U_a and U_b, where $U_a \in \triangle G_D^+$ (or $\triangle G_D^-$) and $U_b \in \triangle G_D^+$ (or $\triangle G_D^-$), if the set of nodes between which the shortest path lengths are affected by U_a covers the set of nodes between which the shortest path lengths are affected by U_b, we say U_a *eliminates* U_b or U_b *is eliminated by* U_a, denoted as $U_a \succeq U_b$ or $U_b \preceq U_a$.

Definition 2 (Elimination Relationship Type II in Data Graphs): Given two updates U_a and U_b, where $U_a \in \triangle G_{D_E}^+$ (or $\triangle G_{D_E}^-$) and $U_b \in \triangle G_{D_E}^+$ (or $\triangle G_{D_E}^-$), with an update U_i (i = a or b), if the shortest path between two nodes has been affected, we put these affected nodes into $\mathit{Aff_N}(U_i)$. $U_a \succeq U_b$ or $U_b \preceq U_a$ if and only if $\mathit{Aff_N}(U_a)$

$\supseteq$ *Aff_N*(U_b). Likewise, $U_a \preceq U_b$ or $U_b \succeq U_a$ if and only if *Aff_N*$(U_a) \subseteq$ *Aff_N*(U_b).

Example 4.3: Recall the case shown in Fig. 4.3. With U_a, the shortest path lengths from *Adam* to all the other nodes in G_D are affected, then *Aff_N*$(U_a) = \{Adam$, $Bella$, $Charles$, $David$, $Ervin$, $Fiona$, $Green\}$. With U_b, the shortest path lengths from *Charles* to *Ervin* are affected, then *Aff_N*$(U_b) = \{Charles, Ervin\}$. Since *Aff_N*$(U_a) \supseteq$ *Aff_N*(U_b), $U_a \succeq U_b$ or $U_b \preceq U_a$.

Remark: Although the updates in Fig. 4.1 contains the updates in Fig. 4.2, Fig. 4.2 illustrates the Elimination Relationship Type I more directly, while Fig. 4.2 contains all the queries and updates of data graphs, which is used to illustrate our targeted problem and the motivation.

4.1.2 Detecting Elimination Relationships in Data Graphs

It is clear that the elimination relationships are not limited to the insertion and deletion of the same node and the edge. Moreover, the set of affected nodes of each update is the critical factor in detecting elimination relationships, and it is non-trivial to identify the set of affected nodes. Therefore, we first generate the shortest path length matrix, $SLen$, to record the shortest path length between each pair of nodes in G_D. Since the updates of G_D can lead to a change of $SLen$, and it is very costly to re-compute the shortest path length matrix, in this paper, we adopt the method proposed in [94] to incrementally update $SLen$ when G_D is updated, which can avoid re-computing the shortest path length for the whole matrix, thereby reducing time consumption. Below we introduce the details.

Detect Elimination Relationship Type I in Data Graphs (DER-I): With two updates $U_a \in \triangle G_D^+$ and $U_b \in \triangle G_D^-$, we first adopt the method proposed in [94] to obtain the updated $SLen$ (denoted as $SLen_{new}$). Specifically, for the pair of nodes where the shortest path length between them are affected, we adopt *Dijkstra*'s algorithm to

Algorithm 5: Detect Elimination Relationship Type I (DER-I)

Input: G_P, G_D, $\triangle G_D$, $SLen$
Output: The elimination relationships of the updates

```
1 for each pair of updates U_a and U_b ∈ △G_D do
2     if the shortest path lengths between the nodes are not affected then
3         Keep the shortest path lengths in SLen_new the same as those in SLen;
4     else
5         Apply Dijkstra's algorithm for updating the shortest path lengths
          between the affected nodes in SLen_new;
6     if SLen_new = SLen then
7         U_a ⇔ U_b;
8 Return the elimination relationships of the updates;
```

update the corresponding shortest path length to obtain $SLen_{new}$. Then, we compare $SLen$ with $SLen_{new}$. If there is no update in $SLen_{new}$, then $U_a \Leftrightarrow U_b$. The pseudo-code is shown in *Algorithm 5*.

Detect Elimination Relationship Type II in Data Graphs (DER-II): With two updates $U_a \in \triangle G_D^+$ (or $\triangle G_D^-$) and $U_b \in \triangle G_D^+$ (or $\triangle G_D^-$), we first update $SLen$ for each update and then compare the updated $SLen_{new}$ with these two updates. If $\mathit{Aff_N}(U_a) \supseteq \mathit{Aff_N}(U_b)$, then $U_a \succeq U_b$ or $U_b \preceq U_a$. The pseudo-code is shown in *Algorithm 6*.

Example 4.4: Recall the case shown in Fig. 4.3. We first compute the shortest path length matrix $SLen$ for the G_D prior to the updates. The results are shown in Table 4.2. With U_a and U_b, we update the shortest path length matrices as shown in Table 4.3 and Table 4.4 respectively. Compared with $SLen$, with U_a, the shortest path lengths from *Adam* to all the other nodes are affected, then $\mathit{Aff_N}(U_a) = \{Adam, Bella, Charles, David, Ervin, Fiona, Green\}$. With U_b, the shortest path lengths from *Charles* to *Ervin* are affected, then $\mathit{Aff_N}(U_b) = \{Charles, Ervin\}$. Because $\mathit{Aff_N}(U_a) \supseteq \mathit{Aff_N}(U_b)$, then $U_a \succeq U_b$ or $U_b \preceq U_a$.

Theorem 4.1: *The order of the updates in* $\triangle G^+_{D_E}$ *and* $\triangle G^-_{D_E}$ *does not affect the cor-*

Algorithm 6: Detect Elimination Relationship Type II (DER-II)

Input: G_P, G_D, $\triangle G_D$, $SLen$
Output: The elimination relationships of the updates

```
1  for each pair of updates U_a and U_b ∈ △G_D do
2      if the shortest path lengths between the nodes are not affected then
3          Keep the shortest path lengths in SLen_new as that in SLen;
4      else
5          Apply Dijkstra's algorithm for updating the shortest path lengths
           between the affected nodes in SLen_new;
6      Put the affected nodes into Aff_N(U_a);
7      Put the affected nodes into Aff_N(U_b);
8      if Aff_N(U_a) ⊇ Aff_N(U_b) then
9          U_a ⪰ U_b;
10 Return the elimination relationships of the updates;
```

Table 4.2: $SLen$ of G_D in Fig. 4.1(c)

	Adam	Bella	Charles	David	Ervin	Fiona	Green
Adam	0	2	1	2	2	3	3
Bella	∞	0	∞	1	3	2	2
Charles	∞	1	0	1	1	2	2
David	∞	∞	∞	0	2	1	1
Ervin	∞	∞	∞	∞	0	∞	∞
Fiona	∞	∞	∞	∞	1	0	∞
Green	∞	∞	∞	∞	∞	∞	0

rectness of the detection of elimination relationship Type I.

The Proof of Theorem 4.1: When U_a is applied to G_D prior to U_b, suppose $U_a \Leftrightarrow U_b$. Then, according to the definition of an elimination relationship *Type I*, there is no affected node with U_a and U_b. When U_b is applied to G_D prior to U_a, suppose U_a and U_b do not have the elimination relationship. Then there exists at least one node n_i such that (1) $n_i \in \mathit{Aff_N}(U_b)$ and $n_i \notin \mathit{Aff_N}(U_a)$; or (2) $n_i \in \mathit{Aff_N}(U_a)$ and $n_i \notin \mathit{Aff_N}(U_b)$. If $n_i \in \mathit{Aff_N}(U_b)$ and $n_i \notin \mathit{Aff_N}(U_a)$, then n_i is not affected by U_a. However, it contradicts $n_i \in \mathit{Aff_N}(U_a)$ when U_a is applied to G_D. If $n_i \in \mathit{Aff_N}(U_a)$ and $n_i \notin \mathit{Aff_N}(U_b)$, then n_i is not affected by U_b. However, this contradicts $n_i \in \mathit{Aff_N}(U_b)$ when U_b is applied to G_D. Therefore, *Theorem 4.1* is

Table 4.3: $SLen_{new}$ with U_a in Fig. 4.1(a)

	Adam	Bella	Charles	David	Ervin	Fiona	Green
Adam	0	∞	∞	∞	∞	∞	∞
Bella	∞	0	1	1	2	2	2
Charles	∞	∞	0	∞	1	1	1
David	∞	∞	∞	0	∞	∞	1
Ervin	∞	∞	∞	∞	0	∞	∞
Fiona	∞	∞	∞	∞	∞	0	∞
Green	∞	∞	∞	∞	∞	∞	0

Table 4.4: $SLen_{new}$ with U_b in Fig. 4.1(b)

	Adam	Bella	Charles	David	Ervin	Fiona	Green
Adam	0	2	1	2	∞	3	3
Bella	∞	0	∞	1	3	2	2
Charles	∞	1	0	1	∞	2	2
David	∞	∞	∞	0	2	1	1
Ervin	∞	∞	∞	∞	0	∞	∞
Fiona	∞	∞	∞	∞	1	0	∞
Green	∞	∞	∞	∞	∞	∞	0

proven. □

Theorem 4.2: *The order of the updates in* $\triangle G^{+}_{D_E}$ *and* $\triangle G^{-}_{D_E}$ *does not affect the correctness of the detection of elimination relationship Type II.*

The Proof of Theorem 4.2: When U_a is applied to G_D prior to U_b, suppose $U_a \succeq U_b$. Then, according to the definition of an elimination relationship of *Type II*, $Aff_N(U_a) \supseteq Aff_N(U_b)$, namely, for any node $n_i \in Aff_N(U_b)$, n_i is also in $Aff_N(U_a)$. When U_b is applied to G_D prior to U_a, suppose U_a and U_b do not have the elimination relationship. Then, there is at least one node n_i such that $n_i \in Aff_N(U_b)$ and $n_i \notin Aff_N(U_a)$. However, this contradicts $n_i \in Aff_N(U_a)$ when U_a is applied to G_D. Therefore, *Theorem 4.2* is proven. □

Complexity: The complexity of the generation and the updates of $SLen$ is $\mathcal{O}(|N_D|(|N_D|+$

$|E_D|$) [94]. In the worst case, *DER-I* and *DER-II* need to check $SLen_{new}$ for each update, then the complexity of either of *DER-I* and *DER-II* is $\mathcal{O}(|N_D|(|N_D| + |E_D|) + |\triangle G_D||N_D|^2)$, where $|N_D|$ and $|E_D|$ are the number of the nodes and the number of the edges respectively in G_D, and $|\triangle G_D|$ is the scale of the updates of G_D.

Summary: The above methods can detect the elimination relationship between any pair of updates. However, it is computationally expensive to investigate each of the elimination relationships for delivering GPNM results. It is worth noting that there exists a hierarchical structure between these elimination relationships. If we can generate an index to record the structure of these elimination relationships, the query processing time of subsequent queries can be reduced by investigating the structure. The following section introduces the details.

4.2 Elimination Hierarchy Tree (EH-Tree) for the Updates in Data Graphs

As introduced in Section 4.1, if the set of affected nodes of update U_a can cover that of update U_b, then U_a eliminates U_b. In addition, there is a hierarchical structure between the elimination relationships. Therefore, in our method, we first identify the affected nodes of each update, based on which, we then identify the hierarchical structure between the elimination relationships and record the elimination hierarchy by using a tree structure, which is called EH-Tree.

4.2.1 Identifying the Affected Nodes

To generate the EH-Tree index, we first need to identify the affected nodes of each update. The steps are as follows:

- **Step 1:** For each update $U_i \in \triangle G_D$, set $Aff_N(U_i) = \emptyset$, and then update the $SLen$ to obtain the $SLen_{new}$;

- **Step 2:** Compare $SLen_{new}$ with $SLen$. For any pair of nodes between which the shortest path length has been changed, put the two nodes into $Aff_N(U_i)$;
- **Step 3:** Return $Aff_N(U_i)$.

Table 4.5: $SLen_{new}$ with U_a in Example 4.5

	Adam	Bella	Charles	David	Ervin	Fiona	Green
Adam	0	∞	∞	∞	∞	∞	∞
Bella	∞	0	∞	1	3	2	2
Charles	∞	1	0	1	1	2	2
David	∞	∞	∞	0	2	1	1
Ervin	∞	∞	∞	∞	0	∞	∞
Fiona	∞	∞	∞	∞	1	0	∞
Green	∞	∞	∞	∞	∞	∞	0

Table 4.6: $SLen_{new}$ with U_b in Example 4.5

	Adam	Bella	Charles	David	Ervin	Fiona	Green
Adam	0	2	1	2	2	3	∞
Bella	∞	0	∞	1	3	2	∞
Charles	∞	1	0	1	1	2	∞
David	∞	∞	∞	0	2	1	∞
Ervin	∞	∞	∞	∞	0	∞	∞
Fiona	∞	∞	∞	∞	1	0	∞
Green	∞	∞	∞	∞	∞	∞	0

Table 4.7: $SLen_{new}$ with U_c in Example 4.5

	Adam	Bella	Charles	David	Ervin	Fiona	Green
Adam	0	2	1	2	2	∞	3
Bella	∞	0	∞	1	∞	∞	2
Charles	∞	1	0	1	1	∞	2
David	∞	∞	∞	0	∞	∞	1
Ervin	∞	∞	∞	∞	0	∞	∞
Fiona	∞	∞	∞	∞	1	0	∞
Green	∞	∞	∞	∞	∞	∞	0

Example 4.5: Recall the case shown in Fig. 4.1. Suppose U_a, U_b, U_c and U_d are to remove edge $e(Adam, Charles)$, $e(David, Green)$, $e(David, Fiona)$ and $e(Charles,$

Table 4.8: $SLen_{new}$ with U_d in Example 4.5

	Adam	Bella	Charles	David	Ervin	Fiona	Green
Adam	0	2	1	2	∞	3	3
Bella	∞	0	∞	1	3	2	2
Charles	∞	1	0	1	∞	2	2
David	∞	∞	∞	0	2	1	1
Ervin	∞	∞	∞	∞	0	∞	∞
Fiona	∞	∞	∞	∞	1	0	∞
Green	∞	∞	∞	∞	∞	∞	0

Table 4.9: The affected nodes of the updates in Example 4.5

Update	Affected nodes
U_a	Adam, Bella, Charles, David, Ervin, Fiona, Green
U_b	Adam, Bella, Charles, David, Green
U_c	Adam, Bella, Charles, David, Ervin, Fiona
U_d	Adam, Charles, Ervin

$Ervin$) from G_D respectively. The corresponding $SLen_{new}$ are shown in Table 4.5, Table 4.6, Table 4.7 and Table 4.8 respectively. With U_a, the shortest path lengths from *Adam* to other nodes are affected, then $Aff_N(U_a) = \{Adam, Bella, Charles, David, Ervin, Fiona, Green\}$; with U_b, the shortest path lengths from *Adam*, *Bella*, *Charles*, *David* to *Green* are affected, then $Aff_N(U_b) = \{Adam, Bella, Charles, David, Green\}$; with U_c, the shortest path lengths from *Adam*, *Bella*, *Charles*, *David* to *Fiona* and the shortest path lengths from *Bella*, *David* to *Ervin* are affected, then $Aff_N(U_c)$=$\{Adam, Bella, Charles, David, Ervin, Fiona\}$; with U_d, the shortest path lengths from *Adam* to *Ervin*, and from *Charles* to *Ervin* are affected, then $Aff_N(U_d) = \{Adam, Charles, Ervin\}$. The affected nodes of all the four updates are listed in Table 4.9.

4.2.2 EH-Tree For the Updates in Data Graphs Establishment

We present the details of the generation of EH-Tree as follows. The pseudo-code is shown in *Algorithm 7*.

- **Step 1:** Firstly, for each update, we use the above-mentioned method to identify

the affected nodes. Each tree node in EH-Tree denotes one update and stores the affected nodes of the update.

- **Step 2:** Based on the affected nodes of each update, EH-Tree adopts the property of balance tree to improve the query efficiency. Then, we have the following strategies: (a) the update that has the maximum number of affected nodes is set as the root of an EH-Tree; (b) if the affected nodes of one update can be covered by the root, then this update is set as a child tree node of the root; (c) the number of affected nodes of one update in each tree node must be greater than or equal to the number of affected nodes of its *left* tree node, and less than or equal to the number of affected nodes of its *right* tree node.

- **Step 3:** We then recursively insert all the updates into the EH-Tree.

Example 4.6: Recall U_a, U_b, U_c and U_d in Example 4.5. As U_a has the maximum number of affected nodes in all the updates, it is set as the root of EH-Tree; with U_b, as the set of affected nodes of U_a covers that of U_b, U_b is set as the *left* child node of U_a as shown in Fig. 4.4(a); with U_c, as the set of affected nodes of U_a also covers that of U_c, and U_c has the larger number of affected nodes than U_b, U_c is set as the *right* child node of U_a as shown in Fig. 4.4(b); with U_d, as the set of affected nodes of U_c covers that of U_d, U_d is set as the *left* child node of U_c. The completed EH-Tree is shown in Fig. 4.4(c).

4.2.3 EH-Tree For the Updates in Data Graphs Maintenance

After building an EH-Tree index, next, we introduce how to maintain the existing EH-Tree when facing new updates.

In an EH-Tree, each tree node represents one update in data graphs. When facing a new coming update U_n, we need to decide where it should be inserted as a new tree node into the existing EH-Tree. As the elimination relationship is illustrated by the affected nodes of updates, we compare the set of affected nodes between U_n and the

Algorithm 7: EH-Tree For the Updates in Data Graphs Establishment

Input: G_P, G_D, $\triangle G_D$, $SLen$

Output: the address of the root of an EH-Tree

```
1  for each update U_i ∈ △G_D do
2      if the shortest path lengths between the the nodes are not affected then
3          Keep the shortest path lengths in SLen_new as that in SLen;
4      else
5          Apply Dijkstra's algorithm for updating the shortest path lengths
           between the affected nodes in SLen_new;
6      Compare SLen_new with SLen;
7      Put the affected nodes into Aff_N(U_i);
8      if U_i has the maximum number of affected nodes then
9          U_i is set as the root of an EH-Tree;
10     else
11         for each update U_j ∈ △G_D do
12             if the set affected nodes of U_i covers that of U_j then
13                 U_j is set as the child node of U_i;
14                 if the number of affected nodes of U_j is less than or equal to that
                   by other child nodes then
15                     U_j is set as the left child node;
16                 else
17                     U_j is set as the right child node;
18 return the address of the root of an EH-Tree;
```

updates in the existing EH-tree to insert U_n. The detailed steps are shown as follows. The corresponding pseudo-code is shown in *Algorithm 8*.

(1) Firstly, for each new coming update U_n, we identify $Aff_N(U_n)$ based on the method proposed in Section 4.2.1.

(2) Let U_m denote one of the existing updates, then, we compare $Aff_N(U_n)$ with $Aff_N(U_m)$. Based on the comparison results, we perform the following processes:

(a) Let U_{root} denote the root in the existing EH-Tree. Set $U_m = U_{root}$, if $Aff_N(U_n) \supseteq Aff_N(U_m)$, then U_n is set as the new root of the EH-Tree, and the existing root is set as the only child node of U_n. Otherwise, go to (b).

Algorithm 8: EH-Tree For the Updates in Data Graphs Maintenance

Input: a new coming update U_n, EH-Tree
Output: the updated EH-Tree

1 Identify $Aff_N(U_n)$;
2 let U_m denote one of the tree nodes in EH-Tree;
3 let U_{root} denote the tree root in EH-Tree;
4 set $U_m = U_{root}$;
5 **if** $Aff_N(U_n) \supseteq Aff_N(U_m)$ *and* U_m *is the tree root in EH-Tree* **then**
6   U_n is set as the new root of EH-Tree;
7   U_m is set as the only child node of U_n;
8 **else**
9   **if** $Aff_N(U_n) \subset Aff_N(U_m)$ **then**
10   let U_l denote the one of the left child nodes of U_m;
11   let U_r denote the one of the right child nodes of U_m;
12   **if** $Aff_N(U_n) \supseteq Aff_N(U_r)$ *and* $Aff_N(U_n) \supseteq Aff_N(U_l)$ **then**
13   U_n is set as the only child node of U_m;
14   U_r, U_l is set as the right, left child node of U_n;
15   **else**
16   **if** $Aff_N(U_n) \supseteq Aff_N(U_r)$ *and* $Aff_N(U_n) \not\supseteq Aff_N(U_l)$ **then**
17   U_i is set as the right child of U_m;
18   U_r is set as the only child node of U_n;
19   **else**
20   **if** U_r *is not the leaf node in EH-Tree and* $Aff_N(U_n) \subset Aff_N(U_r)$ **then**
21   INSERT(U_n, U_r);
22   **if** U_l *is not the leaf node in EH-Tree and* $Aff_N(U_n) \subset Aff_N(U_l)$ **then**
23   INSERT(U_n, U_l);
24   **if** $Aff_N(U_n) \not\supseteq Aff_N(U_r)$ *and* $Aff_N(U_n) \supseteq Aff_N(U_l)$ **then**
25   U_n is set as the left child node of U_m;
26   U_l is set as the only child node of U_n;
27   **else**
28   **if** U_r *is not the leaf node in EH-Tree and* $Aff_N(U_n) \subset Aff_N(U_r)$ **then**
29   INSERT(U_n, U_r);
30   **if** U_l *is not the leaf node in EH-Tree and* $Aff_N(U_n) \subset Aff_N(U_l)$ **then**
31   INSERT(U_n, U_l);
32 **return** the updated EH-Tree;

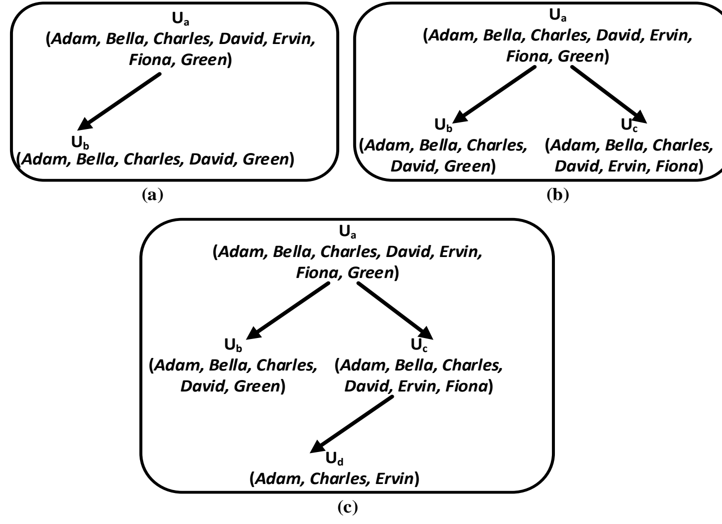


Figure 4.4: The EH-Tree of Example 4.6

(b) If $Aff_N(U_n) \subset Aff_N(U_m)$, let U_l denote one of the left children nodes of U_m and let U_r denote one of the right children nodes of U_m, if $Aff_N(U_n) \supseteq Aff_N(U_r)$ and $Aff_N(U_n) \supseteq Aff_N(U_l)$, then U_n is set as the only child node of U_m, and U_r, U_l are set as the right and left children nodes of U_n respectively. Otherwise,

- if $Aff_N(U_n) \supseteq Aff_N(U_r)$ and $Aff_N(U_n) \not\supseteq Aff_N(U_l)$, then U_n is set as the right child of U_m and U_r is set as the only child node of U_n;
- if $Aff_N(U_n) \not\supseteq Aff_N(U_r)$ and $Aff_N(U_n) \supseteq Aff_N(U_l)$, then U_n is set as the left child node of U_m and U_l is set as the only child node of U_n.

(c) If $Aff_N(U_n) \subset Aff_N(U_r)$, set $U_r = U_m$ and go to (a) until reaching one of the leaf nodes of the EH-Tree; If $Aff_N(U_n) \subset Aff_N(U_l)$, set $U_l = U_m$ and go to (a) until reaching one of the leaf nodes of the EH-Tree.

Example 4.7: Recall U_a, U_b, U_c and U_d in Example 4.5. The corresponding EH-Tree of these updates is shown in Fig. 4.4 (c). Suppose the new coming update U_n is a newly added edge $e(Bella, Green)$ in G_D. As $Aff_N(U_n) \subset Aff_N(U_a)$, we need to compare $Aff_N(U_n)$ with $Aff_N(U_b)$, and $Aff_N(U_c)$ respectively. As $Aff_N(U_n) \subset$

$Aff_N(U_b)$ and U_b is a leaf node, then U_n is set as the only child node of U_b.

Complexity: Since an EH-Tree is a balanced tree, the time complexity of the tree establishment and search are $\mathcal{O}|\triangle G_D|\log|\triangle G_D|$ and $\mathcal{O}\log|\triangle G_D|$ respectively [1], where $|\triangle G_D|$ is the number of updates in G_D. For each update in EH-Tree, we need to save all the nodes in the data graph maximally, therefore, the space complexity of EH-Tree is $\mathcal{O}|\triangle G_D||N_D|$.

4.3 EH-GPNM Algorithm

4.3.1 Algorithm Overview

EH-GPNM contains three major parts, each solving a challenging problem. All of them demonstrate the novelty and effectiveness of our proposed EH-GPNM. Firstly, since the elimination relationships are not limited to the insertion and deletion of the same node and the edge and it is non-trivial to identify the affected nodes of each update, EH-GPNM has a strategy to identify the affected nodes for each update. Based on the affected nodes, EH-GPNM can detect the elimination relationships effectively. Then, since it is computationally expensive to deliver GPNM results by investigating each of the elimination relationships between the updates, EH-GPNM generates a tree index (EH-Tree) to record the hierarchical structure of the elimination relationships. Finally, by searching the EH-Tree, our EH-GPNM algorithm applies an incremental GPNM procedure for the rest of the updates to identify the GPNM results without any need to consider the effect of the eliminated updates. This approach can greatly save query processing time (see details in Section 4.4).

4.3.2 The Process of EH-GPNM

After building the EH-Tree, when facing a subsequent query, EH-GPNM first searches the EH-Tree to efficiently detect the elimination relationships between multiple

Algorithm 9: EH-GPNM

Input: G_P, G_D, $SQuery$, $\triangle G_D$
Output: $SQuery$
1 Generate an EH-Tree;
2 **for** *each* $U_i \in \triangle G_D$ **do**
3 Check the EH-Tree;
4 **if** U_i *is the parent node of* U_j $(i \neq j)$ **then**
5 U_i can eliminate U_j;
6 Incrementally identifies the GPNM results for the updates;
7 **return** $SQuery$;

updates and then incrementally identifies the GPNM results. The detailed steps of EH-GPNM are shown below. The pseudo-code is shown in *Algorithm 9*.

- **Step 1:** For each update $U_i \in \triangle G_D$, EH-GPNM first searches the EH-Tree to detect the elimination relationships between the updates.
- **Step 2:** EH-GPNM then recursively finds the elimination relationship for each update until all the updates in $\triangle G_D$ have been investigated.
- **Step 3:** After searching the EH-Tree, we do not need to consider the effect of the eliminated updates. We apply the following incremental GPNM procedure for the rest of the updates to identify the GPNM results. The details of the incremental GPNM procedure are introduced in *Chapter 3*.

Complexity: (1) **Time Complexity:** Since EH-GPNM first searches the EH-Tree, and then incrementally identifies the GPNM results for the updates, EH-GPNM achieves $\mathcal{O}(|N_D|(|N_D| + |E_D|) + (|\triangle G_D| - |U_D|)(|N_D|^2) + |\triangle G_D| \log |\triangle G_D|)$ in time complexity, where $|U_D|$ is the number of the updates that can be eliminated in G_D.
(2) **Space Complexity:** Since EH-GPNM uses a matrix structure to record the shortest path length between each pair of nodes and generates a balanced tree structure to index the elimination relations, its space complexity is $\mathcal{O}((|N_D|^2 + |\triangle G_D||N_D|)$. In addition, in real-world social networks, not all the pairs of nodes are reachable (i.e., the shortest path lengths between these pairs of nodes are taken as infinite), which makes

the shortest path length matrix $SLen$ sparse. Therefore, we can compress the sparse matrix of a data graph by using the Hybrid format [9]. Then the space complexity can be reduced to $2|N_D||K|$, where $|K|$ is the maximum number of non-infinite values in a row and $|N_D|$ is the number of nodes in a data graph. By this method, we can save the storage space because $|K|$ is usually much smaller than $|N_D|$.

Example 4.8: Recall the case shown in Fig. 4.1, and suppose there are four updates between the two queries (i.e., U_a, U_b, U_c and U_d in Example 4.5). The EH-Tree of these four updates is shown in Fig. 4.4(c), where U_a is the root, that is, U_a eliminates all the other updates. Therefore, our method only needs to apply the incremental GPNM procedure for the update U_a, which can greatly save the query processing time.

4.4 Experiments on EH-GPNM

4.4.1 Experimental Setting

Datasets: We used five real-world social graphs that are available at *snap.stanford.edu*. The details are shown in Table 4.10.

Table 4.10: The sizes of datasets on EH-GPNM

Name	#Nodes	#Edges
Ask Ubuntu	159,316	964,437
Facebook	134,833	1,380,293
Super User	194,085	1,443,339
Wiki Talk	1,140,149	7,833,140
LiveJournal	4,847,571	68,993,773

To the best of our knowledge, there are no existing real-world datasets with labels. As there are no fixed patterns for the labels in a social network, without loss of generality, the well-known existing works in GPSM [39, 41] and GPNM [40, 106] randomly set the classes of labels in their data-sets for experiments. Similarly, we randomly set the labels of the nodes. For each dataset, we set the number of labels as 20, 40, 60, 80

and 100 respectively.

Pattern Graph Generation and Parameter Setting: We used a graph generator, *socnetv*[2], to generate pattern graphs, controlled by 3 parameters: (1) the number of nodes, (2) the number of edges, and (3) the bounded path length on each edge. Since the numbers of nodes and edges in a pattern graph are usually not large [39], they are set between 6 and 10. Since the bounded path length on each edge is usually a small integer [39], we randomly set the bounded path length on each edge from 1 to 3.

Updates of G_D: In each experiment, we removed m edges and m nodes from G_D, at the same time, we also inserted n new edges and n new nodes into G_D, where both m and n increase from 100 to 500 with a step of 100. Therefore, $\triangle G_D$ increases from 200 to 1,000 with a step of 200 in each experiment.

Comparison Methods: As discussed in *Section 2*, there is no existing GPNM method in the literature which takes the relationships of updates into consideration. Therefore, in the experiments, we implemented the following GPNM methods:

- **TopKDAG**: TopKDAG is the most promising static GPNM method proposed in [40], which does not take the updates of G_D into consideration. When facing any update in G_D, TopKDAG needs to recompute the GPNM results starting from scratch.
- **INC-GPNM**: INC-GPNM is the most promising incremental GPNM method proposed in [106], which takes the updates of G_D into consideration. INC-GPNM needs to perform an incremental GPNM procedure for each of the updates in G_D.
- **NEH-GPNM**: In order to investigate the performance of EH-Tree, we implemented the GPNM algorithm without EH-Tree, called NEH-GPNM.

[2]https://socnetv.org/

Implementation: All the three algorithms were implemented using GCC 4.8.2 running on a server with Intel Xeon-E5 2630 2.60GHz CPU, 256GB RAM, and Red Hat 4.8.2-16 operating system. For each dataset, we considered 5 sets of updates and 5 sets of pattern graphs, and the experiments were conducted on each dataset for 5 independent runs. Therefore, there are 125=5*5*5 results of the query processing time on each dataset.

4.4.2 Experimental Results and Analysis

Figs. 4.5 to 4.9 depict the average query processing time with the varying sizes of $\triangle G_D$ on different sizes of G_P. The results and analysis are as follows.

Results-1 (Efficiency): With the increase of the size of the datasets, the average processing time of EH-GPNM is always less than that of TopKDAG, INC-GPNM and NEH-GPNM in all the cases of experiments, and the average processing time of NEH-GPNM is always less than that of INC-GPNM and TopKDAG in all the cases of experiments. The detailed results are given in Table 4.11, and the comparisons between the methods are shown in Table 4.12. On average, (1) EH-GPNM can reduce the query processing time by *51.23%*, *22.59%* and *10.31%* compared with that of TopKDAG, INC-GPNM and NEH-GPNM respectively; and (2) Based on statistics, NEH-GPNM can reduce the query processing time by *45.69%* and *13.68%* compared with that of TopKDAG and INC-GPNM respectively. The improvement remains consistent when the size of datasets has significantly increased.

Analysis-1: As discussed in *Section 4.1*, if there exist elimination relationships among the updates, both NEH-GPNM and EH-GPNM require less execution time than INC-GPNM and TopKDAG as they can avoid performing an incremental GPNM procedure for each of the updates. Compared with NEH-GPNM, EH-GPNM has better efficiency

Figure 4.5: The average query processing time in Ask Ubuntu on EH-GPNM

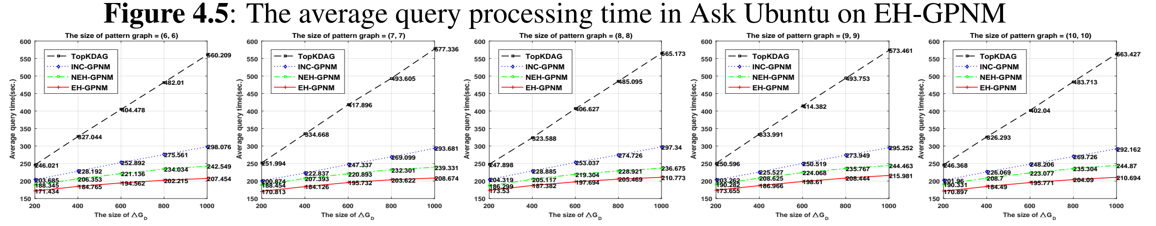


Figure 4.6: The average query processing time in Facebook on EH-GPNM

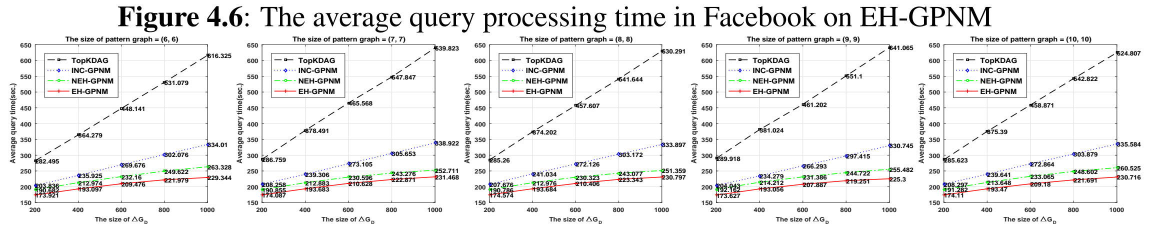


Figure 4.7: The average query processing time in Super User on EH-GPNM

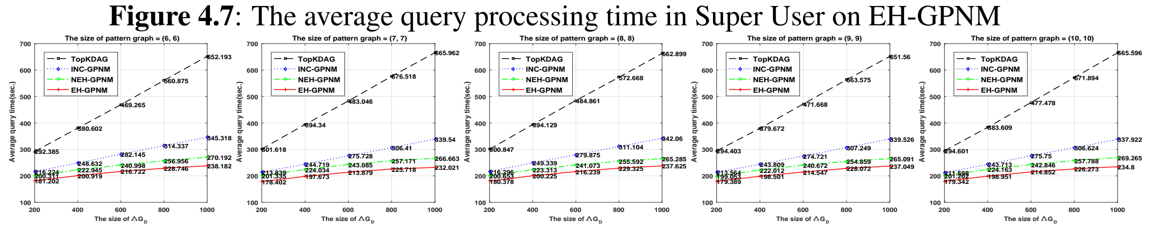


Figure 4.8: The average query processing time in Wiki Talk on EH-GPNM

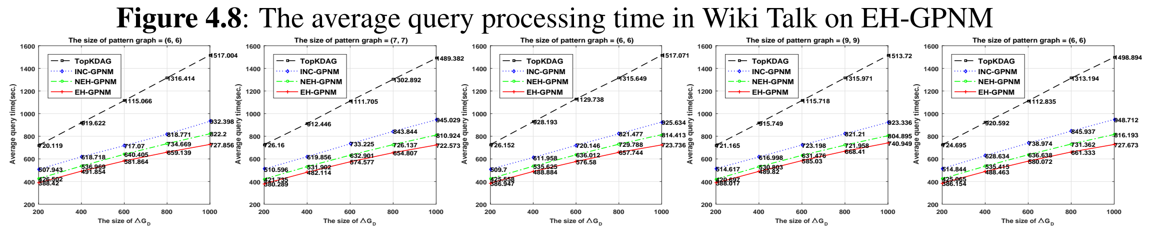


Figure 4.9: The average query processing time in LiveJournal on EH-GPNM

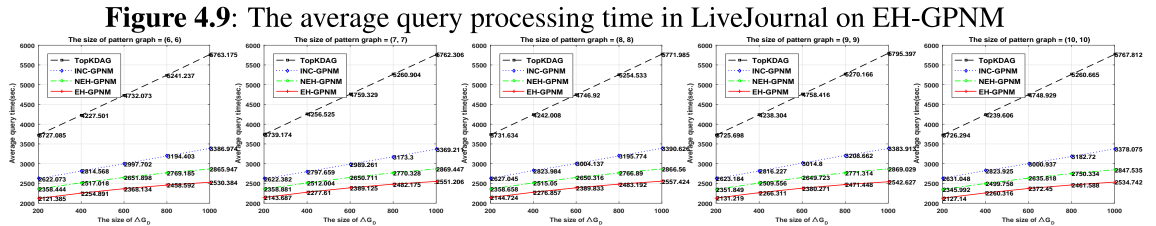


Table 4.11: The average query processing time based on different datasets on EH-GPNM

Dataset	EH-GPNM	NEH-GPNM	INC-GPNM	TopKDAG
Ask Ubuntu	193.91s	218.50s	249.48s	408.47s
Facebook	205.67s	227.71s	270.47s	458.46s
Super User	211.56s	237.86s	277.60s	477.85s
Wiki Talk	568.53s	627.21s	724.51s	1160.01s
LiveJournal	2359.09s	2628.49s	3002.90s	4749.91s
Average	707.59s	787.95s	904.99s	1450.94s

Table 4.12: Comparison with TopKDAG, INC-GPNM and NEH-GPNM based on different datasets on EH-GPNM

Dataset	with TopKDAG	with INC-GPNM	with NEH-GPNM
Ask Ubuntu	**52.53% less**	**22.27% less**	**11.25% less**
Facebook	**55.14% less**	**23.96% less**	**9.68% less**
Super User	**55.73% less**	**23.79% less**	**11.05% less**
Wiki Talk	**52.99% less**	**21.52% less**	**9.35% less**
LiveJournal	**50.33% less**	**21.43% less**	**10.24% less**
Average	**51.23% less**	**22.59% less**	**10.31% less**

as it can avoid checking each pair of the updates by searching the EH-Tree.

Results-2 (Scalability): With the increase of the scale of $\triangle G_D$ from 200 to 1,000, the processing time of both INC-GPNM and TopKDAG increases fast while the processing time of both NEH-GPNM and EH-GPNM increase slowly compared with that of INC-GPNM and TopKDAG, which shows the better scalability of NEH-GPNM and EH-GPNM. Moreover, EH-GPNM has the best scalability among all the four algorithms. The detailed results are given in Table 4.13, and the comparisons between the methods are shown in Table 4.14.

Analysis-2: With the increase of the scale of $\triangle G_D$, since TopKDAG needs to recompute the results starting from scratch for each update and INC-GPNM needs to perform an incremental GPNM procedure for each update to find the matching nodes, the scale of $\triangle G_D$ have a significant influence on their query processing time. While NEH-GPNM and EH-GPNM consider the elimination relationships between $\triangle G_D$,

Table 4.13: The average query processing time based on different scales of $\triangle G_D$ on EH-GPNM

Scale of $\triangle G_D$	EH-GPNM	NEH-GPNM	INC-GPNM	TopKDAG
200	609.09s	671.82s	752.05s	1056.99s
400	666.72s	737.74s	828.98s	1250.07s
600	716.16s	795.62s	905.35s	1442.15s
800	757.18s	845.99s	981.08s	1633.99s
1000	789.60s	888.59s	1057.52s	1827.47s

Table 4.14: Comparison with TopKDAG, INC-GPNM and NEH-GPNM based on different scales of $\triangle G_D$ on EH-GPNM

Scale of $\triangle G_D$	with TopKDAG	with INC-GPNM	with NEH-GPNM
200	**42.38% less**	**19.01% less**	**9.34% less**
400	**46.67% less**	**19.57% less**	**9.63% less**
600	**50.34% less**	**20.90% less**	**9.99% less**
800	**53.66% less**	**22.82% less**	**10.50% less**
1000	**56.79% less**	**25.33% less**	**11.14% less**

the query processing time of NEH-GPNM and EH-GPNM increase slowly compared with that of INC-GPNM and TopKDAG. Because of the EH-Tree index, EH-GPNM can efficiently find the elimination relationships for each pair of updates in $\triangle G_D$, which means that it has the best scalability among all the four algorithms.

Summary: The experimental results and analysis have demonstrated that the proposed EH-GPNM provides an effective means to answer GPNM queries with the updates of a data graph. In addition, we have also proposed a tree structure to index the elimination relationships between the updates, and with our proposed index, EH-GPNM can greatly save query processing time, which enables EH-GPNM to outperform NEH-GPNM in efficiency. Compared to TopKDAG, INC-GPNM and NEH-GPNM, EH-GPNM can reduce the query processing time by an average of *51.23%*, *22.59%* and *10.31%* respectively. In particular, when facing a large number of updates in a data graph, EH-GPNM has much better performance.

4.5 Conclusion

In this paper, we have proposed a GPNM method called EH-GPNM considering multiple updates in data graphs. EH-GPNM is the first work which considers the elimination relationships among the updates in data graphs, and thus, can efficiently deliver node matching results and reduce the query processing time. The experimental results on five real-world social graphs have demonstrated the efficiency of our proposed method.

Chapter 5

Incremental Graph Pattern based Node Matching Considering the Elimination Relationships Among Updates in Both Pattern Graphs and Data Graphs

As we introduced in Section 1.1.1, nodes and edges in both G_P and G_D are usually updated frequently over time. However, in a large-scale social graph that is updated with a high frequency, the algorithm INC-GPNM algorithm introduced in Chapter 3 is still computationally expensive because it ignores the relationships that exist among the updates in both G_P and G_D, and thus, when facing any update, it has to perform an incremental GPNM procedure for each of the updates. The EH-GPNM algorithm introduced in Chapter 4 considers the updates in G_D only. When facing updates in the pattern graph, it still has to perform the incremental GPNM procedure for each of the updates in the pattern graph. Therefore, a new efficient GPNM method is in demand.

Inspired by EH-GPNM, we realized that there exists elimination relationships among the updates. EH-GPNM only considers the single-graph elimination relationships among the updates in data graph. However, in real life, the elimination relationships exist among the updates not only in the data graph but also in the pattern graph,

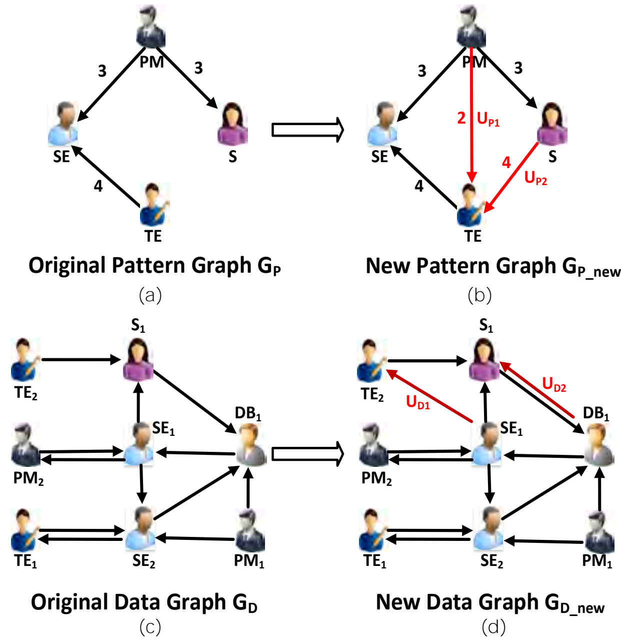


Figure 5.1: The elimination relationships among the updates in both G_P and G_D

and even exist among the updates from different graphs (one update from the pattern graph and the other one from the data graph). Below Example 5.1 illustrates the details of our motivations.

Table 5.1: The node matching results of Example 5.1

Nodes in G_P	Matching nodes in G_D
PM	PM_1
SE	SE_1, SE_2
S	S_1
TE	TE_1, TE_2

Example 5.1: Based on the pattern graph and data graph shown in Fig. 5.1(c) and Fig. 5.1(a) respectively, the original GPNM matching results are shown in Table 5.1. Suppose there are two updates in the pattern graph, where PM needs to be associated with a TE within two hops (denoted as U_{P1} in Fig. 5.1(b)), and an S needs to be associated with a TE within four hops (denoted as U_{P2} in Fig. 5.1(b)). And there are also two updates in data graph, where SE_1 establishes the collaboration relationship with TE_2 (denoted as U_{D1} in Fig. 5.1(d)) and DB_1 establishes the collaboration rela-

tionship with S_1 (denoted as U_{D2} in Fig. 5.1(d)). The new pattern graph G_{P_new} and new data graph G_{D_new} are shown in Fig. 5.1(b) and Fig. 5.1(d) respectively.

Based on these two updated graphs, the INC-GPNM [106] has to apply the incremental procedure four times because there are a total of four updates in G_P and G_D, leading to low efficiency. However, in practice, one update can be eliminated by another update. It is easy to understand that in each single graph, if one edge (node) is firstly removed from (or inserted into) G_D (G_P) and then inserted back to (or removed from) G_D (G_P), the effects of the two updates can eliminate each other. Therefore, there may exist elimination relationships among the updates in a single graph of G_P or G_D, and we term this kind of elimination relationships of a single graph as *single-graph elimination relationships*. More importantly, one update in a graph may eliminate an update in another graph, we term this kind of elimination relationships as *cross-graph elimination relationships*. In Example 5.1, although in update U_{P1}, a PM needs to be associated with a TE within 2 hops, it indeed leads to no change in the GPNM results. This is because in another update U_{D1}, SE_1 happens to establish the collaboration with TE_2, making all the PMs in the data graph be connected with a TE within 2 hops. Therefore, the effects of U_{P1} and U_{D1} eliminate each other.

This example motivates us to develop a new GPNM solution which considers the elimination relationships among the updates to efficiently answer GPNM queries. When facing an updated pattern graph and an updated data graph, we can compute the GPNM result for the original pattern graph, and then deliver the new GPNM result by analyzing the elimination relationships among the updates, instead of performing the incremental GPNM procedure for each of the updates.

This new GPNM solution is significant for the social graph searches in large-scale and frequently updated social networks, such as Facebook and Twitter.

In this new solution, there are three major challenges. Firstly, it is non-trivial to identify the elimination relationships among the updates because there exist both single-graph elimination relationships and cross-graph elimination relationships. Therefore, the first challenge of our work is **CH1**: *how to effectively detect the elimination*

relationships of the updates. Secondly, if update U_a eliminates update U_b, and update U_b eliminates update U_c, there exists a hierarchical structure of them, which applies to all the elimination relationships. As it is computationally expensive to deliver GP-NM results by investigating each of the elimination relationships among the updates, it is beneficial to build up an index to record the hierarchical structure of all the elimination relationships. Therefore, the second challenge of our work is **CH2**: *how to build up an index structure to record the hierarchical structure of all the elimination relationships covering both single-graph elimination relationships and cross-graph elimination relationships, which supports the development of an efficient algorithm to deliver the GPNM results by making use of the index.* Thirdly, in the GPNM procedure, we need to compute the shortest path length between any two nodes, which is very time-consuming. Therefore, the third challenge of our work is **CH3**: *how to efficiently compute the shortest path length between any two nodes to speed up the GPNM procedure.*

5.1 Elimination Relationships in Both Pattern Graphs and Data Graphs

In this section, we first analyze three types of elimination relationships. Then, we propose the effective methods to detect these elimination relationships. We further build up an index to record the hierarchical structure of these elimination relationships.

5.1.1 Elimination Relationship Types in Both Pattern Graphs and Data Graphs

The elimination relationships can be categorized into three types. Below we analyze the *elimination relationships* for these three types respectively.

Single-graph elimination relationships in G_P (Type I): For each update U_{Pi} in pat-

tern graph G_P, we need to identify the nodes in data graph G_D that has the possibility to be added into or removed from the matching results. We call these nodes as *candidate nodes* and put these candidate nodes into the set of candidate nodes (denoted as $Can_N(U_{Pi})$). Given two updates U_{Pi} and U_{Pj}, if the set of candidate nodes of an update U_{Pi} covers that of U_{Pj}, i.e., $Can_N(U_{Pi}) \supseteq Can_N(U_{Pj})$, we say U_{Pi} *eliminates* U_{Pj}, denoted as $U_{Pi} \sqsupseteq U_{Pj}$.

Remark: $Can_N(U_{Pi})$ can be divided into two subsets: a) $Can_AN(U_{Pi})$, which represents the set of candidate nodes that has the possibility to be *added into* the matching results; b) $Can_RN(U_{Pi})$, which represents the set of candidate nodes that has the possibility to be *removed from* the matching results.

Single-graph elimination relationships in G_D (Type II): In GPNM, we need to investigate if the shortest path length between each pair of nodes in G_D can satisfy the requirements of the bounded path length in G_P. For each update U_{Di} in data graph G_D, if the shortest path between two nodes has been affected by U_{Di}, we call these nodes as *affected nodes* and put these affected nodes into the set of affected nodes (denoted as $Aff_N(U_{Di})$). Given two updates U_{Di} and U_{Dj}, if the set of affected nodes of an update U_{Di} covers that of U_{Dj}, i.e., $Aff_N(U_{Dj}) \supseteq Aff_N(U_{Di})$, we say U_{Di} *eliminates* U_{Dj}, denoted as $U_{Di} \succeq U_{Dj}$.

Cross-graph elimination relationships between G_P and G_D (Type III): For an update U_{Pi} from a pattern graph G_P and an update U_{Di} from a data graph G_D, if these two updates keep the GPNM results unchanged, then U_{Pi} *and* U_{Di} *eliminate each other*, denoted as $U_{Di} \Leftrightarrow U_{Pi}$.

Algorithm 10: DER-I

Input: G_P, G_D, $\triangle G_P$, $SLen$
Output: The type I elimination relationships of the updates

```
1  for each pair of updates U_Pa and U_Pb ∈ △G_P do
2      if U_Pa and U_Pb ∈ △G_P^- then
3          for each pair of nodes u_i and v_i in IQuery do
4              if SLen(u_i, v_i) > the bounded path length on U_Pa then
5                  Put u_i, v_i into Can_RN(U_Pa);
6              if SLen(u_i, v_i) > the bounded path length on U_Pb then
7                  Put u_i, v_i into Can_RN(U_Pb);
8          if Can_RN(U_Pa) ⊇ Can_RN(U_Pb) then
9              U_Pa ⊒ U_Pb;
10     if U_Pa and U_Pb ∈ △G_P^+ then
11         for each pair of nodes u_i and v_i in G_D do
12             if SLen(u_i, v_i) < the bounded path length on U_Pa then
13                 Put u_i, v_i into Can_AN(U_Pa);
14             if SLen(u_i, v_i) < the bounded path length on U_Pb then
15                 Put u_i, v_i into Can_AN(U_Pb);
16         if Can_AN(U_Pa) ⊇ Can_AN(U_Pb) then
17             U_Pa ⊒ U_Pb;
18 Return type I elimination relationships of the updates;
```

5.1.2 Detecting Elimination Relationships in Both Pattern Graphs and Data Graphs

Below we introduce the detailed steps for detecting the three types of elimination relationships respectively.

Detect Type I elimination relationships (DER-I): For each update in the pattern graph, we first identify the nodes that have the possibility to be removed from or added into the original matching results. Then if the set of candidate nodes of an update U_{Pi} covers that of U_{Pj}, then U_{Pi} eliminates U_{Pj}. Below are the detailed steps of detecting Type I elimination relationships. The pseudo-code is shown in *Algorithm 10*.

- **Step 1:** We first build up the shortest path length matrix, $SLen$, to record the

shortest path length between each pair of nodes in G_D;

- **Step 2:** For each U_{Pi}, if $U_{Pi} \in \triangle G_P^+$, we then inspect if the shortest path length between the pair of nodes in $IQuery$ can satisfy the bounded path length constrain on U_{Pi}, if not, we put these nodes into $Can_RN(U_{Pi})$ as they cannot satisfy the bounded path length constrain of the newly added edge and have to be removed from $IQuery$; If $U_{Pi} \in \triangle G_D^-$, we then inspect if the shortest path length between the pair of nodes in G_D can satisfy the bounded path length constrain on U_{Pi}, if not, we put these nodes into $Can_AN(U_{Pi})$ as the edge with the shortest path length constrain they cannot satisfy has been deleted and these nodes can be added into $IQuery$;

- **Step 3:** For each pair of U_{Pa} and $U_{Pb} \in \triangle G_P$, if $Can_N(U_{Pa}) \supseteq Can_N(U_{Pb})$, then $U_{Pa} \sqsupseteq U_{Pb}$.

Example 5.2: Recall G_P and G_D shown in Fig. 5.1(c) and Fig. 5.1(a) respectively, $IQuery$ is shown in Table 5.1. U_{P1} is to insert edge $e(PM, TE)$ with a bounded path length 2 into G_P and U_{P2} is to insert edge $e(S, TE)$ with a bounded path length 4 into G_P shown in Fig. 5.1(b). The *SLen* of G_D in Fig. 5.1(c) is shown in Table 5.2. With U_{P1}, because the PM needs to be associated with TE within 2 steps and the shortest path length between PM_2 and TE_2 is ∞, which is larger than the bounded path length 2, then PM_2 and TE_2 are added into $Can_RN(U_{P1})$. After PM_2 and TE_2 are set as candidate nodes, we need to check if the nodes connected to PM_2 and TE_2 can be set as candidate nodes. Because the shortest path length between PM_1 and S_1, the shortest path length between PM_1 and SE_1, SE_2, and the shortest path length betweenSE_2, SE_1 and TE_1 are all less than the corresponding bounded path length in G_P, then only PM_2 and TE_2 are added into $Can_RN(U_{P1})$; With U_{P2}, only TE_2 is added into $Can_RN(U_{P2})$. The set of candidate nodes of U_{P1} and U_{P2} are shown in Table 5.3. Because $Can_RN(U_{P1}) \supseteq Can_RN(U_{P2})$, then $U_{P1} \sqsupseteq U_{P2}$.

Table 5.2: $SLen$ of G_D in Fig. 5.1(c).

	PM_1	PM_2	SE_1	SE_2	S_1	TE_1	TE_2	DB_1
PM_1	0	3	2	1	3	2	∞	1
PM_2	∞	0	1	2	2	3	∞	3
SE_1	∞	1	0	1	1	2	∞	2
SE_2	∞	3	2	0	3	1	∞	1
S_1	∞	3	2	3	0	4	∞	1
TE_1	∞	4	3	1	4	0	∞	2
TE_2	∞	4	3	4	1	5	0	2
DB_1	∞	2	1	2	2	3	∞	0

Table 5.3: The set of candidate nodes of U_{Pi}

Updates in pattern graph	$Can_RN(U_{Pi})$
U_{P1}	PM_2, TE_2
U_{P2}	TE_2

Theorem 5.1: *The order of the updates in $\triangle G_P$ does not affect the correctness of the detection of Type I elimination relationships.*

The Proof of Theorem 5.1: When U_{Pa} is applied to G_P prior to U_{Pb}, suppose $U_{Pa} \sqsupseteq U_{Pb}$. Then, according to the definition of an elimination relationship of *Type I*, $Can_N(U_{Pa}) \sqsupseteq Can_N(U_{Pb})$, namely, for any node $n_i \in Can_N(U_{Pb})$, n_i is also in $Can_N(U_{Pa})$. When U_{Pb} is applied to G_D prior to U_{Pa}, suppose U_{Pa} and U_{Pb} do not have the elimination relationship. Then, there is at least one node n_i such that $n_i \in Can_N(U_{Pb})$ and $n_i \notin Can_N(U_{Pa})$. However, this contradicts $n_i \in Can_N(U_{Pa})$ when U_{Pa} is applied to G_D. Therefore, *Theorem 1* is proven. $\square$

Detect Type II elimination relationships (DER-II): For each update in the data graph, we first detect the nodes where the shortest path length in data graph between them are changed by each update (denoted as *affected nodes*). Then, if the set of affected nodes of an update U_{Di} covers that of U_{Pj}, then U_{Pi} eliminates U_{Pj}. Below are the detailed steps of detecting Type II elimination relationships. The pseudo-code is shown in *Algorithm 11*.

Algorithm 11: DER-II

Input: G_P, G_D, $\triangle G_D$, $SLen$
Output: The type II elimination relationships of the updates

1. **for** *each pair of updates* U_{Da} *and* $U_{Db} \in \triangle G_D$ **do**
2.   **if** *the shortest path lengths between the nodes are not affected* **then**
3.   Keep the shortest path lengths in $SLen_new$ as that in $SLen$;
4.   **else**
5.   Apply the Dijkstra's algorithm for updating the shortest path lengths between the affected nodes in $SLen_{new}$;
6.   Put the affected nodes into $Aff_N(U_{Da})$;
7.   Put the affected nodes into $Aff_N(U_{Db})$;
8.   **if** $Aff_N(U_{Da}) \supseteq Aff_N(U_{Db})$ **then**
9. $U_{Da} \succeq U_{Db}$;
10. Return type II elimination relationships of the updates;

- **Step 1:** We first update $SLen$ to get the updated shortest path length matrix, $SLen_{new}$, for each update in data graph;

- **Step 2:** For each update U_{Di}, we compare the $SLen_{new}$ with $SLen$, if the shortest path length of two nodes is changed due to U_{Di}, we put these nodes into $Aff_N(U_{Di})$;

- **Step 3:** For each pair of updates U_{Da} and U_{Db}, if $Aff_N(U_{Da}) \supseteq Aff_N(U_{Db})$, then $U_{Da} \succeq U_{Db}$.

Table 5.4: $SLen_{new}$ with U_{D1}.

	PM_1	PM_2	SE_1	SE_2	S_1	TE_1	TE_2	DB_1
PM_1	0	3	2	1	3	2	3	1
PM_2	∞	0	1	2	2	3	2	3
SE_1	∞	1	0	1	1	2	1	2
SE_2	∞	3	2	0	3	1	3	1
S_1	∞	3	2	3	0	4	3	1
TE_1	∞	4	3	1	4	0	4	2
TE_2	∞	4	3	4	1	5	0	2
DB_1	∞	2	1	2	2	3	2	0

Example 5.3: Recall G_P and G_D shown in Fig. 5.1(c) and Fig. 5.1(a) respectively, $IQuery$ is shown in Table 5.1. U_{D1} is to insert edge $e(SE_1, TE_2)$ into G_D and U_{D2}

Table 5.5: $SLen_{new}$ with U_{D2}.

	PM_1	PM_2	SE_1	SE_2	S_1	TE_1	TE_2	DB_1
PM_1	0	3	2	1	2	2	∞	1
PM_2	∞	0	1	2	2	3	∞	3
SE_1	∞	1	0	1	1	2	∞	2
SE_2	∞	3	2	0	2	1	∞	1
S_1	∞	3	2	3	0	4	∞	1
TE_1	∞	4	3	1	3	0	∞	2
TE_2	∞	4	3	4	1	5	0	2
DB_1	∞	2	1	2	1	3	∞	0

is to insert edge $e(DB_1, S_1)$ into G_D shown in Fig. 5.1(d). The *SLen* of G_D in Fig. 5.1(c) is shown in Table 5.2. the $SLen_{new}$ of U_{D1} and U_{D2} in Fig. 5.1(d) are shown in Table 5.4 and Table 5.5 respectively. With U_{D1}, because the shortest path lengths from all the other nodes to TE_1 are changed, then all the nodes in data graph are set as the affected nodes of U_{D1}. With U_{D2}, because the shortest path lengths from PM_1, SE_2, TE_1 and DB_1 to S_1 are are changed, then PM_1, SE_2, TE_1, DB_1 and S_1 are set as affected nodes. The set of affected nodes of U_{D1} and U_{D2} are shown in Table 5.6. Because $\mathit{Aff_N}(U_{D1}) \supseteq \mathit{Aff_N}(U_{D2})$, then $U_{D1} \succeq U_{D2}$.

Table 5.6: The affected nodes of U_{D1} and U_{D2}

Updates in data graph	The *affected nodes*
U_{D1}	PM_1, PM_2, SE_1, SE_2, S_1, TE_1, TE_2, DB_1
U_{D2}	PM_1, SE_2, S_1, TE_1, DB_1

Theorem 5.2: *The order of the updates in $\triangle G_D$ does not affect the correctness of the detection of Type II elimination relationship.*

The Proof of Theorem 5.2: When U_{Da} is applied to G_D prior to U_{Db}, suppose $U_{Da} \succeq U_{Db}$. Then, according to the definition of the elimination relationships of *Type II*, $\mathit{Aff_N}(U_{Da}) \supseteq \mathit{Aff_N}(U_{Db})$, namely, for any node $n_i \in \mathit{Aff_N}(U_{Db})$, n_i is also in $\mathit{Aff_N}(U_{Da})$. When U_{Db} is applied to G_D prior to U_{Da}, suppose U_{Da} and U_{Db} do not have the elimination relationship. Then, there is at least one node

Algorithm 12: DER-III

Input: G_P, G_D, $\triangle G_P$, $\triangle G_D$, $SLen$, $SLen_{new}$
Output: The type III elimination relationships of the updates

```
1 for each update U_Pi ∈ △G_P do
2     Perform DER-I to get Can_N(U_Pi);
3 for each update U_Di ∈ △G_D do
4     Perform DER-II to get Aff_N(U_Di);
5 for each pair of nodes u_i, v_i in U_Pi do
6     if SLen_new(u_i, v_i) ≤ the bounded path length on U_Pi then
7         U_Di ⇔ U_Pi;
8 Return type III elimination relationships of the updates;
```

n_i such that $n_i \in Aff_N(U_{Db})$ and $n_i \notin Aff_N(U_{Da})$. However, this contradicts $n_i \in Aff_N(U_{Da})$ when U_{Da} is applied to G_D. Therefore, *Theorem 2* is proven. □

Detect Type III elimination relationships (DER-III): For an update U_{Pi} from a pattern graph and an update U_{Di} from a data graph, we need to inspect if these two updates keep the GPNM results unchanged. Below are the detailed steps of detecting Type III elimination relationships. The pseudo-code is shown in *Algorithm 12*.

- **Step 1:** For the update U_{Pi} from a pattern graph, we identify the candidate nodes for U_{Pi}.

- **Step 2:** For the update U_{Di} from a data graph, we identify affected nodes for U_{Di}.

- **Step 3:** Based on $Can_N(U_{Pi})$ and $Aff_N(U_{Di})$, if $Aff_N(U_{Di}) \supseteq Can_N(U_{Pi})$, which means the shortest path length between any nodes in the set of candidate nodes is changed due to the update U_{Di}, we inspect the updated shortest path length matrix $SLen_{new}$ to check if the shortest path length of the candidate nodes can satisfy the new pattern graph. If so, no node should be added into or deleted from the matching results; therefore, $U_{Pi} \Leftrightarrow U_{Di}$.

Example 5.4: Recall G_P and G_D shown in Fig. 5.1(c) and Fig. 5.1(a) respectively, $IQuery$ is shown in Table 5.1. U_{P1} is to insert edge $e(PM, TE)$ with a bounded

path length 2 into G_P shown in Fig. 5.1(b) and U_{D1} is to insert edge $e(SE_1, TE_2)$ into G_D shown in Fig. 5.1(d). Based on Example 5.2 and Example 5.3, we have $Can_N(U_{Pi})$={PM_2, TE_2} and $Aff_N(U_{Di})$={PM_1, PM_2, SE_1, SE_2, S_1, TE_1, TE_2, DB_1}, then $Aff_N(U_{D1}) \supseteq Can_N(U_{P1})$. Since $AFF(PM_2, TE_2)$ = $(\infty, 2)$, the shortest path length between PM_2 and TE_2 can still satisfy the bounded path length on the newly inserted edge. Therefore, $U_{P1} \Leftrightarrow U_{D1}$.

Complexity: The complexity of the generation and the updates of $SLen$ is $\mathcal{O}(|N_D|(|N_D|+|E_D|)$ [94]. In the worst case, *DER-I*, *DER-II* and *DER-III* need to check $SLen_{new}$ for each update, then the complexity of each of *DER-I*, *DER-II* and *DER-III* is $\mathcal{O}(|N_D|(|N_D|+|E_D|)+|\triangle G||N_D|^2)$, where $|N_D|$ and $|E_D|$ are the number of the nodes and the number of the edges respectively in G_D, and $|\triangle G|$ is the scale of the updates.

5.1.3 EH-Tree for the Updates in Both Pattern Graphs and Data Graphs

EH-Tree for the Updates in Both Pattern Graphs and Data Graphs Establishment: As it is computationally expensive to deliver GPNM results by investigating each of the elimination relationships among the updates, we build up an index to record the hierarchical structure of the elimination relationships. This index structure can efficiently help detect the elimination relationships between each pair of updates. We present the details of the generation of EH-Tree as follows. The pseudo-code is shown in *Algorithm 13*.

- **Step 1:** Firstly, for each update, we use the method mentioned in Section 5.1 to identify the affected nodes for each update in data graph and identify the candidate nodes for each update in pattern graph. Each tree node in EH-Tree denotes an update and stores the affected nodes or candidates of the update.

- **Step 2:** Based on the affected nodes and candidate nodes of each update, we have the following strategies: (a) the update that has the maximum number of

affected nodes or candidate nodes is set as the root of an EH-Tree; (b) if the affected nodes of one update U_{Di} can be covered by another update U_{Dj}, then U_{Di} is set as a child tree node of U_{Dj}; (c) if the candidate nodes of one update U_{Pi} can be covered by another update U_{Pj}, then U_{Pi} is set as a child tree node of U_{Pj}; (d) if U_{Di} and U_{Pj} can eliminate each other, then we can set the U_{Pi} as a child tree node of U_{Di} or set the U_{Di} as a child tree node of U_{Pi}.

- **Step 3:** We then recursively insert all the updates into the EH-Tree.

Example 5.5: Recall U_{D1}, U_{D2}, U_{P1} and U_{P2} in Fig .5.1. As U_{D1} has the maximum number of affected nodes in all the updates, it is set as the root of EH-Tree; with U_{D2}, as the set of affected nodes of U_{D1} covers that of U_{D2}, U_{D2} is set as the child node of U_{D1}; with U_{P1}, as the set of candidate nodes of U_{P1} covers that of U_{P2}, U_{P2} is set as child node of U_{P1}; Because U_{D1} and U_{P1} can eliminate each other, U_{P1} is set as the child node of U_{D1}. The completed EH-Tree is shown in Fig. 5.2.

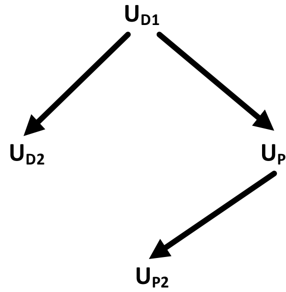


Figure 5.2: The EH-Tree of Example 5.5

5.2 Graph Partition

5.2.1 Label-based Partition

It is computational expensive to construct the shortest path length matrix $SLen$ and update the $SLen_{new}$. Therefore, in this section, we propose a graph partition method

---

Algorithm 13: EH-Tree for the Updates in Both Pattern Graphs and Data Graphs Establishment

---

```
    Input: G_P, G_D, △G_D, △G_P, SLen
    Output: the address of the root of an EH-Tree
 1  for each update U_Di ∈ △G_D do
 2      Detect affected nodes for U_Di;
 3      Put the affected nodes into Aff_N(U_i);
 4      if U_Di has the maximum number of affected nodes then
 5          U_Di is set as the root of an EH-Tree;
 6      else
 7          for each update U_Dj ∈ △G_D do
 8              if the set affected nodes of U_Di covers that of U_Dj then
 9                  U_Dj is set as the child node of U_Di;
10                  if the number of affected nodes of U_Dj is less than or equal to
                    that by other child nodes then
11                      U_Dj is set as the left child node;
12                  else
13                      U_Dj is set as the right child node;
14  for each update U_Pi ∈ △G_P do
15      Detect candidate nodes for U_Pi;
16      Put the candidate nodes into Can_N(U_i);
17      if U_Pi has the maximum number of affected nodes then
18          U_Pi is set as the root of an EH-Tree;
19      else
20          for each update U_Pj ∈ △G_P do
21              if the set candidate nodes of U_Pi covers that of U_Pj then
22                  U_Pj is set as the child node of U_Pi;
23                  if the number of candidate nodes of U_Pj is less than or equal to
                    that by other child nodes then
24                      U_Pj is set as the left child node;
25                  else
26                      U_Pj is set as the right child node;
27  for each pair of update U_Pi and U_Di do
28      if U_Pi and U_Di eliminate each other then
29          U_Pi is set as the child node of U_Di or U_Di is set as the child node of U_Pi;
30  return the address of the root of an EH-Tree;
```

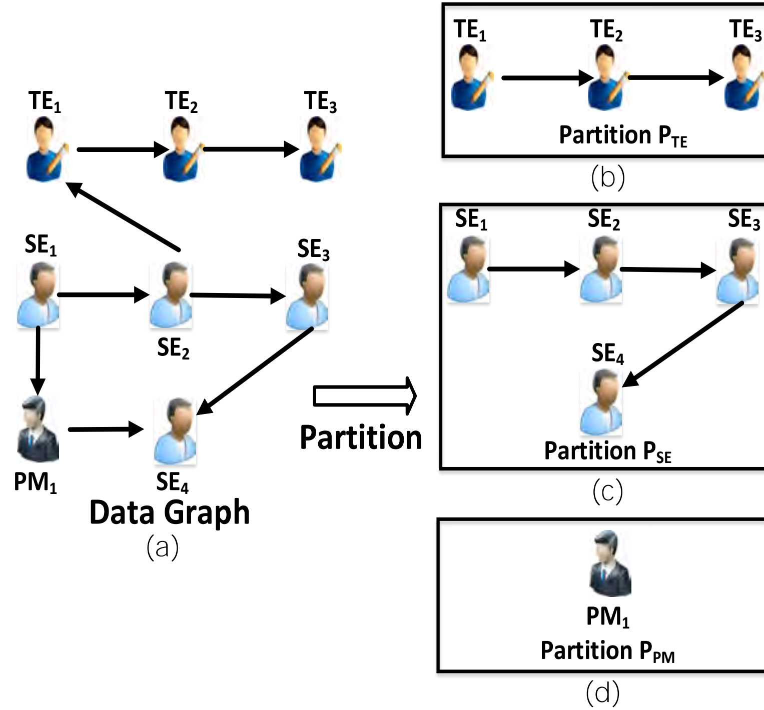


Figure 5.3: Label-based Partition

to improve the efficiency of computing the shortest path length between any two n-odes. Based on the observation that people with the same role (e.g., has the same job title) usually connect with each other closely [16], we put the nodes that have the same label in a data graph and their corresponding edges into the same partition. Then the shortest path computation will be processed distributively based on the partitions.

Example 5.6: Fig. 5.3(a) depicts a data graph, where it has three different labels of nodes, namely, TE, SE and PM respectively. Based on the different labels of the nodes, we divide the data graph into three partitions, denoted as partition P_{TE}, P_{SE} and P_{PM} respectively.

After the partition, we need to preserve the connectivity of the data graph. Then our partition method records the cross-partition edges in the partitions where the starting nodes are in. For example, in Fig. 5.3(a), we record $e(SE_2, TE_1)$ in the partition P_{SE}. Before introducing the process of computing shortest path length, we first define some nodes with properties below.

Definition 1. inner bridge node: Given a partition P_i, a node v_i ($v_i \in P_i$) is termed as an *inner bridge node* of P_i if there is an edge $e(v_i, v_j)$ in data graph and $v_j \notin P_i$. Let $IB(P_i)$ denote the set of the inner bridge nodes of partition P_i.

Example 5.7: In Fig. 5.3, SE_2 is an inner bridge node of P_{SE}, because there exists an edge $e(SE_2, TE_1)$ in Fig. 5.3(a) and $TE_1 \notin P_{SE}$.

Definition 2. outer bridge node: Given a partition P_i, a node v_j ($v_j \notin P_i$) is termed as a outer bridge node of P_i if there exists an edge $e(v_i, v_j)$ in data graph and $v_i \in P_i$. Let $OB(P_i)$ denote the set of the bridges nodes of partition P_i.

Example 5.8: In Fig. 5.3, PM_1 is a outer bridge node of P_{SE} because there exists an edge $e(SE_2, PM_1)$ in Fig. 5.3(a).

We use a table to record the *inner bridge nodes* and *outer bridge nodes* in each partition. For example, the *inner bridge nodes* and *outer bridge nodes* for partition P_{SE} are shown in Table 5.7.

Table 5.7: The inner bridge nodes and outer bridge nodes of T_{SE}

inner bridge nodes	outer bridge nodes
SE_1	PM_1
SE_2	TE_1

5.2.2 Graph Partition based Shortest Path Length Computation

We divide computation of the shortest path length into two sub-processes, i.e., **sub-process-1**: computing the shortest path length between any two nodes in the same partition, and **sub-process-2**: computing the shortest path length between any two nodes in different partitions. Below we introduce these two sub-processes in detail.

sub-process-1: For each partition P_i, if $OB(P_i) = \emptyset$, we apply the Dijkstra's algorithm in this partition to compute the shortest path length. Otherwise, we apply the following steps. The pseudo-code is shown in *Algorithm 14*.

- **Step 1:** In each partition P_i, we denote the nodes in P_i as v_{Pi}, for each pair of n-odes from v^a_{Pi} to v^b_{Pi}, we first apply the Dijkstra's algorithm to compute the shortest path length value from v^a_{Pi} to v^b_{Pi} in this partition (denoted as $SP_{Pi}(v^a_{Pi}, v^b_{Pi})$) and set the shortest path length value (denotes as $SP_D(v^a_{Pi}, v^b_{Pi})$) from v^a_{Pi} to v^b_{Pi} in the data graph as $SP_{Pi}(v^a_{Pi}, v^b_{Pi})$;
- **Step 2:** For each outer bridge node v^c_{Pj} ($v^c_{Pj} \in OB(P_i)$), if $BN(p_j) = \emptyset$, then the shortest path length between v^a_{Pi} and v^b_{Pi} is still $SP_D(v^a_{Pi}, v^b_{Pi})$. Otherwise, if one of the outer bridge nodes of P_j belongs to partition P_i, we combine the partitions of P_i and P_j, and then apply the Dijkstra's algorithm to compute the shortest path length between v^a_{Pi} and v^b_{Pi} in the combined partition. If the new shortest path length is less than $SP_D(v^a_{Pi}, v^b_{Pi})$, we update $SP_D(v^a_{Pi}, v^b_{Pi})$ with the newly computed shortest path length;
- **Step 3:** We recursively apply *Step 2* to update $SP_D(v^a_{Pi}, v^b_{Pi})$ until no partition can be combined with P_i.

Example 5.9: To compute the shortest path length between any two nodes in P_{SE} in Fig. 5.3, because there are two outer bridge nodes in P_{SE}, i.e., PM_1 and TE_1, and P_{TE} has no outer bridge node and the outer bridge node of P_{PM} belongs to P_{SE}, we combine P_{SE} and P_{PM}. Then, we apply the Dijkstra's algorithm to compute the shortest path length in the combined partition. The shortest path length matrix of P_{SE} is shown in Table 5.8.

sub-process-2: For each partition P_i, if $OB(P_i) = \emptyset$, the shortest path length from any node in P_i to any node in other partitions is infinity. Otherwise, we apply the following steps. The pseudo-code is shown in *Algorithm 15*.

- **Step 1:** We first apply *sub-process-1* to compute the shortest path length between the nodes in same partition;
- **Step 2:** For each inner bridge node v^a_{Pi} in P_i with the outer bridge node v^a_{Pj}, we first set $SP_D(v^a_{Pi}, v^a_{Pj}) = 1$;

Algorithm 14: sub-process-1

Input: G_D, partitions of G_D
Output: The shortest path length between any two nodes in the same partition

```
1  for each partition P_i do
2      if OB(P_i) is ∅ then
3          for each pair of nodes from v^a_{Pi} to v^b_{Pi} in P_i do
4              Apply the Dijkstra's algorithm to compute SP_D(v^a_{Pi}, v^b_{Pi});
5      else
6          for each pair of nodes from v^a_{Pi} to v^b_{Pi} in P_i do
7              Apply the Dijkstra's algorithm to compute SP_{Pi}(v^a_{Pi}, v^b_{Pi});
8              Set SP_D(v^a_{Pi}, v^b_{Pi})=SP_{Pi}(v^a_{Pi}, v^b_{Pi});
9              for each outer bridge node in P_i that belongs to P_j do
10                 if BN(P_j) is ∅ then
11                     Return SP_D(v^a_{Pi}, v^b_{Pi});
12                 else
13                     if one of the outer bridge nodes in P_j belongs to P_i then
14                         Combine P_i and P_j;
15                         Apply the Dijkstra's algorithm to compute the shortest
                           path length from v^a_{Pi} to v^b_{Pi} in the combined partition;
16                         Update SP_D(v^a_{Pi}, v^b_{Pi});
17                     else
18                         Recursively inspect the outer bridge nodes of P_j until no
                           partition can be combined with P_i;
19                         Update SP_D(v^a_{Pi}, v^b_{Pi});
20 Return the shortest path length between any two nodes in the same partition;
```

- **Step 3:** For each node v^b_{Pj} in the same partition of v^a_{Pj}, we set $SP_D(v^a_{Pi}, v^b_{Pj}) = SP_D(v^a_{Pi}, v^a_{Pj}) + SP_D(v^a_{Pj}, v^b_{Pj})$; And for each node v^b_{Pi} in partition P_i, we set $SP_D(v^b_{Pi}, v^b_{Pj}) = SP_D(v^b_{Pi}, v^a_{Pi}) + SP_D(v^a_{Pi}, vv^b_{Pj})$.

Example 5.10: To compute the shortest path length from SE to TE in Fig. 5.3, because TE_1 is the outer bridge node of SE_2, then we set $SP_D(SE_2, TE_1) = 1$. For each node in the partition P_{TE}, $SP_D(SE_2, TE_2) = 1 + 1 = 2$ and $SP_D(SE_2, TE_3) = 1 + 2 = 3$. Since the shortest path length from SE_3 and SE_4 to SE_2 are infinity, the shortest path length from SE_3 and SE_4 to all the nodes in P_{TE} are all infinity. The shortest path length matrix between each node in P_{SE} to each node in P_{TE} is shown

Table 5.8: The shortest path length matrix of P_{SE}

	SE_1	SE_2	SE_3	SE_4
SE_1	0	1	2	2
SE_2	∞	0	1	2
SE_3	∞	∞	0	1
SE_4	∞	∞	∞	0

Algorithm 15: sub-process-2

Input: G_D, partition of G_D

Output: The shortest path length between any two nodes in different partitions

1 **for** *each partition* P_i **do**
2 **if** $OB(P_i)$ *is* $\emptyset$ **then**
3   The shortest path length from any node in P_i to any node in other partitions is infinity;
4 **else**
5   sub-process-1;
6   **for** *each inner bridge node* v^a_{Pi} *in* P_i *with the outer bridge node* v^a_{Pj} **do**
7   set $SP_D(v^a_{Pi}, v^a_{Pj}) = 1$;
8   **for** *each node* v^b_{Pj} *in the same partition of* v^a_{Pj} **do**
9   set $SP_D(v^a_{Pi}, v^b_{Pj}) = SP_D(v^a_{Pi}, v^a_{Pj}) + SP_D(v^a_{Pj}, v^b_{Pj})$;
10   **for** *each node* v^b_{Pi} *in partition* P_i **do**
11   set $SP_D(v^b_{Pi}, v^b_{Pj}) = SP_D(v^b_{Pi}, v^a_{Pi}) + SP_D(v^a_{Pi}, vv^b_{Pj})$;

12 Return the shortest path length between any two nodes in different partitions;

in Table 5.9.

Table 5.9: The shortest path length matrix from P_{SE} to P_{TE}

	TE_1	TE_2	TE_3
SE_1	2	3	4
SE_2	1	2	3
SE_3	∞	∞	∞
SE_4	∞	∞	∞

Complexity: In the worst case, we need to combine all the partitions to compute the shortest path length matrix. Therefore, the time complexity is $\mathcal{O}(|E_D|+|N_D|log|N_D|)$.

5.3 UA-GPNM Algorithm

In this section, we propose a new Updates-Aware GPNM algorithm, called UA-GPNM. It first searches the EH-Tree to efficiently detect both the single-graph elimination relationships and the cross-graph elimination relationships, and then incrementally delivers the GPNM results. The detailed steps of UA-GPNM are shown below. The pseudo-code is shown in *Algorithm 16*.

- **Step 1:** For each update $U_i \in \triangle G_D$ or $U_i \in \triangle G_P$, UA-GPNM first searches the EH-Tree to detect the elimination relationships among the updates.
- **Step 2:** UA-GPNM then recursively finds the elimination relationships for each update until all the updates have been investigated.
- **Step 3:** After searching the EH-Tree, we apply the incremental GPNM procedure for uneliminated updates to deliver the GPNM results. The details of the incremental GPNM procedure can be found in [106].

Complexity: Since UA-GPNM first searches the EH-Tree, and then incrementally deliver the GPNM results for the updates, UA-GPNM achieves $\mathcal{O}(|N_D|(|N_D|+|E_D|)+(|\triangle G| - |U_e|)(|N_D|^2) + |\triangle G| \log |\triangle G|)$ in time complexity, where $|U_e|$ is the number of the updates that can be eliminated.

5.4 Experiments on UA-GPNM

We now present the results and the analysis of experiments conducted on five real-world social graphs to evaluate the performance of our proposed UA-GPNM.

5.4.1 Experimental Setting

Datasets: We have used five real-world social graphs that are available at *snap.stanford.edu*. The details are shown in Table 5.10.

Algorithm 16: UA-GPNM

```
    Input: G_P, G_D, △G_P, △G_D, IQuery
    Output: SQuery
 1  Build up the EH-Tree for all the updates;
 2  for each U_Pi ∈ △G_P do
 3      Check the EH-Tree;
 4      if U_Pi is the parent node of U_Pj (i ≠ j) then
 5          U_Pi can eliminate U_Pj;
 6      else
 7          if U_Pi is the parent node of U_Di (i ≠ j) then
 8              U_Pi can eliminate U_Di;
 9  for each U_Di ∈ △G_D do
10      Check the EH-Tree;
11      if U_Di is the parent node of U_Dj (i ≠ j) then
12          U_Di can eliminate U_Dj;
13      else
14          if U_Di is the parent node of U_Pi (i ≠ j) then
15              U_Di can eliminate U_Pi;
16  Incrementally delivers the GPNM results for the updates;
17  return SQuery;
```

Pattern Graph Generation and Parameter Setting: We used a graph generator, *socnetv*[1], to generate pattern graphs, controlled by 3 parameters: (1) the number of nodes, (2) the number of edges, and (3) the bounded path length on each edge. Since the numbers of nodes and edges in a pattern graph are usually not large [39], they are set between 6 and 10. Since the bounded path length on each edge is usually a small integer [39], we randomly set the bounded path length on each edge from 1 to 3.

Updates of G_D: In each experiment, we removed m_G edges and m_G nodes from G_D; at the same time, we also inserted n_G new edges and n_G new nodes into G_D, where both m_G and n_G increase from 100 to 500 with a step of 100.

Updates of G_P: In each experiment, we removed m_P nodes and n_P edges from G_P, and add n_P new nodes and n_P new edges into G_P, where $1 \leq m_P \leq 5$, and $1 \leq n_P$

[1] https://socnetv.org/

Table 5.10: The sizes of datasets for UA-GPNM

Name	#Nodes	#Edges
email-EU-core	1,005	25,571
DBLP	317,080	1,049,866
Amazon	334,863	925,872
Youtube	1,134,890	2,987,624
LiveJournal	3,997,962	34,681,189

≤ 5.

Remark: In each experiment, let $\triangle G(\triangle G_P, \triangle G_D)$ denote the updates, where $\triangle G_P$ denotes the updates in G_P and $\triangle G_D$ denotes the updates in G_D.

Comparison Methods: As discussed in Chapter 2, there is no existing GPNM method in the literature which takes the relationships of updates in both pattern graphs and data graphs, and the partition strategy into consideration. Therefore, in the experiments, we implemented the following GPNM methods:

- **INC-GPNM**: INC-GPNM [106] takes the updates of G_D and G_P into consideration. INC-GPNM needs to perform an incremental GPNM procedure for each of the updates in G_D or G_P.

- **EH-GPNM**: EH-GPNM [105] only considers the elimination relationships in data graph, when facing any update in pattern graphs, it needs to perform an incremental GPNM procedure for each of the updates in G_P.

- **UA-GPNM-NoPar**: UA-GPNM-NoPar takes the relationships of updates in both pattern graph and data graph into consideration. However, it does not have the partition strategy.

Implementation: All the three algorithms were implemented using GCC 4.8.2 running on a server with Intel Xeon-E5 2630 2.60GHz CPU, 256GB RAM, and Red Hat 4.8.2-16 operating system. For each dataset, we considered 5 sets of updates and 5 sets of pattern graphs, and the experiments were conducted on each dataset for 5 independent runs. Therefore, there are a total of 125=5*5*5 results of the query processing

time on each dataset.

Figs. 5.4-5.8 depict the average query processing time with the varying sizes of $\triangle G$ on different sizes of G_P. The results and analysis are as follows.

5.4.2 Experimental Results and Analysis

Results-1 (Efficiency): With the increase of the size of the datasets, the average processing time of UA-GPNM is always less than that of INC-GPNM, EH-GPNM and UA-GPNM-NoPar in all the cases of experiments. The detailed results are given in Table 5.11, and the comparisons between the methods are shown in Table 5.12. On average, (1) UA-GPNM can reduce the query processing time by *58.60%*, *35.29%* and *17.70%* compared with that of INC-GPNM, EH-GPNM and UA-GPNM-NoPar respectively. The improvement remains consistent when the size of datasets has significantly increased.

Table 5.11: The average query processing time based on different datasets for UA-GPNM

Dataset	**UA-GPNM**	**UA-GPNM-NoPar**	**EH-GPNM**	**INC-GPNM**
email-EU-core	3.31s	3.98s	5.25s	8.27s
DBLP	210.34s	262.71s	322.38s	501.25s
Amazon	225.48s	278.37s	346.15s	536.85s
Youtube	497.70s	602.41s	753.03s	1185.23s
LiveJournal	1567.48s	1911.56s	2449.19s	3765.27s
Average	500.86s	611.70s	755.20s	1199.38s

Analysis-1: As we discussed in Section 5.1, if there exist elimination relationships among the updates, both UA-GPNM and UA-GPNM-NoPar require less execution time than INC-GPNM and EH-GPNM as they can avoid performing an incremental GPNM procedure for each of the updates. Compared with UA-GPNM-NoPar, UA-GPNM has better efficiency as it divides the data graphs into small subgraphs, saving the query processing time when applying the the Dijkstra's algorithms.

Figure 5.4: The average query processing time in email-EU-core on UA-GPNM

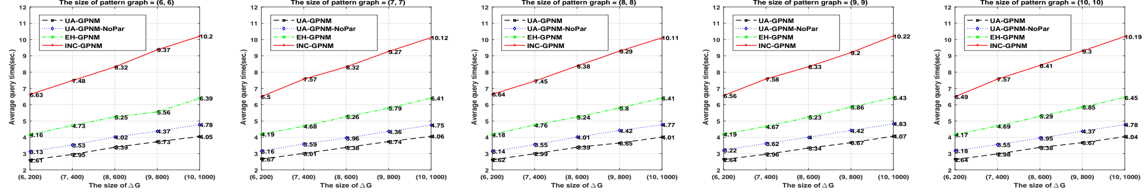


Figure 5.5: The average query processing time in DBLP on UA-GPNM

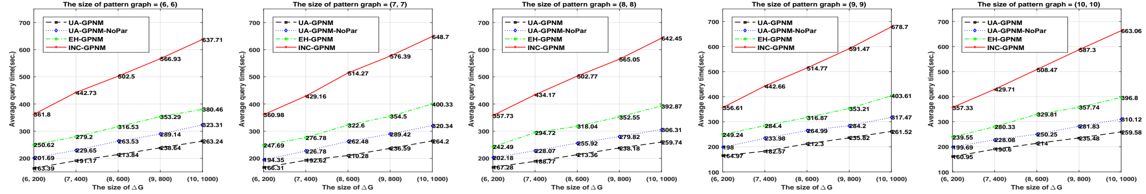


Figure 5.6: The average query processing time in Amazon on UA-GPNM

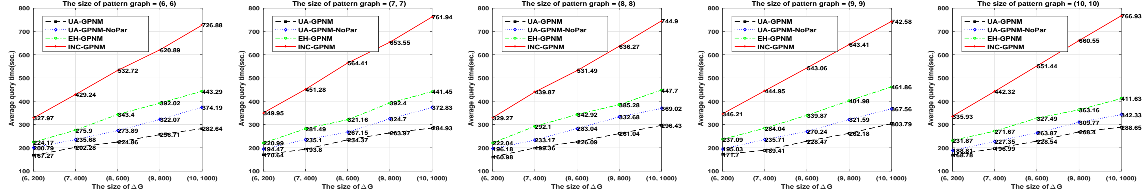


Figure 5.7: The average query processing time in Youtube on UA-GPNM

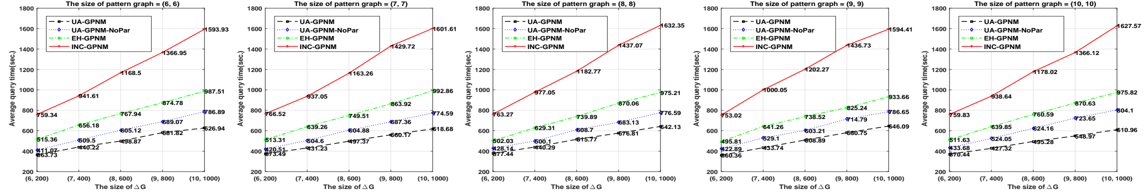


Figure 5.8: The average query processing time in LiveJournal on UA-GPNM

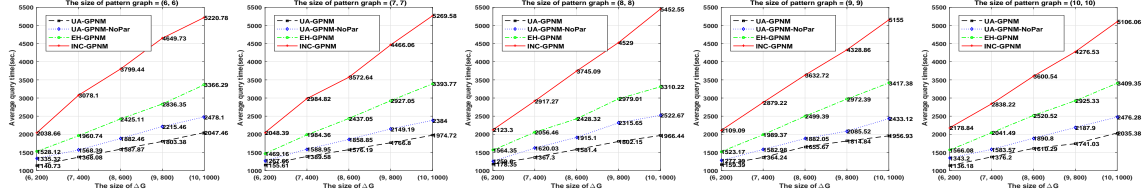


Table 5.12: Comparison with INC-GPNM, EH-GPNM and UA-GPNM-NoPar based on different datasets for UA-GPNM

Dataset	with INC-GPNM	with EH-GPNM	with UA-GPNM-NoPar
email-EU-core	**59.98% less**	**36.95% less**	**16.83% less**
DBLP	**58.04% less**	**34.75% less**	**19.77% less**
Amazon	**58.00% less**	**34.86% less**	**18.99% less**
Youtube	**58.60% less**	**33.91% less**	**14.91% less**
LiveJournal	**58.37% less**	**36.01% less**	**18.00% less**
Average	**58.60% less**	**35.29% less**	**17.70% less**

Results-2 (Scalability): With the increase of the scale of $\triangle G$ from (6, 200) to (10, 1000), the processing time of both INC-GPNM and EH-GPNM increases fast while the processing time of both UA-GPNM and UA-GPNM-NoPar increase slowly compared with that of INC-GPNM and EH-GPNM, which shows the better scalability of UA-GPNM and UA-GPNM-NoPar. Moreover, UA-GPNM has the best scalability among all the four algorithms. The detailed results are given in Table 5.13, and the comparisons between the methods are shown in Table 5.14.

Table 5.13: The average query processing time based on different scales of $\triangle G$ for UA-GPNM

Scale of $\triangle G$	UA-GPNM	UA-GPNM-NoPar	EH-GPNM	INC-GPNM
(6, 200)	371.64s	423.46s	503.03s	712.67s
(7, 400)	439.23s	513.71s	643.29s	956.63s
(8, 600)	510.02s	606.03s	774.87s	1182.12s
(9, 800)	571.69s	700.35s	907.19s	1417.40s
(10, 1000)	636.42s	786.02s	1038.96s	1625.27s

Table 5.14: Comparison with INC-GPNM, EH-GPNM and UA-GPNM-NoPar based on different scales of $\triangle G$ for UA-GPNM

Scale of $\triangle G$	with INC-GPNM	with EH-GPNM	with UA-GPNM-NoPar
(6, 200)	**47.85% less**	**26.12% less**	**12.24% less**
(7, 400)	**54.09% less**	**31.72% less**	**14.50% less**
(8, 600)	**56.86% less**	**34.18% less**	**15.84% less**
(9, 800)	**59.67% less**	**36.98% less**	**18.37% less**
(10, 1000)	**60.84% less**	**38.74% less**	**19.03% less**

Analysis-2: With the increase of the scale of $\triangle G$, since INC-GPNM needs to perform an incremental GPNM procedure for each update to find the matching nodes, the scale of $\triangle G$ have a significant influence on their query processing time. While UA-GPNM consider the elimination relationships among the updates, the query processing time of UA-GPNM increases slowly compared with that of INC-GPNM, EH-GPNM and UA-GPNM-NoPar, which means that it has the best scalability among all the four algorithms.

Summary: The experimental results have demonstrated that the proposed UA-GPNM provides an effective means to answer GPNM queries with the updates of a data graph and a pattern graph. In addition, we have also proposed a tree structure to index the elimination relationships between the updates, and with our proposed index and partition method, UA-GPNM can greatly save query processing time. Compared to INC-GPNM, EH-GPNM and UA-GPNM-NoPar, UA-GPNM can reduce the query processing time by an average of *58.60%*, *35.29%* and *17.70%* respectively. In particular, when facing a large number of updates in a data graph, UA-GPNM has much better performance.

5.5 Conclusion

In this paper, we have proposed a GPNM method called UA-GPNM considering multiple updates in both data graphs and pattern graphs. UA-GPNM can efficiently deliver node matching results, and can reduce the query processing time. To the best of our knowledge, UA-GPNM is the first GPNM method which takes the elimination relationships in both pattern graphs and data graphs into consideration. The experimental results on five real-world social graphs have demonstrated the efficiency of our proposed method and superiority over the state-of-art GPNM methods. In our future work, we will work on (1) the improvement of space complexity by designing new index structures, and (2) a new approach to selecting the top-k matching nodes.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

As a popular data model for representing the relationships of different data, graphs have been widely used in various fields such as social networks, social security and biology. Graph Pattern based Subgraph Matching (GPSM) is a fundamental problem in graph analysis. GPSM aims to find all the matching subgraphs of a pattern graph G_P in a data graph G_D. It has been increasingly used in knowledge discovery, traffic network analysis, intelligence analysis, and social networks analysis, among other applications. Conventional subgraph matching solutions are based on the *subgraph isomorphism* problem, in which matches are depend strictly on graph structure. However, the *subgraph isomorphism* problem is an NP-Complete problem [46], which makes it computationally expensive to find the exact matching subgraphs, especially in large graphs. To address this problem, *Bounded Graph Simulation* (BGS) was proposed, which has fewer restrictions but more capacity to extract more useful subgraphs with better efficiency because it supports simulation relations instead of an exact match of edges and nodes. The subgraph isomorphism-based and BGS-based subgraph matching methods aim to find the entire subgraphs in G_D. However, in some applications, such as group finding and expert recommendation, people are more interested in finding some nodes based on a specified structure between them, leading to the *Graph Pattern based Node Matching* (GPNM) problem. In real life, the pattern graph and data graph are usually updated frequently over time. The existing GPNM methods do

not consider any update of a pattern graph or a data graph. Therefore, with the updates of a pattern graph and/or a data graph, they have to perform a new GPNM procedure from scratch to deliver the node matching results, which consumes much more query processing time.

In this thesis, we focus on the three challenges in improving the efficiency of delivering the GPNM results.

(1) The first challenge is how to efficiently deliver the node matching results rather than performing a whole GPNM procedure from scratch (that consumes much more query processing time) when facing frequent updating pattern graphs and data graphs.

(2) Although both the pattern graphs and data graphs are updated frequently, not all the updates in a pattern graph G_P or a data graph G_D essentially affect the GPNM matching results. For example, if one edge (node) is firstly removed from (or inserted into) G_D (G_P) and then inserted back to (or removed from) G_D (G_P), the effects of the two updates can eliminate each other. It is non-trivial to identify the elimination relationships among the updates because there exist both single-graph elimination relationships and cross-graph elimination relationships. Therefore, the second challenge of this thesis is: *how to effectively detect the elimination relationships of the updates.* In addition, if update U_a eliminates update U_b, and update U_b eliminates update U_c, there exists a hierarchical structure of them, which applies to all the elimination relationships. As it is computationally expensive to deliver GPNM results by investigating each of the elimination relationships among the updates, it is beneficial to build up an index to record the hierarchical structure of all the elimination relationships. Therefore, another challenging problem of elimination relationships is *how to build up an index structure to record the hierarchical structure of all the elimination relationships covering both single-graph elimination relationships and cross-graph elimination relationships, which supports the development of an efficient algorithm to deliver the GPNM results by making use of the index.*

(3) In the GPNM procedure, we need to inspect whether the shortest path length between two nodes can satisfy the path length constraints on the pattern graph. The com-

putation of the shortest path length between any two nodes is very time-consuming especially in large data graphs (e.g., social networks and traffic networks). In order to overcome this bottleneck, we propose a strategy to partition the graph into subgraph to speed up the GPNM procedure. In the partition strategy, we need to ensure that the connectivity of the data graph will not be destroyed and the shortest path length between any two nodes can be efficiently updated when the graphs are updated. Therefore, the third challenge of this thesis is *how to efficiently compute the shortest path length between any two nodes to accelerate the GPNM procedure without destroying the connectivity of the graphs.*

Targeting these three challenges, we have proposed our solutions.

- We first proposed an INCremental Graph Pattern node Matching method, called INC-GPNM, to deliver the GPNM results based on the updates of both pattern graphs and data graphs. Instead of recomputing the GPNM results from scratch when both pattern graphs and data graphs are updated, in GPNM, we first build an index to incrementally record the shortest path length range between different label types in G_D, and then identify the affected parts of G_D in GPNM including nodes and edges w.r.t. the updates of G_P and G_D. Moreover, based on the index structure and our novel search strategies, INC-GPNM can efficiently deliver node matching results taking the updates of G_P and G_D as input, and can greatly reduce the query processing time with improved time complexity. Extensive experiments on seven real-world social graphs demonstrate that our method greatly outperforms the GPNM method in efficiency.

- In real life, many typical pattern graphs frequently and repeatedly appear in users' queries in a short period of time, e.g., social graph searches on Facebook. To deliver a GPNM result in such applications, the existing GPNM methods have to recompute the matching results starting from scratch or perform incremental GPNM procedure for each of the updates in the data graph, which are computationally expensive. To address this problem, in this paper, we first analyze

the elimination relationships between multiple updates in G_D and the hierarchical structure between these elimination relationships. Then, we generate an Elimination Hierarchy Tree (EH-Tree) to index the elimination relationships and propose an EH-Tree based GPNM method, called EH-GPNM, considering the elimination relationships between multiple updates in G_D. EH-GPNM first delivers the GPNM result of an initial query, and then delivers the GPNM result of a subsequent query, based on the initial GPNM result and the multiple updates of G_D that occur between those two queries. The experimental results on five real-world social graphs demonstrate that our proposed EH-GPNM is much more efficient.

- Inspired by EH-GPNM, we realized that the elimination relationships not only exist among the updates in the data graph, but also among the updates in the pattern graph and even in the cross updates from G_P and G_D. To further improve the GPNM efficiency when both G_P and G_D are updated frequently, in this thesis, we propose a more efficient GPNM method, called UA-GPNM. UA-GPNM first detect the elimination relations between multiple independent updates in G_P and G_D, and also the cross elimination relationships between the updates from G_P and G_D, then UA-GPNM generates an EH-Tree to index all the elimination relationships. In addition, we also propose a graph partition strategy in UA-GPNM to speed up the GPNM procedure. The experiments show that UA-GPNM can achieve better efficiency compared with INC-GPNM and EH-GPNM when facing the updates of G_P and G_D.

6.2 Future Work

In this thesis, we have mainly discussed the incremental GPNM problem. There are still unresolved issues and thus, our future work will include the following:

- In the GPNM procedure, our algorithms can find the nodes that can match the

pattern graphs. In large real-life social graphs, the excessive number of matching nodes may be returned by our algorithms. In our future work, we aim to propose a measurement to rank these matching nodes and can efficiently select Top-k matching nodes from all the matching results.

- In BGS, the shortest path length is the constraint in the pattern graphs. Therefore, in our matching procedure, we use a matrix to record the shortest path length between any two nodes in the data graphs, which consumes a large amount of space. In our future work, we aim to propose a new index to record the shortest path length or propose a new technique to compress the storage space.

Appendix A

The Notations in the Thesis

Table A.1: The Notations in Chapter 3

Notations	Explanations
G_D	a data graph
G_P	a pattern graph
G_{D_new}	an updated data graph
G_{P_new}	an updated pattern graph
$\triangle G_D$	the updates of G_D
$\triangle G_P$	the updates of G_P
$e(v_i, v_j)$	a directed edge from v_i to v_j
V	a set of vertices in G_D
E	a set of edges in G_D
$f_A(u)$	the attributes of a node u in G_D
V_P	a set of vertices in G_P
E_P	a set of edges in G_P
$f_v(u)$	the attributes of a node u in G_P
$f_e(u, v)$	the bounded path length on $e(u, v)$ in G_P
$M(G_P, G_D)$	the matching result of G_P in G_D based on BGS
N_{u_i}	the matching nodes set of u_i
N'_{u_i}	the updated matching nodes set of u_i
$\triangle G^+_{P_E}$	the insertions of edges for G_P
$\triangle G^-_{P_E}$	the deletions of edges for G_P
$\triangle G^+_{P_N}$	the insertions of nodes for G_P
$\triangle G^-_{P_N}$	the deletions of nodes for G_P
$\triangle G^+_{D_E}$	the insertions of edges for G_D
$\triangle G^-_{D_E}$	the deletions of edges for G_D
$\triangle G^+_{D_N}$	the insertions of nodes for G_D
$\triangle G^-_{D_N}$	the deletions of nodes for G_D

Table A.2: The Notations in Chapter 3 (continued)

Notations	**Explanations**
$SLen$	the shortest path length matrix between each pair of nodes in G_D
R_{SLen}	the shortest path length range matrix between each category of nodes in G_D
AFF	the set of affected pairs of nodes where the shortest path length between each pair of nodes is changed.

Table A.3: The Notations in Chapter 4

Notations	**Explanations**
G_D	a data graph
G_P	a pattern graph
G_{D_new}	an updated data graph
$\triangle G_D$	the updates of G_D
$e(v_i, v_j)$	a directed edge from v_i to v_j
V	a set of vertices in G_D
E	a set of edges in G_D
$f_A(u)$	the attributes of a node u in G_D
V_P	a set of vertices in G_P
E_P	a set of edges in G_P
$f_v(u)$	the attributes of a node u in G_P
$f_e(u, v)$	the bounded path length on $e(u, v)$ in G_P
$M(G_P, G_D)$	the matching result of G_P in G_D based on BGS
$IQuery$	the GPNM result of the initial query
$SQuery$	the GPNM result of the subsequent query
$\triangle G^+_{D_E}$	the insertions of edges for G_D
$\triangle G^-_{D_E}$	the deletions of edges for G_D
$\triangle G^+_{D_N}$	the insertions of nodes for G_D
$\triangle G^-_{D_N}$	the deletions of nodes for G_D
$\triangle G^+_D$	the insertions of edges or nodes for G_D
$\triangle G^-_D$	the deletions of nodes or nodes for G_D
U_i	one update in $\triangle G_D$
$SLen$	the shortest path length matrix between each pair of nodes in G_D
$Aff_N(U_i)$	the set of affected nodes of U_i
$Aff_N(U_a, U_b)$	the set of affected nodes with U_a and U_b
$AFF[u_i, v_j] = [a, b]$	the shortest path length from u_i to v_j is changed from a to b

Table A.4: The Notations in Chapter 5

Notations	Explanations
G_D	a data graph
G_P	a pattern graph
G_{D_new}	an updated data graph
G_{P_new}	an updated pattern graph
$\triangle G_D$	the updates of G_D
$\triangle G_P$	the updates of G_P
$e(v_i, v_j)$	a directed edge from v_i to v_j
V	a set of vertices in G_D
E	a set of edges in G_D
$f_A(u)$	the attributes of a node u in G_D
V_P	a set of vertices in G_P
E_P	a set of edges in G_P
$f_v(u)$	the attributes of a node u in G_P
$f_e(u, v)$	the bounded path length on $e(u, v)$ in G_P
$M(G_P, G_D)$	the matching result of G_P in G_D based on BGS
$IQuery$	the GPNM result of the initial query
$SQuery$	the GPNM result of the subsequent query
$\triangle G^+_{D_E}$ / $\triangle G^-_{D_E}$	the insertions / deletions of edges for G_D
$\triangle G^+_{D_N}$ / $\triangle G^-_{D_N}$	the insertions / deletions of nodes for G_D
$\triangle G^+_D$ / $\triangle G^-_D$	the insertions / deletions of edges or nodes for G_D
$\triangle G^+_{P_E}$ / $\triangle G^-_{P_E}$	the insertions / deletions of edges for G_P
$\triangle G^+_{P_N}$ / $\triangle G^-_{P_N}$	the insertions / deletions of nodes for G_P
$\triangle G^+_P$ / $\triangle G^-_P$	the insertions / deletions of edges or nodes for G_P
U_{Di}	one update in $\triangle G_D$
U_{Pi}	one update in $\triangle G_P$
$SLen$	the shortest path length matrix between each pair of nodes in G_D
$Can_N(U_{Pi})$	the set of candidate nodes of U_{Pi}
$Aff_N(U_{Di})$	the set of affected nodes of U_{Di}
$AFF[u_i, v_j] = [a, b]$	the shortest path length from u_i to v_j is changed from a to b
P_i	one partition
$IB(P_i)$	the set of inner bridge nodes of P_i
$OB(P_i)$	the set of outer bridge nodes of P_i
v_{P_i}	one node in P_i

Appendix B

The Acronyms in the Thesis

Table B.1: The Acronyms in All the Chapters

Sections	**Explanations**	**Acronyms**
Chapter 1&2&3&4&5&6	Graph Pattern based Node Matching	GPNM
Chapter 1&2&3&4&5&6	Graph Pattern Matching	GPM
Chapter 1&3&4&5	Shortest Path Length Matrix	SLen
Chapter 1&2	Bounded Graph Simulation	BGS
Chapter 1	Subgraph Isomorphism	SI
Chapter 3	Shortest Path Length Range Matrix	R_{SLen}
Chapter 4&5	Detect Elimination Relationships	DER
Chapter 4&5	Elimination Hierarchy Tree	EH-Tree

Bibliography

[1] G. M. Adelson-Velskii and E. M. Landis. An information organization algorithm. In *Doklady Akademia Nauk SSSR*, volume 146, pages 263–266, 1962.

[2] B. Aleman-Meza, C. Halaschek-Wiener, S. S. Sahoo, A. Sheth, and I. B. Arpinar. Template based semantic similarity for security applications. In *International Conference on Intelligence and Security Informatics*, pages 621–622. Springer, 2005.

[3] N. Armenatzoglou, R. Ahuja, and D. Papadias. Geo-social ranking: functions and query processing. *The VLDB JournalThe International Journal on Very Large Data Bases*, 24(6):783–799, 2015.

[4] N. Armenatzoglou, S. Papadopoulos, and D. Papadias. A general framework for geo-social query processing. *Proceedings of the VLDB Endowment*, 6(10):913–924, 2013.

[5] J. P. Bagrow. Evaluating local community methods in networks. *Journal of Statistical Mechanics: Theory and Experiment*, 2008(05):P05001, 2008.

[6] J. Bang-Jensen and G. Z. Gutin. *Digraphs: theory, algorithms and applications*. Springer Science & Business Media, 2008.

[7] N. Barbieri, F. Bonchi, E. Galimberti, and F. Gullo. Efficient and effective community search. *Data mining and knowledge discovery*, 29(5):1406–1433, 2015.

[8] A. Baykasoglu, T. Dereli, and S. Das. Project team selection using fuzzy optimization approach. *Cybernetics and Systems: An International Journal*, 38(2):155–185, 2007.

[9] N. Bell and M. Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proceedings of the conference on high performance computing networking, storage and analysis*, page 18. ACM, 2009.

[10] T. Y. Berger-Wolf and J. Saia. A framework for analysis of dynamic social networks. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 523–528. ACM, 2006.

[11] G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, and S. Sudarshan. Keyword searching and browsing in databases using banks. In *Proceedings 18th International Conference on Data Engineering*, pages 431–440. IEEE, 2002.

[12] F. Bi, L. Chang, X. Lin, and W. Zhang. An optimal and progressive approach to online search of top-k influential communities. *Proceedings of the VLDB Endowment*, 11(9):1056–1068, 2018.

[13] P. Bogdanov, B. Baumer, P. Basu, A. Bar-Noy, and A. K. Singh. As strong as the weakest link: Mining diverse cliques in weighted graphs. In *Joint European conference on machine learning and knowledge discovery in databases*, pages 525–540. Springer, 2013.

[14] K. M. Borgwardt, H.-P. Kriegel, and P. Wackersreuther. Pattern mining in frequent dynamic subgraphs. In *Sixth International Conference on Data Mining (ICDM'06)*, pages 818–822. IEEE, 2006.

[15] C. Bothorel, J. D. Cruz, M. Magnani, and B. Micenkova. Clustering attributed graphs: models, measures and methods. *Network Science*, 3(3):408–444, 2015.

[16] U. Brandes, J. Lerner, U. Nagel, and B. Nick. Structural trends in network ensembles. In *Complex networks*, pages 83–97. Springer, 2009.

[17] N. Bruno, N. Koudas, and D. Srivastava. Holistic twig joins: optimal xml pattern matching. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 310–321. ACM, 2002.

[18] J. Brynielsson, J. Högberg, L. Kaati, C. Mårtenson, and P. Svenson. Detecting social positions using simulation. In *2010 International Conference on Advances in Social Networks Analysis and Mining*, pages 48–55. IEEE, 2010.

[19] H. Bunke and B. T. Messmer. Efficient attributed graph matching and its application to image analysis. In *International Conference on Image Analysis and Processing*, pages 44–55. Springer, 1995.

[20] V. Carletti, P. Foggia, P. Ritrovato, M. Vento, and V. Vigilante. A parallel algorithm for subgraph isomorphism. In *International Workshop on Graph-Based Representations in Pattern Recognition*, pages 141–151. Springer, 2019.

[21] D. Chakrabarti and C. Faloutsos. Graph mining: Laws, generators, and algorithms. *ACM computing surveys (CSUR)*, 38(1):2, 2006.

[22] M. Charikar. Greedy approximation algorithms for finding dense components in a graph. In *International Workshop on Approximation Algorithms for Combinatorial Optimization*, pages 84–95. Springer, 2000.

[23] L. Chen, A. Gupta, and M. E. Kurul. Stack-based algorithms for pattern matching on dags. In *Proceedings of the 31st international conference on Very large data bases*, pages 493–504. VLDB Endowment, 2005.

[24] S. Chen, R. Wei, D. Popova, and A. Thomo. Efficient computation of importance based communities in web-scale networks using a single machine. In *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*, pages 1553–1562. ACM, 2016.

[25] S.-J. Chen and L. Lin. Modeling team member characteristics for the formation of a multifunctional team in concurrent engineering. *IEEE Transactions on Engineering Management*, 51(2):111–124, 2004.

[26] H. Cheng, Y. Zhou, X. Huang, and J. X. Yu. Clustering large attributed information networks: an efficient incremental computing approach. *Data Mining and Knowledge Discovery*, 25(3):450–477, 2012.

[27] J. Cheng, J. X. Yu, B. Ding, S. Y. Philip, and H. Wang. Fast graph pattern matching. In *2008 IEEE 24th International Conference on Data Engineering*, pages 913–922. IEEE, 2008.

[28] A. Clauset. Finding local community structure in networks. *Physical review E*, 72(2):026132, 2005.

[29] T. Coffman, S. Greenblatt, and S. Marcus. Graph-based technologies for intelligence analysis. *Communications of the ACM*, 47(3):45–47, 2004.

[30] D. J. Cook and L. B. Holder. *Mining graph data*. John Wiley & Sons, 2006.

[31] L. Cordella, P. Foggia, C. Sansone, and M. Vento. Evaluating performance of the vf graph matching algorithm. In *Proc. of the 10th International Conference on Image Analysis and Processing, IEEE Computer Society Press*, pages 1172–1177, 1999.

[32] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. An improved algorithm for matching large graphs. In *3rd IAPR-TC15 workshop on graph-based representations in pattern recognition*, pages 149–159, 2001.

[33] W. Cui, Y. Xiao, H. Wang, Y. Lu, and W. Wang. Online search of overlapping communities. In *Proceedings of the 2013 ACM SIGMOD international conference on Management of data*, pages 277–288. ACM, 2013.

[34] W. Cui, Y. Xiao, H. Wang, and W. Wang. Local search of communities in large graphs. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 991–1002. ACM, 2014.

[35] H. S. de Andrade and C. L. Sales. Pattern match query in a large graph database. *Encontros Universitários da UFC*, 2(1):1544, 2009.

[36] B. Ding, J. X. Yu, S. Wang, L. Qin, X. Zhang, and X. Lin. Finding top-k min-cost connected trees in databases. In *2007 IEEE 23rd International Conference on Data Engineering*, pages 836–845. IEEE, 2007.

[37] S. Djoko. Substructure discovery using minimum description length principle and background knowledge. In *AAAI*, page 1442, 1994.

[38] W. Fan, J. Li, S. Ma, N. Tang, and Y. Wu. Adding regular expressions to graph reachability and pattern queries. In *2011 IEEE 27th International Conference on Data Engineering*, pages 39–50. IEEE, 2011.

[39] W. Fan, J. Li, S. Ma, N. Tang, Y. Wu, and Y. Wu. Graph pattern matching: from intractable to polynomial time. *Proceedings of the VLDB Endowment*, 3(1-2):264–275, 2010.

[40] W. Fan, X. Wang, and Y. Wu. Diversified top-k graph pattern matching. *Proceedings of the VLDB Endowment*, 6(13):1510–1521, 2013.

[41] W. Fan, X. Wang, and Y. Wu. Incremental graph pattern matching. *ACM Transactions on Database Systems (TODS)*, 38(3):18, 2013.

[42] W. Fan, Y. Wu, and J. Xu. Adding counting quantifiers to graph patterns. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1215–1230. ACM, 2016.

[43] Y. Fang, R. Cheng, S. Luo, and J. Hu. Effective community search for large attributed graphs. *Proceedings of the VLDB Endowment*, 9(12):1233–1244, 2016.

[44] A. Gajewar and A. Das Sarma. Multi-skill collaborative teams based on densest

subgraphs. In *Proceedings of the 2012 SIAM International Conference on Data Mining*, pages 165–176. SIAM, 2012.

[45] B. Gallagher. Matching structure and semantics: A survey on graph-based pattern matching. In *AAAI Fall Symposium: Capturing and Using Patterns for Evidence Detection*, pages 45–53, 2006.

[46] M. R. Garey and D. S. Johnson. *Computers and intractability*, volume 29. wh freeman New York, 2002.

[47] S. Gillani, G. Picard, and F. Laforest. Continuous graph pattern matching over knowledge graph streams. In *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems*, pages 214–225. ACM, 2016.

[48] R. Giugno and D. Shasha. Graphgrep: A fast and universal method for querying graphs. In *Object recognition supported by user interaction for service robots*, volume 2, pages 112–115. IEEE, 2002.

[49] A. V. Goldberg. *Finding a maximum density subgraph*. University of California Berkeley, CA, 1984.

[50] K. Golenberg, B. Kimelfeld, and Y. Sagiv. Keyword proximity search in complex data graphs. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 927–940. ACM, 2008.

[51] G. Gou and R. Chirkova. Efficient algorithms for exact ranked twig-pattern matching over graphs. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 581–594. ACM, 2008.

[52] S. Greenblatt, S. Marcus, and T. Darr. Tmods-integrated fusion dashboard-applying fusion of fusion systems to counter-terrorism. In *Proc. International Conference on Intelligence Analysis*, 2005.

[53] D. Greene, D. Doyle, and P. Cunningham. Tracking the evolution of communities in dynamic social networks. In *2010 international conference on advances in social networks analysis and mining*, pages 176–183. IEEE, 2010.

[54] S. Günnemann, B. Boden, and T. Seidl. Db-csc: a density-based approach for subspace clustering in graphs with feature vectors. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 565–580. Springer, 2011.

[55] H. He, H. Wang, J. Yang, and P. S. Yu. Blinks: ranked keyword searches on graphs. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 305–316. ACM, 2007.

[56] X. Huang, H. Cheng, R.-H. Li, L. Qin, and J. X. Yu. Top-k structural diversity search in large networks. *Proceedings of the VLDB Endowment*, 6(13):1618–1629, 2013.

[57] X. Huang, H. Cheng, R.-H. Li, L. Qin, and J. X. Yu. Top-k structural diversity search in large networks. *The VLDB JournalThe International Journal on Very Large Data Bases*, 24(3):319–343, 2015.

[58] X. Huang, H. Cheng, L. Qin, W. Tian, and J. X. Yu. Querying k-truss community in large and dynamic graphs. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 1311–1322. ACM, 2014.

[59] X. Huang, H. Cheng, and J. X. Yu. Dense community detection in multi-valued attributed networks. *Information Sciences*, 314:77–99, 2015.

[60] X. Huang and L. V. Lakshmanan. Attribute-driven community search. *Proceedings of the VLDB Endowment*, 10(9):949–960, 2017.

[61] X. Huang, L. V. Lakshmanan, and J. Xu. Community search over big graphs: Models, algorithms, and opportunities. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*, pages 1451–1454. IEEE, 2017.

[62] X. Huang, L. V. Lakshmanan, J. X. Yu, and H. Cheng. Approximate closest community search in networks. *Proceedings of the VLDB Endowment*, 9(4):276–287, 2015.

[63] V. Kacholia, S. Pandit, S. Chakrabarti, S. Sudarshan, R. Desai, and H. Karambelkar. Bidirectional expansion for keyword search on graph databases. In *Proceedings of the 31st international conference on Very large data bases*, pages 505–516. VLDB Endowment, 2005.

[64] M. Kargar and A. An. Discovering top-k teams of experts with/without a leader in social networks. In *Proceedings of the 20th ACM international conference on Information and knowledge management*, pages 985–994. ACM, 2011.

[65] M. Kargar and A. An. Keyword search in graphs: Finding r-cliques. *Proceedings of the VLDB Endowment*, 4(10):681–692, 2011.

[66] K. Kim, I. Seo, W.-S. Han, J.-H. Lee, S. Hong, H. Chafi, H. Shin, and G. Jeong. Turboflux: A fast continuous subgraph matching system for streaming graph data. In *Proceedings of the 2018 International Conference on Management of Data*, pages 411–426. ACM, 2018.

[67] Y. Koren, S. C. North, and C. Volinsky. Measuring and extracting proximity graphs in networks. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 1(3):12, 2007.

[68] R. Kumar, J. Novak, and A. Tomkins. Structure and evolution of online social networks. In *Link mining: models, algorithms, and applications*, pages 337–357. Springer, 2010.

[69] L. Lai, L. Qin, X. Lin, and L. Chang. Scalable subgraph enumeration in mapreduce: a cost-oriented approach. *The VLDB JournalThe International Journal on Very Large Data Bases*, 26(3):421–446, 2017.

[70] T. Lappas, K. Liu, and E. Terzi. Finding a team of experts in social networks. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 467–476. ACM, 2009.

[71] E. L. Lawler. A procedure for computing the k best solutions to discrete optimization problems and its application to the shortest path problem. *Management science*, 18(7):401–405, 1972.

[72] V. E. Lee, N. Ruan, R. Jin, and C. Aggarwal. A survey of algorithms for dense subgraph discovery. In *Managing and Mining Graph Data*, pages 303–336. Springer, 2010.

[73] J. Leskovec and J. J. Mcauley. Learning to discover social circles in ego networks. In *Advances in neural information processing systems*, pages 539–547, 2012.

[74] R.-H. Li, Q. Dai, L. Qin, G. Wang, X. Xiao, J. X. Yu, and S. Qiao. Efficient signed clique search in signed networks. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pages 245–256. IEEE, 2018.

[75] R.-H. Li, L. Qin, J. X. Yu, and R. Mao. Influential community search in large networks. *Proceedings of the VLDB Endowment*, 8(5):509–520, 2015.

[76] R.-H. Li, J. Su, L. Qin, J. X. Yu, and Q. Dai. Persistent community search in temporal networks. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pages 797–808. IEEE, 2018.

[77] Y. Li, R. Chen, J. Xu, Q. Huang, H. Hu, and B. Choi. Geo-social k-cover group queries for collaborative spatial computing. *IEEE Transactions on Knowledge and Data Engineering*, 27(10):2729–2742, 2015.

[78] Y. Li, L. Zou, M. T. Özsu, and D. Zhao. Time constrained continuous subgraph search over streaming graphs. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 1082–1093. IEEE, 2019.

[79] G. Liu, Q. Shi, K. Zheng, A. Liu, Z. Li, and X. Zhou. An efficient method for top-k graph based node matching. *World Wide Web*, 22(3):945–966, 2019.

[80] G. Liu, K. Zheng, Y. Wang, M. A. Orgun, A. Liu, L. Zhao, and X. Zhou. Multi-constrained graph pattern matching in large-scale contextual social graphs. In *2015 IEEE 31st International Conference on Data Engineering*, pages 351–362. IEEE, 2015.

[81] Z. Liu and Y. Chen. Identifying meaningful return information for xml keyword search. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 329–340. ACM, 2007.

[82] Z. Liu and Y. Chen. Identifying meaningful return information for xml keyword search. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 329–340. ACM, 2007.

[83] B. Lyu, L. Qin, X. Lin, L. Chang, and J. X. Yu. Scalable supergraph search in large graph databases. In *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*, pages 157–168. IEEE, 2016.

[84] S. Ma, Y. Cao, W. Fan, J. Huai, and T. Wo. Strong simulation: Capturing topology in graph pattern matching. *ACM Transactions on Database Systems (TODS)*, 39(1):4, 2014.

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

[86] B. T. Messmer. Efficient graph matching algorithms. 1995.

[87] B. T. Messmer and H. Bunke. *Subgraph isomorphism in polynomial time*. Universität Bern. Institut für Informatik und Angewandte Mathematik, 1995.

[88] S. Milgram. The small world problem. *Psychology today*, 2(1):60–67, 1967.

[89] M. R. Morris, J. Teevan, and K. Panovich. What do people ask their social networks, and why?: a survey study of status message q&a behavior. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 1739–1748. ACM, 2010.

[90] R. Myers, R. Wison, and E. R. Hancock. Bayesian graph edit distance. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 22(6):628–635, 2000.

[91] M. H. Namaki, P. Lin, and Y. Wu. Event pattern discovery by keywords in graph streams. In *2017 IEEE International Conference on Big Data (Big Data)*, pages 982–987. IEEE, 2017.

[92] A. Ntoulas, J. Cho, and C. Olston. What's new on the web?: the evolution of the web from a search engine perspective. In *Proceedings of the 13th international conference on World Wide Web*, pages 1–12. ACM, 2004.

[93] S. Parthasarathy, S. Tatikonda, and D. Ucar. A survey of graph mining techniques for biological datasets. In *Managing and mining graph data*, pages 547–580. Springer, 2010.

[94] G. Ramalingam and T. Reps. On the computational complexity of dynamic graph problems. *Theoretical Computer Science*, 158(1-2):233–277, 1996.

[95] Y. Ruan, D. Fuhry, and S. Parthasarathy. Efficient community detection in large networks using content and links. In *Proceedings of the 22nd international conference on World Wide Web*, pages 1089–1098. ACM, 2013.

[96] M. Rudolf. Utilizing constraint satisfaction techniques for efficient graph pattern matching. In *International Workshop on Theory and Application of Graph Transformations*, pages 238–251. Springer, 1998.

[97] D. C. Schmidt and L. E. Druffel. A fast backtracking algorithm to test directed graphs for isomorphism using distance matrices. *Journal of the ACM (JACM)*, 23(3):433–445, 1976.

[98] K. Semertzidis and E. Pitoura. Durable graph pattern queries on historical graphs. In *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*, pages 541–552. IEEE, 2016.

[99] L. G. Shapiro and R. M. Haralick. Structural descriptions and inexact matching. *IEEE transactions on pattern analysis and machine intelligence*, (5):504–519, 1981.

[100] D. Shasha, J. T. Wang, and R. Giugno. Algorithmics and applications of tree and graph searching. In *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 39–52. ACM, 2002.

[101] Q. Shi, G. Liu, K. Zheng, A. Liu, Z. Li, L. Zhao, and X. Zhou. Multi-constrained top-k graph pattern matching in contextual social graphs. In *2017 IEEE International Conference on Web Services (ICWS)*, pages 588–595. IEEE, 2017.

[102] N. Shrivastava, A. Majumder, and R. Rastogi. Mining (social) network graphs to detect random link attacks. In *2008 IEEE 24th International Conference on Data Engineering*, pages 486–495. IEEE, 2008.

[103] M. Sozio and A. Gionis. The community-search problem and how to plan a successful cocktail party. In *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 939–948. ACM, 2010.

[104] A. Stotz, R. Nagi, and M. Sudit. Incremental graph matching for situation awareness. In *2009 12th International Conference on Information Fusion*, pages 452–459. IEEE, 2009.

[105] G. Sun, G. Liu, Y. Wang, M. A. Orgun, Q. Z. Sheng, and X. Zhou. Incremental graph pattern based node matching with multiple updates. *IEEE Transactions on Knowledge and Data Engineering*, 2019.

[106] G. Sun, G. Liu, Y. Wang, M. A. Orgun, and X. Zhou. Incremental graph pattern based node matching. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pages 281–292. IEEE, 2018.

[107] S. Sun, Y. Che, L. Wang, and Q. Luo. Efficient parallel subgraph enumeration on a single machine. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 232–243. IEEE, 2019.

[108] L. Tang and H. Liu. Graph mining applications to social network analysis. In *Managing and Mining Graph Data*, pages 487–513. Springer, 2010.

[109] H. Tong and C. Faloutsos. Center-piece subgraphs: problem definition and fast solutions. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 404–413. ACM, 2006.

[110] H. Tong, C. Faloutsos, C. Faloutsos, B. Gallagher, and T. Eliassi-Rad. Fast best-effort pattern matching in large attributed graphs. In *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 737–746. ACM, 2007.

[111] H. Tong, C. Faloutsos, C. Faloutsos, B. Gallagher, and T. Eliassi-Rad. Fast best-effort pattern matching in large attributed graphs. In *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 737–746. ACM, 2007.

[112] W.-H. Tsai and K.-S. Fu. Error-correcting isomorphisms of attributed relational graphs for pattern analysis. *IEEE Transactions on systems, man, and cybernetics*, 9(12):757–768, 1979.

[113] J. Ugander, L. Backstrom, C. Marlow, and J. Kleinberg. Structural diversity in social contagion. *Proceedings of the National Academy of Sciences*, 109(16):5962–5966, 2012.

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

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

[116] B. Viswanath, A. Post, K. P. Gummadi, and A. Mislove. An analysis of social network-based sybil defenses. *ACM SIGCOMM Computer Communication Review*, 41(4):363–374, 2011.

[117] C. Wang and L. Chen. Continuous subgraph pattern search over graph streams. In *2009 IEEE 25th International Conference on Data Engineering*, pages 393–404. IEEE, 2009.

[118] X. Wang, L. Chai, Q. Xu, Y. Yang, J. Li, J. Wang, and Y. Chai. Efficient subgraph matching on large rdf graphs using mapreduce. *Data Science and Engineering*, 4(1):24–43, 2019.

[119] T. Washio and H. Motoda. State of the art of graph-based data mining. *Acm Sigkdd Explorations Newsletter*, 5(1):59–68, 2003.

[120] S. White and P. Smyth. Algorithms for estimating relative importance in networks. In *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 266–275. ACM, 2003.

[121] H. Wi, S. Oh, J. Mun, and M. Jung. A team formation model based on knowledge and collaboration. *Expert Systems with Applications*, 36(5):9121–9134, 2009.

[122] M. Wolverton, P. Berry, I. W. Harrison, J. D. Lowrance, D. N. Morley, A. C. Rodriguez, E. H. Ruspini, and J. Thomere. Law: A workbench for approximate pattern matching in relational data. In *IAAI*, volume 3, pages 143–150, 2003.

[123] Y. Wu, R. Jin, J. Li, and X. Zhang. Robust local community detection: on free rider effect and its elimination. *Proceedings of the VLDB Endowment*, 8(7):798–809, 2015.

[124] Z. Xu, Y. Ke, Y. Wang, H. Cheng, and J. Cheng. A model-based approach to attributed graph clustering. In *Proceedings of the 2012 ACM SIGMOD international conference on management of data*, pages 505–516. ACM, 2012.

[125] D.-N. Yang, C.-Y. Shen, W.-C. Lee, and M.-S. Chen. On socio-spatial group query for location-based social networks. In *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 949–957. ACM, 2012.

[126] Y. Yang, D. Yan, H. Wu, J. Cheng, S. Zhou, and J. Lui. Diversified temporal subgraph pattern mining. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1965–1974. ACM, 2016.

[127] Z. Yang, A. W.-C. Fu, and R. Liu. Diversified top-k subgraph querying in a large graph. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1167–1182. ACM, 2016.

[128] J. X. Yu, L. Qin, and L. Chang. Keyword search in databases. *Synthesis Lectures on Data Management*, 1(1):1–155, 2009.

[129] Y. Yuan, G. Wang, and L. Chen. Pattern match query in a large uncertain graph. In *Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management*, pages 519–528. ACM, 2014.

[130] Y. Yuan, G. Wang, L. Chen, and B. Ning. Efficient pattern matching on big uncertain graphs. *Information Sciences*, 339:369–394, 2016.

[131] Y. Yuan, G. Wang, J. Y. Xu, and L. Chen. Efficient distributed subgraph similarity matching. *The VLDB JournalThe International Journal on Very Large Data Bases*, 24(3):369–394, 2015.

[132] Y. Zhang, G. Yin, and Q. Zhao. An incremental graph pattern matching based dynamic cold-start recommendation method. In *International Conference of Pioneering Computer Scientists, Engineers and Educators*, pages 182–195. Springer, 2016.

[133] Y. Zhou, H. Cheng, and J. X. Yu. Graph clustering based on structural/attribute similarities. *Proceedings of the VLDB Endowment*, 2(1):718–729, 2009.

[134] L. Zou, L. Chen, and M. T. Özsu. Distance-join: Pattern match query in a large graph database. *Proceedings of the VLDB Endowment*, 2(1):886–897, 2009.

[135] A. Zzkarian and A. Kusiak. Forming teams: an analytical approach. *IIE transactions*, 31(1):85–97, 1999.