This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2019.2942294, IEEE Transactions on Knowledge and Data Engineering

Incremental Graph Pattern based Node Matching with Multiple Updates

Guohao Sun, Guanfeng Liu, Yan Wang, *Senior Member, IEEE,* Mehmet A. Orgun, *Senior Member, IEEE,* Quan Z. Sheng, *Member, IEEE,* and Xiaofang Zhou, *Fellow, IEEE*

Abstract—*Graph Pattern based Node Matching* (GPNM) has been proposed to find all the matches of the nodes in a data graph G_D based on a given pattern graph G_P. GPNM has been increasingly adopted in many applications such as group finding and expert recommendation, in which data graphs are frequently updated over time. Moreover, 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 perform an incremental GPNM procedure for each of the updates in the data graph, which is 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 than the state-of-the-art GPNM methods.

Index Terms—Graph pattern matching, updates of graph, elimination relationship.

1 Introduction

1.1 Background

Graph Pattern based Subgraph Matching (GPSM) is to find all the matching subgraphs of a pattern graph G_P in a data graph G_D. Conventional subgraph matching solutions are based on the NP-Complete *subgraph isomorphism* problem [1], [2], [3], which makes it computationally expensive to find the exact matching subgraphs. To address this problem, Fan et al., proposed *Bounded Graph Simulation* (BGS) [4], 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 node in G_D and G_P has a label (e.g., representing a person's job title), and 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" [5], on average, any two people can be connected in about six hops. Therefore, k is usually set as a small integer in social networks [4].

- *G. Sun, G. Liu, Y. Wang, M. A. Orgun and Q. Sheng are with the Department of Computing, Macquarie University, Sydney, NSW 2109, Australia.*
  E-mail: guohao.sun@students.mq.edu.au, {guanfeng.liu, yan.wang, mehmet.orgun, michael.sheng}@mq.edu.au
- *X. Zhou is with the School of Information Technology and Electrical Engineering, The University of Queensland, St Lucia, QLD 4072, Australia.*
  E-mail: zxf@itee.uq.edu.au

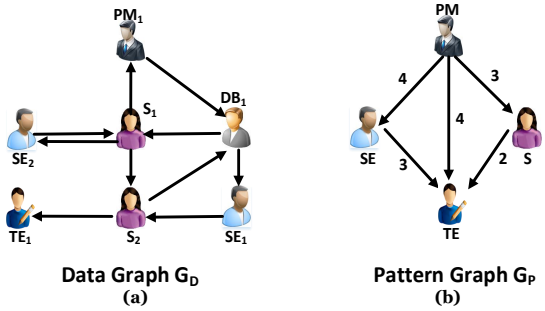


Fig. 1: Graph Pattern based Node Matching

The 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 [6] and expert recommendation [7], [8], 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 [9], with an example discussed below.

TABLE 1: The node matching results of Example 1

Nodes in G_P	Matching nodes in G_D
PM	PM_1
SE	SE_1, SE_2
S	S_1, S_2
TE	TE_1

Example 1 (GPNM Problem): Fig. 1(a) depicts a data graph G_D, where each node denotes a person, labeled with her or his 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(b), where an IT project

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2019.2942294, IEEE Transactions on Knowledge and Data Engineering

needs four types of people, namely, PM, SE, TE, and S respectively. In BGS [4], an edge can be associated with an integer to show the constraint of the maximum path length between two nodes. For example, in Fig.1 (b), a PM needs to connect with an SE, an S and a TE within 4, 3, and 4 hops respectively. The GPNM results are shown in TABLE 1.

The existing subgraph matching methods can be applied to solve the GPNM problem. However, they need to deliver the entire matching subgraphs, rather than matching nodes only, which incurs a high time complexity [4], [10]. Therefore, Fan et al., [11] 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_D that commonly exist in real scenarios [12]. For example, in group finding in Online Social Networks (OSNs) [6], the joining of new users or the withdrawal of existing users in OSNs results in the updates of G_D. When facing each of such updates, the existing GPNM methods [9], [11] have to perform a new GPNM procedure from scratch, wasting resources and leading to low efficiency.

In order to improve efficiency, a state-of-the-art GPNM method, called INC-GPNM [13], has been proposed. INC-GPNM first incrementally records the shortest path length range between different types of labels in G_D and then identifies the affected area of G_D w.r.t. the updates of G_D. The affected area of G_D includes the edges and nodes. As GPNM aims to find the matching nodes in G_D, the affected area of each update can be illustrated by the affected nodes (the shortest path length between these nodes has changed) in each update. Thus, INC-GPNM can improve the efficiency of GPNM when G_D has been updated. However, in a large-scale social graph that is updated with high frequency, INC-GPNM is still computationally expensive as it ignores the relationships that exist among the updates and has to perform an incremental GPNM procedure for each of the updates in G_D. For example, on Facebook, on average, in each minute, 400 new users join in, 510,000 comments are posted, 317,000 statuses are updated, and 147,000 photos are uploaded[1].

1.2 Motivations and Problems

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[2]. 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 1 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 10 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 the state-of-the-art method, INC-GPNM [13], has to perform an incremental GPNM 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 2 illustrates the details of our motivations.

Example 2 (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. 2(a) is the pattern graph corresponding to the initial query. The initial data graph is shown in Fig. 2(c), and the matching result for the initial query based on the initial data graph is shown in Table 2. The subsequent query given by another travel agency staff *Bella* is shown in Fig. 2(b), and three updates occur between the initial query and the subsequent query, i.e., *Update* U_1 in Fig. 2(d), *Update* U_2 in Fig. 2(e) and *Update* U_3 in Fig. 2(f). Fig. 2(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 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 (denoted as *GPNM-MU*), INC-GPNM [13] 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 2.

This example shows that we need to develop a new GPNM solution which considers the multiple updates in

1. https://sproutsocial.com/insights/facebook-stats-for-marketers/
2. http://en.wikipedia.org/wiki/Facebook_Graph_Search

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2019.2942294, IEEE Transactions on Knowledge and Data Engineering

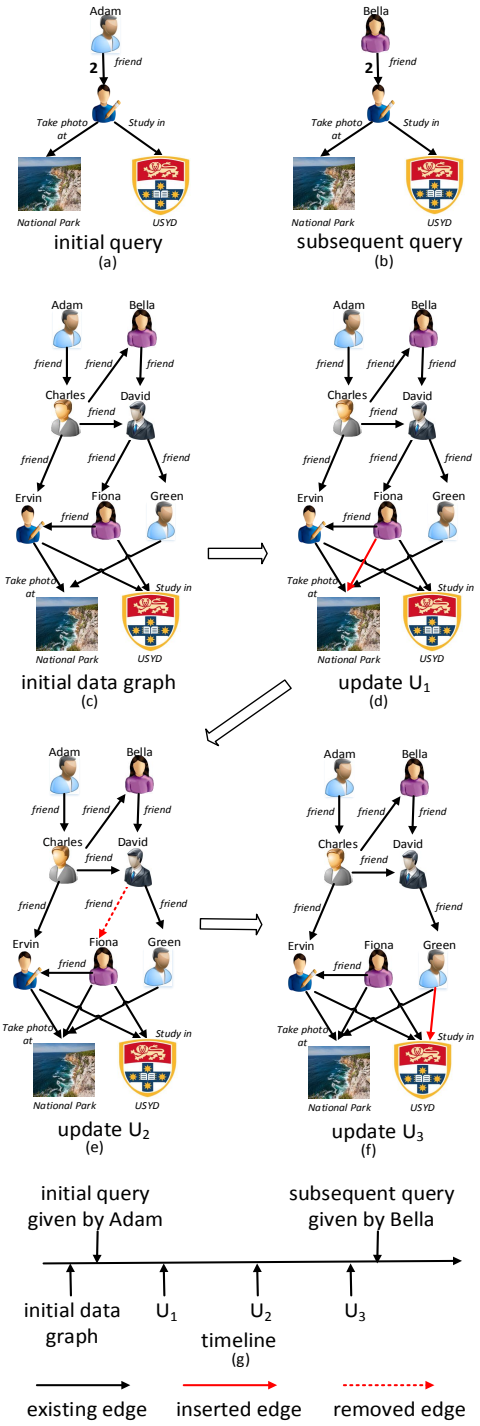


Fig. 2: GPNM with multiple updates

TABLE 2: The matching results of the initial query and the subsequent query in Example 2

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

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 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 our work 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 our work 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.*

1.3 Contributions

In this paper, we propose an efficient GPNM method to answer GPNM queries with multiple updates in data graphs. To the best of our knowledge, our method is the first GPNM solution that takes the elimination relationships between multiple updates in a data graph G_D into consideration. The contributions of our work are summarized as follows:

(1) We first propose an effective method to find the elimination relationships existing in updates by comparing the affected nodes for each pair of updates.

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

(3) We further propose an EH-Tree based GPNM algorithm called EH-GPNM that considers multiple updates in a data graph.

(4) The experiments conducted on five real-world social graphs demonstrate that our EH-GPNM method significantly outperforms the state-of-the-art GPNM methods [11], [13], by reducing the query processing time with an average of 51.23% and 22.59% respectively.

The rest of this paper is organized as follows. We first review the related work in Section 2. Then we introduce the necessary concepts and formulate the main problem in Section 3. Section 4 analyzes the elimination relationships. Section 5 proposes and discusses the EH-Tree index, and the new algorithm, EH-GPNM, in detail. Section 6 discusses the experimental results, and Section 7 concludes the paper.

2 Related Work

The existing GPM methods can be classified into two categories based on their delivered matching results: 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.

2.1 GPSM

GPSM is to find all the matching subgraphs of G_P in G_D. For example, Ren et al., [14] proposed a method to cache

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2019.2942294, IEEE Transactions on Knowledge and Data Engineering

several query graphs and reorder the sequence of these queries to improve the efficiency of query answers. In the light of the intractability of the NP-complete problem of subgraph isomorphism, an approximate solution BGS [4] has been studied to find inexact matching subgraphs. Yuan et al., [15] proposed a method to retrieve all qualified matches of a query pattern in the noisy, incomplete, and inaccurate graph. In [16], they further investigated subgraph similarity matching for a Web-scale graph deployed in a distributed environment. Furthermore, Yuan et al., [17] 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. Lyu et al., [18] proposed an indexing algorithm that selects features directly from data graphs without relying on a frequent subgraph mining algorithm. In the application of community finding, Fang et al., [19] proposed a method which aims to return an attributed community for an attributed graph, in which the attributed community is a subgraph which satisfies both structure cohesiveness and keyword cohesiveness. Fang et al., [20] studied scalable subgraph enumeration in MapReduce, considering that existing solutions for subgraph enumeration are not sufficiently scalable to handle large graphs.

However, social graphs are frequently updated [12], and it is computationally expensive to perform a new procedure from scratch to find matching subgraphs when facing any updates. Therefore, Fan et al., [21] proposed an incremental approximate method to find the matching subgraphs. 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. Semertzidis et al., [22] focused on labeled graphs that evolve over time. They found the matches that exist for the longest period of time. Sun et al., [23] extended incremental methods to find maximal cliques that contain vertices incident to an edge which has been inserted. Fan et al., [24] further proposed incremental algorithms for four types of typical pattern graphs, which can reduce the computations on big graphs and minimize unnecessary re-computation. Ma et al., [25] proposed a method to find dense subgraphs in temporal networks. They focused on a special class of temporal networks, where the weights associated with edges regularly vary with timestamps. Kim et al., [26] proposed a fast-continuous subgraph matching system called TurboFlux which provides a high throughput over a fast graph update stream. Li et al., [27] aimed to identify the communities that are persistent over time in a temporal network, in which every edge is associated with a timestamp. In addition, Li et al., [28] 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 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 [4], [10]. Therefore, several GPNM methods have been proposed, which aim to find some nodes based on a specified structure between those nodes, such as group finding [6] and expert recommendation [7]. Some of them [9], [29], [30] are proposed to find matches of a specific node via *subgraph isomorphism*, which has the exponential complexity. To improve efficiency, Tong et al., [31] proposed a "Seed-Finder" method that identifies approximate matches for certain pattern nodes. This method only requires cubic time. Based on BGS, Fan et al., [11] revised graph patterns to support a specific output node and define functions to measure match relevance and diversity. Motivated by network analysis applications, Fan et al., [32] proposed quantified matching for a specific pattern node, in which they extend traditional graph patterns with counting quantifiers.

To address the GPNM problem when both pattern graphs and data graphs are updated over time, an incremental GPNM method called INC-GPNM has been proposed in [13]. INC-GPNM first builds an index to incrementally record the shortest path length range between different label types in G_D, and then identifies the affected nodes of G_D in GPNM w.r.t. the updates of G_P and G_D. Moreover, based on the proposed index structure and 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.

2.3 Summary

The existing methods in the above two categories face the efficiency issue when answering GPNM queries that are frequently and repeatedly given by users in a short period of time with the updates in data graphs. Firstly, the GPSM methods cannot be applied in GPNM because of the low efficiency of delivering the entire subgraph structures. Secondly, the state-of-the-art GPNM method INC-GPNM [13] cannot offer good efficiency either, though it adopts an incremental GPNM strategy. This is because that INC-GPNM has to perform the incremental procedure for each of the updates separately in G_D, which is still computationally expensive in a large-scale graph that is updated frequently.

3 PRELIMINARIES

In this section, we introduce the concepts of data graph and pattern graph, and the problem of GPNM and the problem of GPNM with Multiple Updates (GPNM-MU). Table 3 lists the notations used in this paper.

3.1 Data Graph and Pattern Graph

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

- V_D is a set of nodes;
- $E_D \subseteq V_D \times V_D$, in which a tuple ($u$, 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 [33].

Example 3: G_D in Fig. 1(a) depicts a data graph, where each node denotes a person, together with the label of a person, e.g., PM means a *Project Manager*. Each edge denotes a relationship between the two connected nodes, e.g., $e(PM_1, DB_1)$ means PM_1 has a collaboration relationship

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2019.2942294, IEEE Transactions on Knowledge and Data Engineering

TABLE 3: Notations used in this paper

Notation	Meaning
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

with DB_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 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 4: G_P in Fig. 1(b) depicts a pattern graph. The nodes and the edges have the same meaning as G_D in Fig. 1(a). In contrast to G_D, each edge in G_P has an integer as the bounded path length.

Remark: In BGS [4], a bounded path length is the constraint for the maximum path length between two nodes. For example, in Fig. 1(b), edge $e(S, TE)$ of G_P is associated with an integer 2, which means a *secretary* needs to connect with a *test engineer* within 2 hops.

3.2 Graph Pattern based Node Matching (GPNM)

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

Example 5: Recall G_P and G_D shown in Fig. 2(a) and Fig. 2(c) respectively. Instead of finding the whole subgraphs, GPNM aims to find the matching nodes in G_D for each node of G_P. Then, *Charles* and *Ervin* are the matching nodes in the subgraphs that can match G_P based on BGS. The node matching results are shown in Table 2.

3.3 Graph Pattern based Node Matching with Multiple Updates (GPNM-MU)

- **Input:** a pattern graph, G_P, a data graph, G_D, the GPNM result of the initial query (termed as $IQuery$), and a sequence of multiple updates $\triangle G_D$ to G_D.
- **Output:** the GPNM result of the subsequent query (termed as $SQuery$) of G_P in G_{D_new} (G_{D_new} denotes the updated G_D).

Remark: $\triangle G_D$ may include 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. We denote each update in $\triangle G_D$ as U_i.

Example 6: Recall G_P and G_D shown in Fig. 2(a) and Fig. 2(c) respectively, $IQuery = \{Charles, Ervin\}$. U_1 is to insert edge $e(Fiona, NationalPark)$ into G_D shown in Fig. 2(d), U_2 is to remove edge $e(David, Fiona)$ from G_D shown in Fig. 2(e) and U_3 is to insert edge $e(Green, USYD)$ into G_D shown in Fig. 2(f). G_{D_new} is shown in Fig. 2(f). After the multiple updates, $SQuery = \{Charles, Ervin, David, Green\}$.

4 Elimination Relationships

4.1 Elimination Relationship Types

In this section, we analyze the *elimination relationships* between the updates in G_D and discuss how they can be detected. As we introduced in Section 1.2, 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* in Section 1.2). 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* in Section 1.2). 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 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 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.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2019.2942294, IEEE Transactions on Knowledge and Data Engineering

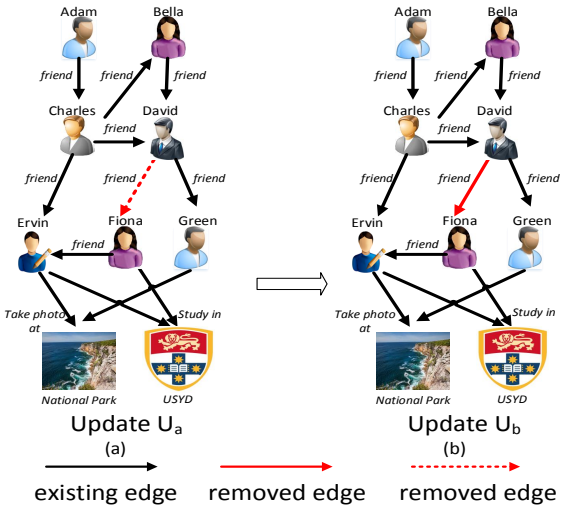


Fig. 3: Elimination Relationship Type I

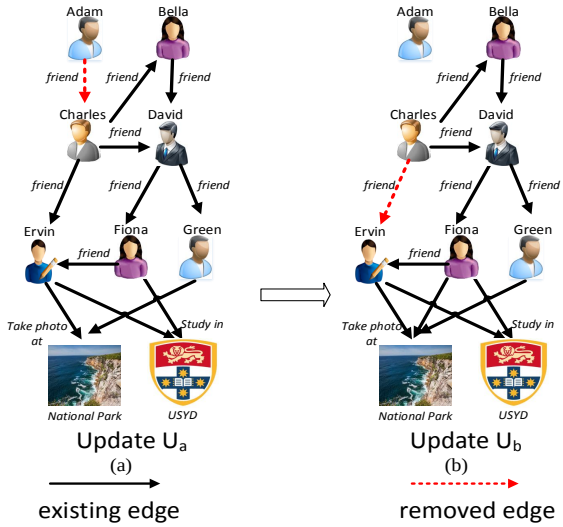


Fig. 4: Elimination Relationship Type II

Definition 1 (Elimination Relationship Type I): 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 $Aff\_N(U_i)$. $Aff\_N(U_a, U_b)$ 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 $Aff\_N(U_a, U_b) = \emptyset$, 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 7: Recall the case shown in Fig. 2, 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. 3(a) and Fig. 3(b) respectively. In this example, since $Aff\_N(U_a, U_b) = \emptyset$, $U_a \Leftrightarrow U_b$.

Elimination Relationship Type II: 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): 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 $Aff\_N(U_i)$. $U_a \succeq U_b$ or $U_b \preceq U_a$ if and only if $Aff\_N(U_a) \supseteq Aff\_N(U_b)$. 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 8: Recall the case shown in Fig. 4. 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. 2 contains the updates in Fig. 3, Fig. 3 illustrates the Elimination Relationship Type I more directly, while Fig. 3 contains all the queries and updates of data graphs, which is used to illustrate our targeted problem and the motivation.

4.2 Detecting Elimination Relationships

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 [34] 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 (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 [34] 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 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 1*.

Detect Elimination Relationship Type II (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 $Aff\_N(U_a) \supseteq Aff\_N(U_b)$, then $U_a \succeq U_b$ or $U_b \preceq U_a$. The pseudo-code is shown in *Algorithm 2*.

Example 9: Recall the case shown in Fig. 4. We first compute the shortest path length matrix $SLen$ for the G_D prior to the updates. The results are shown in Table 4. With U_a and U_b, we update the shortest path length matrices as shown in Table 5 and Table 6 respectively. Compared with $SLen$, with U_a, the shortest path lengths from *Adam*

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2019.2942294, IEEE Transactions on Knowledge and Data Engineering

Algorithm 1: Detect Elimination Relationship Type I (DER-I)

Input: $G_P, G_D, \triangle G_D, SLen$
Output: The elimination relationships of the updates
for *each pair of updates* U_a *and* $U_b \in \triangle G_D$ **do**
  if *the shortest path lengths between the nodes are not affected* **then**
    Keep the shortest path lengths in $SLen\_new$ the same as those in $SLen$;
  else
    Apply Dijkstra's algorithm for updating the shortest path lengths between the affected nodes in $SLen_{new}$;
  if $SLen\_new = SLen$ **then**
    $U_a \Leftrightarrow U_b$;
Return the elimination relationships of the updates;

to all the other nodes 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 $Aff\_N(U_b) = \{Charles, Ervin\}$. Because $Aff\_N(U_a) \supseteq Aff\_N(U_b)$, then $U_a \succeq U_b$ or $U_b \preceq U_a$.

TABLE 4: $SLen$ of G_D in Fig. 2(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

TABLE 5: $SLen_{new}$ with U_a in Fig. 4(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 6: $SLen_{new}$ with U_b in Fig. 4(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

Theorem 1: *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 I.*

The Proof of Theorem 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) $n_i \in Aff\_N(U_b)$ and $n_i \notin Aff\_N(U_a)$; or (2) $n_i \in Aff\_N(U_a)$ and $n_i \notin Aff\_N(U_b)$. If $n_i \in Aff\_N(U_b)$ and $n_i \notin Aff\_N(U_a)$, then n_i is not affected by U_a. However, it contradicts $n_i \in Aff\_N(U_a)$ when U_a is applied to G_D. If $n_i \in Aff\_N(U_a)$ and $n_i \notin Aff\_N(U_b)$, then n_i is not affected by U_b. However, this contradicts

Algorithm 2: Detect Elimination Relationship Type II (DER-II)

Input: $G_P, G_D, \triangle G_D, SLen$
Output: The elimination relationships of the updates
for *each pair of updates* U_a *and* $U_b \in \triangle G_D$ **do**
  if *the shortest path lengths between the nodes are not affected* **then**
    Keep the shortest path lengths in $SLen\_new$ as that in $SLen$;
  else
    Apply Dijkstra's algorithm for updating the shortest path lengths between the affected nodes in $SLen_{new}$;
  Put the affected nodes into $Aff\_N(U_a)$;
  Put the affected nodes into $Aff\_N(U_b)$;
  if $Aff\_N(U_a) \supseteq Aff\_N(U_b)$ **then**
    $U_a \succeq U_b$;
Return the elimination relationships of the updates;

$n_i \in Aff\_N(U_b)$ when U_b is applied to G_D. Therefore, *Theorem 1* is proven. □

Theorem 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 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 2* is proven. □

Complexity: The complexity of the generation and the updates of $SLen$ is $\mathcal{O}(|N_D|(|N_D| + |E_D|)$ [35]. 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.

5 Incremental GPNM Method with Multiple Updates

In this section, we first introduce how to build and maintain an Elimination Hierarchy Tree (EH-Tree) index, and then propose a new GPNM algorithm, called EH-GPNM.

5.1 Algorithm Overview

EH-GPNM contains three major parts, each solving a challenging problem. All of them demonstrate the novelty and

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2019.2942294, IEEE Transactions on Knowledge and Data Engineering

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 6).

5.2 Elimination Hierarchy Tree (EH-Tree)

As introduced in Section 4, 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.

Step 1: Identify the affected nodes of each update. To generate the EH-Tree index, we first need to identify the affected nodes of each update. The steps are as follows:
(1) For each update $U_i \in \triangle G_D$, set $Aff\_N(U_i) = \emptyset$, and then update the $SLen$ to obtain the $SLen_{new}$;
(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 $Aff\_N(U_i)$;
(3) Return $Aff\_N(U_i)$.

TABLE 7: $SLen_{new}$ with U_a in Example 10

	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 8: $SLen_{new}$ with U_b in Example 10

	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

Example 10: Recall the case shown in Fig. 2. Suppose U_a, U_b, U_c and U_d are to remove edge $e(Adam, Charles)$, $e(David, Green)$, $e(David, Fiona)$ and $e(Charles, Ervin)$ from G_D respectively. The corresponding $SLen_{new}$ are shown in Table 7, Table 8, Table 9 and Table 10 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 11.

TABLE 9: $SLen_{new}$ with U_c in Example 10

	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

TABLE 10: $SLen_{new}$ with U_d in Example 10

	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

Step 2: EH-Tree establishment and maintenance.
EH-Tree Establishment: We present the details of the generation of EH-Tree as follows. The pseudo-code is shown in *Algorithm 3*.
(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.
(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.
(3) We then recursively insert all the updates into the EH-Tree.
Remark: EH-Tree adopts the property of a balanced tree to improve query efficiency, and if the affected nodes of one update can be covered by the root, then this update is set as the child tree node of the root. Therefore, the root of the EH-Tree can have more than two 2 children.
Example 11: Recall U_a, U_b, U_c and U_d in Example 10. As U_a has the maximum number of affected nodes in all the

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2019.2942294, IEEE Transactions on Knowledge and Data Engineering

TABLE 11: The affected nodes of the updates in Example 10

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

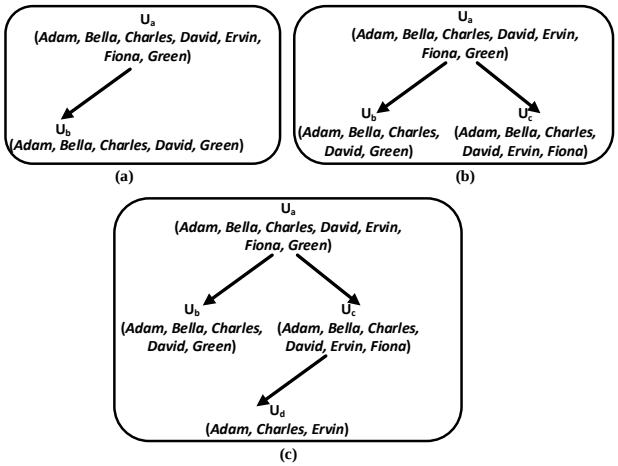


Fig. 5: The EH-Tree of Example 11

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. 5(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. 5(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. 5(c).

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

EH-Tree Maintenance: 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 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 4*.

(1) Firstly, for each new coming update U_n, we identify $Aff\_N(U_n)$ based on the method proposed in Section 5.2.

(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).

(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.

Algorithm 3: EH-Tree Establishment

```
Input: G_P, G_D, △G_D, SLen
Output: the address of the root of an EH-Tree
for each update U_i ∈ △G_D do
    if the shortest path lengths between the the nodes are not affected then
        Keep the shortest path lengths in SLen_new as that in SLen;
    else
        Apply Dijkstra's algorithm for updating the shortest path
          lengths between the affected nodes in SLen_new;
    Compare SLen_new with SLen;
    Put the affected nodes into Aff_N(U_i);
    if U_i has the maximum number of affected nodes then
        U_i is set as the root of an EH-Tree;
    else
        for each update U_j ∈ △G_D do
            if the set affected nodes of U_i covers that of U_j then
                U_j is set as the child node of U_i;
                if the number of affected nodes of U_j is less than or equal to
                  that by other child nodes then
                    U_j is set as the left child node;
                else
                    U_j is set as the right child node;
return the address of the root of an EH-Tree;
```

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 12: Recall U_a, U_b, U_c and U_d in Example 10. The corresponding EH-Tree of these updates is shown in Fig. 5 (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 [36], 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 $\mathcal{O}|\triangle G_D||N_D|$.

5.3 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 updates and then incrementally identifies the GPNM results. The detailed steps of EH-GPNM are shown below. The pseudo-code is shown in Algorithm 5.

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.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2019.2942294, IEEE Transactions on Knowledge and Data Engineering

Algorithm 4: EH-Tree Maintenance

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

```
Identify Aff_N(U_n);
let U_m denote one of the tree nodes in EH-Tree;
let U_root denote the tree root in EH-Tree;
set U_m = U_root;
Function INSERT(U_n, U_m):
    if Aff_N(U_n) ⊇ Aff_N(U_m) and U_m is the tree root in EH-Tree then
        U_n is set as the new root of EH-Tree;
        U_m is set as the only child node of U_n;
    else
        if Aff_N(U_n) ⊂ Aff_N(U_m) then
            let U_l denote the one of the left child nodes of U_m;
            let U_r denote the one of the right child nodes of U_m;
            if Aff_N(U_n) ⊇ Aff_N(U_r) and Aff_N(U_n) ⊇ Aff_N(U_l) then
                U_n is set as the only child node of U_m;
                U_r, U_l is set as the right, left child node of U_n;
            else
                if Aff_N(U_n) ⊇ Aff_N(U_r) and Aff_N(U_n) ⊉ Aff_N(U_l) then
                    U_i is set as the right child of U_m;
                    U_r is set as the only child node of U_n;
                else
                    if U_r is not the leaf node in EH-Tree and
                      Aff_N(U_n) ⊂ Aff_N(U_r) then
                        INSERT(U_n, U_r);
                    if U_l is not the leaf node in EH-Tree and
                      Aff_N(U_n) ⊂ Aff_N(U_l) then
                        INSERT(U_n, U_l);
                if Aff_N(U_n) ⊉ Aff_N(U_r) and Aff_N(U_n) ⊇ Aff_N(U_l) then
                    U_n is set as the left child node of U_m;
                    U_l is set as the only child node of U_n;
                else
                    if U_r is not the leaf node in EH-Tree and
                      Aff_N(U_n) ⊂ Aff_N(U_r) then
                        INSERT(U_n, U_r);
                    if U_l is not the leaf node in EH-Tree and
                      Aff_N(U_n) ⊂ Aff_N(U_l) then
                        INSERT(U_n, U_l);
return the updated EH-Tree;
```

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 *Section 5.4*.

Example 13: Recall the case shown in Fig. 2, and suppose there are four updates between the two queries (i.e., U_a, U_b, U_c and U_d in Example 10). The EH-Tree of these four updates is shown in Fig. 5(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.

5.4 Incremental GPNM Procedure

For each update, based on $SLen$ and $SLen_{new}$, we first build a set, called AFF, to record the changes of the shortest path length between each pair of nodes. Let $AFF[u_i, v_j] = [a, b]$ denote that the shortest path length from u_i to v_j is changed from a to b. Based on AFF, we can investigate whether the updates of G_D have an influence on the initial query results $IQuery$. If there is no influence, then the subsequent query results $SQuery$ are the same as $IQuery$. Otherwise, two methods $DMatch^+$ and $DMatch^-$ are devised to identify the affected parts of G_D and deliver $SQuery$ 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 $DMatch^+$ and $DMatch^-$ are discussed below.

DMatch⁺: If $v \in IQuery$, 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 remain 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 $SQuery$. The pseudo-code of *DMatch*$^+$ is shown in *Algorithm 6* 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 6*);
- **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 6*);
- **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 $u_j \notin IQuery$ 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 6*);
- **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 $SQuery$ (*lines 17-19 in Algorithm 6*).

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 $IQuery$, i.e., $SQuery$ = $IQuery$. 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 $\triangle G^+_{D_E}$.

DMatch⁻: If $v \notin IQuery$, 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 remain unchanged or increase, thus, v still does not satisfy the constraints. Therefore, $DMatch^-$ investigates the affected nodes in $IQuery$ and removes the nodes that do not satisfy the constraints of the pattern graph. The pseudo-code of *DMatch*$^-$ is shown in *Algorithm 7* and explained below.

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

- **Step 1:** For each pair of node (u_i, v_j) in AFF, if $(u_i, v_j) \in IQuery$ and $SLen_{new}(u_i, v_j) > f_e(u, v)$,

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2019.2942294, IEEE Transactions on Knowledge and Data Engineering

Algorithm 5: 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** $SQuery$;

Algorithm 6: DMatch$^+$

Input: $G_P, G_D, IQuery, e(u,v) \in \triangle G^+_{D_E}, SLen_{new}, AFF$
Output: $SQuery$
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** $(v,u) \in G_P$ **then**
10 **for** *each* $u_j \notin IQuery$ **do**
11 **if** $SLen_{new}(v_i, u_j) \leq f_e(v,u)$ **then**
12 Add the u_j into $AddSet()$;
13 **if** $(u,v) \in G_P$ **then**
14 **for** *each* $u_j \notin IQuery$ **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** $SQuery = IQuery \cup Addset()$;

then (a) when there is no other node v_n in $IQuery$ 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 $IQuery$ such that $SLen_{new}(u_n, v_j) \leq f_e(u,v)$, we add u_i into $DeleteSet()$ (*lines 2-9 in Algorithm 7*);

- **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 $SLen_{new}(u_j, v_i) \leq f_e(u,v)$, we add u_j into $DeleteSet()$ (*lines 10-14 in Algorithm 7*);
- **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 $SQuery$ *(lines 15-17 in Algorithm 7)*.

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 $\triangle G^-_{D_E}$.

Example 14: Recall Example 13. After searching the EH-Tree, we only need to apply the incremental GPNM procedure for U_a. Firstly, according to $SLen_{new}$ with U_a shown in Table 5, we have $AFF[Adam, Bella]$=[2, ∞], $AFF[Adam, Charles]$=[1, ∞], $AFF[Adam, David]$=[2, ∞], $AFF[Adam, Ervin]$=[2, ∞], $AFF[Adam, Fiona]$=[3, ∞] and $AFF[Adam, Green]$=[3, ∞]. Since $Charles$ and $Ervin$ are in $IQuery$ and $SLen_{new}(Adam, Charles) = \infty$,

Algorithm 7: DMatch$^-$

Input: $G_P, G_D, IQuery, e(u,v) \in \triangle G^-_{D_E}, SLen_{new}, AFF$
Output: $SQuery$
1 Set $DeleteSet() = \emptyset$;
2 **for** *each pair of node* $(u_i, v_j) \in AFF$ **do**
3 **if** $(u_i, v_j) \in IQuery$ *and* $SLen_{new}(u_i, v_j) > f_e(u,v)$ **then**
4 **for** *each* $v_n \in IQuery (n \neq j)$ **do**
5 **if** *There is no* v_n *such that* $SLen_{new}(u_i, v_n) \leq f_e(u,v)$ **then**
6 Add u_i into $DeleteSet()$;
7 **for** *each* $u_n \in IQuery (n \neq i)$ **do**
8 **if** *There is no* u_n *such that* $SLen_{new}(u_n, v_j) \leq f_e(u,v)$ **then**
9 Add v_j into $DeleteSet()$;
10 **for** *each node* $v_i \in DeleteSet()$ **do**
11 **if** $(v,u) \in G_P$ *and* v_i *is the only node such that* $SLen_{new}(v_i, u_j) \leq f_e(v,u)$ **then**
12 Add u_j into $DeleteSet()$;
13 **if** $(u,v) \in G_P$ *and* v_i *is the only node such that* $SLen_{new}(u_j, v_i) \leq f_e(u,v)$ **then**
14 Add u_j into $DeleteSet()$;
15 **if** *There is no newly added node in* $DeleteSet()$ **then**
16 Break;
17 **return** $SQuery = IQuery \setminus DeleteSet()$;

$SLen_{new}(Adam, Ervin) = \infty$, which means $Adam$ can not connect with $Charles$ or $Ervin$ any more, then $Adam$ and $Ervin$ are added into $DeleteSet()$. Therefore, $SQuery$=$IQuery \setminus DeleteSet()$=$\emptyset$.

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 [37]. 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 $|N_D|$.

6 Experiments

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

6.1 Experimental Setting

Datasets: We used five real-world social graphs that are available at *snap.stanford.edu*. The details are shown in Table 12.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2019.2942294, IEEE Transactions on Knowledge and Data Engineering

TABLE 12: The sizes of datasets

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 GPM [4], [21] and GPNM [11], [13] 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*[3], 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 [4], they are set between 6 and 10. Since the bounded path length on each edge is usually a small integer [4], 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 [11], 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 [13], 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.

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.

3. https://socnetv.org/

6.2 Experimental Results and Analysis

Figs. 6 to 10 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 13, and the comparisons between the methods are shown in Table 14. 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.

TABLE 13: The average query processing time based on different datasets

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	2,359.09s	2,628.49s	3,002.90s	4,749.91s
Average	707.59s	787.95s	904.99s	1,450.94s

TABLE 14: Comparison with TopKDAG, INC-GPNM and NEH-GPNM based on different datasets

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**

Analysis-1: As discussed in *Section 1.2*, 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 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 15, and the comparisons between the methods are shown in Table 16.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2019.2942294, IEEE Transactions on Knowledge and Data Engineering

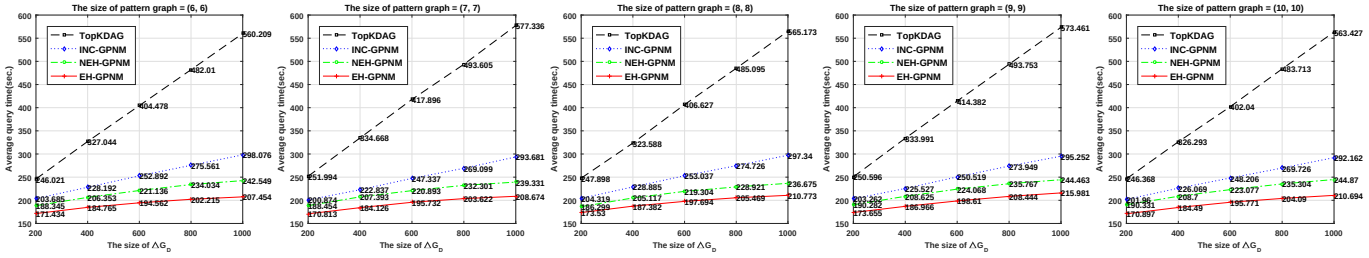


Fig. 6: The average query processing time in *Ask Ubuntu*

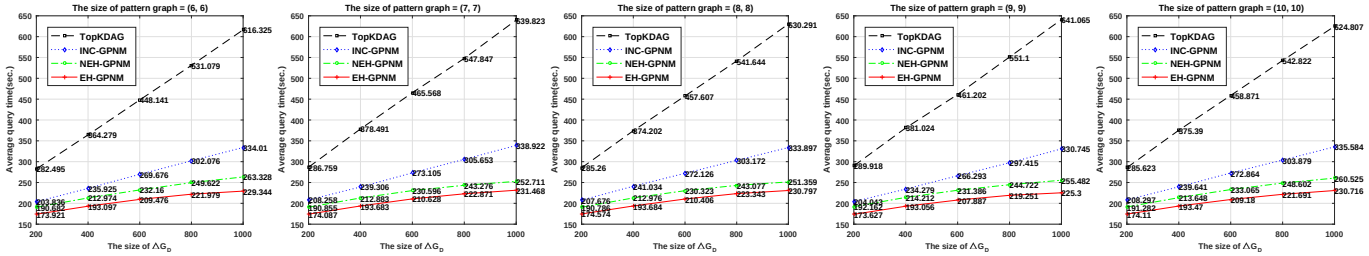


Fig. 7: The average query processing time in *Facebook*

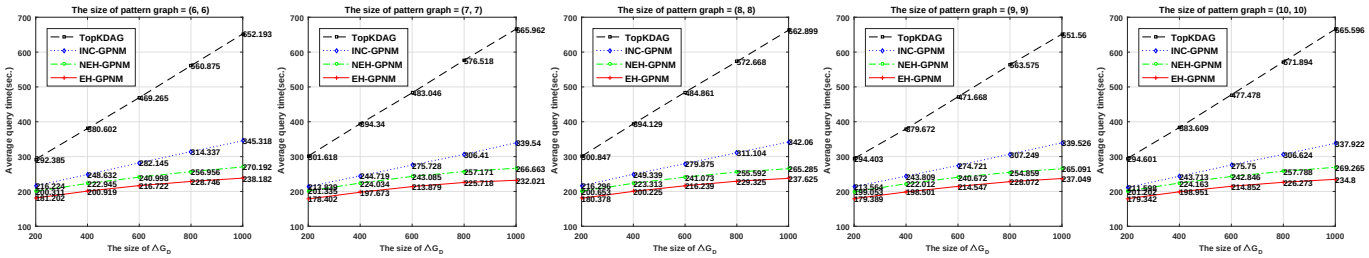


Fig. 8: The average query processing time in *Super User*

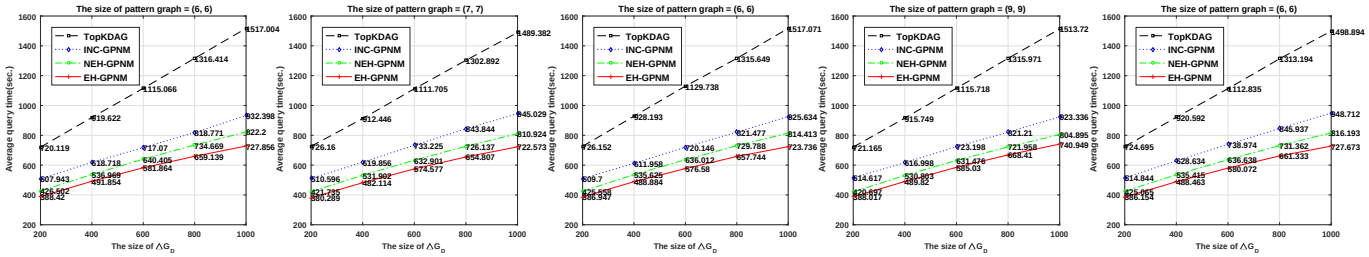


Fig. 9: The average query processing time in *Wiki Talk*

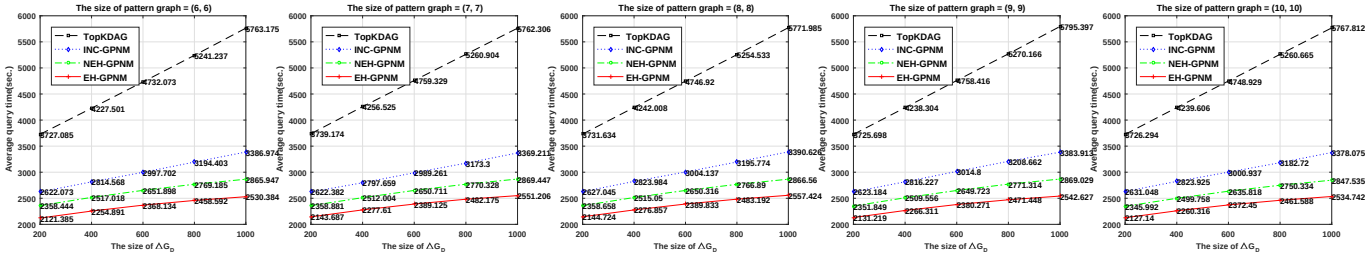


Fig. 10: The average query processing time in *LiveJournal*

TABLE 15: The average query processing time based on different scales of $\triangle G_D$

Scale of $\triangle G_D$	EH-GPNM	NEH-GPNM	INC-GPNM	TopKDAG
200	609.09s	671.82s	752.05s	1,056.99s
400	666.72s	737.74s	828.98s	1,250.07s
600	716.16s	795.62s	905.35s	1,442.15s
800	757.18s	845.99s	981.08s	1,633.99s
1000	789.60s	888.59s	1,057.52s	1,827.47s

TABLE 16: Comparison with TopKDAG, INC-GPNM and NEH-GPNM based on different scales of $\triangle G_D$

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**

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 $\triangle G_D$, 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.

7 Conclusion and Future Work

In this paper, we have proposed a GPNM method called EH-GPNM considering multiple updates in data graphs. EH-GPNM can efficiently deliver node matching results, and can reduce the query processing time. 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 on space complexity by designing new index structures, and (2) a new approach to selecting the top-k matching nodes.

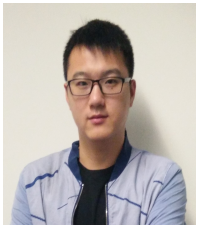
Acknowledgement

This work was supported by Australian Research Council Discovery Project DP180102378.

References

[1] J. R. Ullmann, "An algorithm for subgraph isomorphism," *JACM*, vol. 23, no. 1, pp. 31–42, 1976.
[2] 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*, 2001, pp. 149–159.
[3] M. R. Garey and D. S. Johnson, *Computers and Intractability*. wh freeman New York, 2002, vol. 29.
[4] W. Fan, J. Li, S. Ma, N. Tang, Y. Wu, and Y. Wu, "Graph pattern matching: from intractable to polynomial time," *PVLDB*, vol. 3, no. 1-2, pp. 264–275, 2010.
[5] S. Milgram, "The small world problem," *Psychology today*, vol. 2, no. 1, pp. 60–67, 1967.
[6] T. Lappas, K. Liu, and E. Terzi, "Finding a team of experts in social networks," in *SIGKDD*, 2009, pp. 467–476.
[7] 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 *SIGCHI*, 2010, pp. 1739–1748.
[8] J. Brynielsson, J. Högberg, L. Kaati, C. Mårtenson, and P. Svenson, "Detecting social positions using simulation," in *ASONAM*, 2010, pp. 48–55.
[9] Z. Liu and Y. Chen, "Identifying meaningful return information for xml keyword search," in *SIGMOD*, 2007, pp. 329–340.
[10] W. Fan, J. Li, S. Ma, N. Tang, and Y. Wu, "Adding regular expressions to graph reachability and pattern queries," in *ICDE*, 2011, pp. 39–50.
[11] W. Fan, X. Wang, and Y. Wu, "Diversified graph pattern matching," *PVLDB*, vol. 6, no. 13, pp. 1510–1521, 2013.
[12] T. Y. Berger-Wolf and J. Saia, "A framework for analysis of dynamic social networks," in *SIGKDD*, 2006, pp. 523–528.
[13] G. Sun, G. Liu, Y. Wang, M. A. Orgun, and X. Zhou, "Incremental graph pattern based node matching," in *ICDE*, 2018, pp. 281–292.
[14] X. Ren and J. Wang, "Multi-query optimization for subgraph isomorphism search," *PVLDB*, vol. 10, no. 3, pp. 121–132, 2016.
[15] 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*. ACM, 2014, pp. 519–528.
[16] Y. Yuan, G. Wang, J. Y. Xu, and L. Chen, "Efficient distributed subgraph similarity matching," *VLDB Journal*, vol. 24, no. 3, pp. 369–394, 2015.
[17] Y. Yuan, G. Wang, L. Chen, and B. Ning, "Efficient pattern matching on big uncertain graphs," *Information Sciences*, vol. 339, pp. 369–394, 2016.
[18] B. Lyu, L. Qin, X. Lin, L. Chang, and J. X. Yu, "Scalable supergraph search in large graph databases," in *ICDE*, 2016, pp. 157–168.
[19] Y. Fang, R. Cheng, S. Luo, and J. Hu, "Effective community search for large attributed graphs," *PVLDB*, vol. 9, no. 12, pp. 1233–1244, 2016.
[20] L. Lai, L. Qin, X. Lin, and L. Chang, "Scalable subgraph enumeration in mapreduce: a cost-oriented approach," *PVLDB*, vol. 26, no. 3, pp. 421–446, 2017.
[21] W. Fan, X. Wang, and Y. Wu, "Incremental graph pattern matching," *TODS*, vol. 38, no. 3, p. 18, 2013.
[22] K. Semertzidis and E. Pitoura, "Durable graph pattern queries on historical graphs," in *ICDE*, 2016, pp. 541–552.
[23] S. Sun, Y. Wang, W. Liao, and W. Wang, "Mining maximal cliques on dynamic graphs efficiently by local strategies," in *ICDE*, 2017, pp. 115–118.
[24] W. Fan, C. Hu, and C. Tian, "Incremental graph computations: Doable and undoable," in *SIGMOD*, 2017, pp. 155–169.
[25] S. Ma, R. Hu, L. Wang, X. Lin, and J. Huai, "Fast computation of dense temporal subgraphs," in *ICDE*, 2017, pp. 361–372.
[26] 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 *SIGMOD*. ACM, 2018, pp. 411–426.
[27] R.-H. Li, J. Su, L. Qin, J. X. Yu, and Q. Dai, "Persistent community search in temporal networks," in *ICDE*. IEEE, 2018, pp. 797–808.
[28] 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 *ICDE*. IEEE, 2018, pp. 245–256.
[29] L. Zou, L. Chen, and M. T. Özsu, "Distance-join: Pattern match query in a large graph database," *PVLDB*, vol. 2, no. 1, pp. 886–897, 2009.
[30] A. Marian, S. Amer-Yahia, N. Koudas, and D. Srivastava, "Adaptive processing of top-k queries in xml," in *ICDE*, 2005, pp. 162–173.
[31] H. Tong, C. Faloutsos, B. Gallagher, and T. Eliassi-Rad, "Fast best-effort pattern matching in large attributed graphs," in *SIGKDD*, 2007, pp. 737–746.
[32] W. Fan, Y. Wu, and J. Xu, "Adding counting quantifiers to graph patterns," in *SIGMOD*, 2016, pp. 1215–1230.
[33] T. Lappas, K. Liu, and E. Terzi, "A survey of algorithms and systems for expert location in social networks," in *Social Network Data Analytics*. Springer, 2011, pp. 215–241.
[34] G. Ramalingam and T. Reps, "An incremental algorithm for a generalization of the shortest-path problem," *Journal of Algorithms*, vol. 21, no. 2, pp. 267–305, 1996.
[35] G. Ramalingam and T. Raps, "On the computational complexity of dynamic graph problems," *Theoretical Computer Science*, vol. 158, no. 1-2, pp. 233–277, 1996.
[36] G. M. Adelson-Velskii and E. M. Landis, "An information organization algorithm," in *Doklady Akademia Nauk SSSR*, vol. 146, 1962, pp. 263–266.
[37] 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. ACM, 2009, pp. 18–28.



Guohao Sun received the BS and MS degrees from Soochow University, P.R. China in 2013 and 2015 respectively. He is currently working toward the PhD degree in Department of Computing, Macquarie University, Sydney, Australia. His current research interests include graph mining and social computing.

Guanfeng Liu is currently a Lecturer in the Department of Computing, Macquarie University, Sydney, Australia. He received his PhD degree in Computer Science from Macquarie University in 2013. His research interests include graph database, trust computing, and social computing.

Yan Wang received the BEng, MEng, and the DEng degrees in computer science and technology from Harbin Institute of Technology (HIT), P. R. China, in 1988, 1991, and 1996, respectively. He is currently a professor in the Department of Computing, Macquarie University, Sydney, Australia. His research interests include trust computing, social computing, service computing and recommender systems. He is a senior member of the IEEE.

Mehmet A. Orgun received the BSc and MSc degrees in computer science and engineering from Hacettepe University, Ankara, Turkey in 1982 and 1985, respectively; and the PhD degree from the University of Victoria, Canada, in 1991. He is currently a professor at Macquarie University, Sydney, Australia. His research interests include knowledge discovery, multiagent systems, trusted systems and temporal reasoning. He is a senior member of the IEEE.

Quan Z. Sheng received the PhD degree in computer science from the University of New South Wales, Sydney, Australia, in 2006. He is a professor and head of Department of Computing, Macquarie University. His research interests include Internet of Things, big data analytics, distributed computing, and pervasive computing. He is the recipient of ARC Future Fellowship in 2014, Chris Wallace Award for Outstanding Research Contribution in 2012, and Microsoft Research Fellowship in 2003. He is the author of more than 350 publications. He is a member of the ACM and the IEEE.

Xiaofang Zhou received the BSc and MSc degrees in computer science from Nanjing University, China, and the PhD degree in computer science from the University of Queensland, Australia, in 1984, 1987, and 1994, respectively. He is a professor of computer science with the University of Queensland and adjunct professor in the School of Computer Science and Technology, Soochow University, China. His research interests include spatial and multimedia databases, high performance query processing, web information systems, data mining, bioinformatics, and e-research. He is a fellow of the IEEE.