PatternTree$_{\text{ISO}}$: A Pattern Graph Correlation Framework for Accelerating Subgraph Isomorphism over Massive Graphs

Meizi Zhou[*]
Institute of Information Engineering
Chinese Academy of Sciences
zhoumeizi@iie.ac.cn

Jing Yu
Institute of Information Engineering
Chinese Academy of Sciences
yujing02@iie.ac.cn

Yanbing Liu
Institute of Information Engineering
Chinese Academy of Sciences
liuyanbing@iie.ac.cn

Qiong Dai
Institute of Information Engineering
Chinese Academy of Sciences
daiqiong@iie.ac.cn

Li Guo
Institute of Information Engineering
Chinese Academy of Sciences
guoli@iie.ac.cn

ABSTRACT

Subgraph isomorphism is a fundamental problem in graph analysis and becomes challenging in terms of querying patterns in increasingly massive network graphs. Algorithms for accelerating subgraph isomorphism have been discussed using optimized pruning rules, data graph index, etc. However, almost all the previous study processes each pattern independently and ignores the correlation between different patterns. In most graph matching problems, patterns always show up as a set and pattern containment relationship exists. To make use of such pattern correlations, we propose a genetic framework jointly modeling all the graphs of a pattern set to accelerate subgraph isomorphism. An optimal pattern containment structure is firstly studied to model the patterns in a level-wise matching order. Based on this structure, we present an efficient mechanism of reusing the matching results of a pattern to answer some other patterns. Experimental results on both real-world and synthetic datasets show that, integrated with our framework, the existing subgraph isomorphism algorithms gain significant speedup, one order of magnitude in the best case. This framework can be potentially applied to any other existing subgraph isomorphism algorithms to gain performance benefits.

Keywords

Subgraph isomorphism, pattern containment correlation, Pattern Tree

1. INTRODUCTION

[*]University of Chinese Academy of Sciences

CIKM '16 Oct 24–28, 2016, Indianapolis, United States
© 2016 ACM. ISBN 123-4567-24-567/08/06...$15.00
DOI: 10.475/123_4

Graphs are playing an increasingly important role in modeling complicated structured data in various domains. Among graph-related problems, *subgraph isomorphism search* is one of the most fundamental ones. It aims to find all embeddings of a smaller graph, usually called pattern graph or pattern, in a bigger graph, usually called data graph. This is a NP-Complete problem and various algorithms have been discussed to speed up its performance [14]. Most existing algorithms are based on a backtracking strategy which finds embeddings by incrementally matching candidates for each vertex in the pattern graph and stopping tracking when it predicts that no solution will be found. These algorithms mainly focus on optimizing pruning rules, matching orders, etc, and they process each pattern graph completely independently.

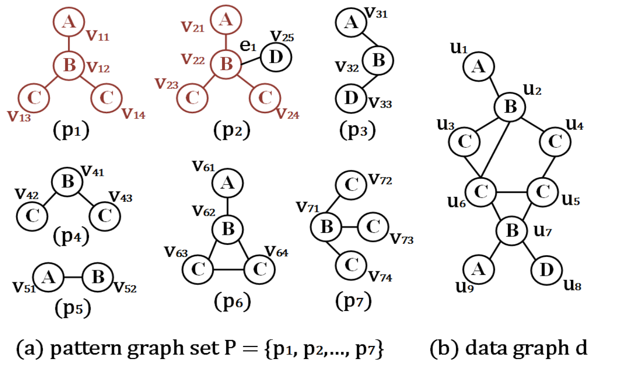


Figure 1: Examples of patterns, pattern containment correlation and a data graph.

In real-world applications, such as biological data analysis [1], recommendation network[9], social relation search [6], etc, numbers of pattern graphs are generally required to be processed at a time. In these pattern graphs, identical (sub)structures may exist between different patterns, which will cause redundant computation cost if matching each pattern independently. As shown in Figure 1, (a) and (b) respectively represent a set of patterns $P = \{p_1, p_2, ..., p_7\}$ and a data graph d. To search all the embeddings of P in d, existing algorithms searches p_1, p_2, ..., p_7 in d sequentially. However, we observe that p_1 is a subgraph of p_2 and that the answers of p_1 is contained in the answers of p_2. Intuitively, if the answers of p_1 is stored and used to answer p_2, the average response time will be reduced. Motivated by this idea, we aim to speed up the matching performance by jointly modeling all the incoming patterns provided that there popularly exist pattern containment relations. Therefore, the main challenge is to explore an effective method to tackle the problem of pattern containment.

This paper mainly focuses on studying the pattern containment correlation. It generally includes two subproblems: (1) given all the patterns, how to mine and store the pattern containment correlation so that it can minimize the total matching cost; (2) given the correlation between two patterns that one is contained in another, how to efficiently answer the bigger pattern by reusing the matched results of the smaller pattern. The goal is to answer more patterns with less referring to the data graph.

To solve the above problems, we propose a novel subgraph isomorphism framework, called PatternTreeISO. In this framework, we build a *Pattern Tree* structure from a given pattern set and make use of the pattern containment correlation recorded in *Pattern Tree* to accelerate subgraph isomorphism search. Main contributions of this paper are summarized below.

- We propose an innovative definition *Pattern Tree* and formulate how pattern graphs can be transformed into a *Pattern Tree* based on their containment correlation, and why the *Pattern Tree* can be utilized to accelerate subgraph isomorphism search.
- We propose a novel subgraph isomorphism framework, PatternTreeISO, which processes pattern graphs on a constructed *Pattern Tree* in a depth-first searching order and avoids redundant computation. It's a generic framework that can be integrated with any existing subgraph isomorphism algorithm.
- We conduct extensive experiments on both real-world and synthetic datasets. And the results show that the speed of existing subgraph isomorphism algorithms can be significantly improved by up to one order of magnitude in the best case.

The rest of this paper is organized as following. Section 2 surveys related work in this area. Section 3 defines some preliminary concepts and a query time cost model. Section 4 introduces the definition and strategies of building an optimal *Pattern Tree*. The matching framework of PatternTreeISO is introduced in Section 5. The performance study is reported in Section 6. Section 7 concludes our research and summarizes the future work.

2. RELATED WORK

Subgraph Isomorphism Algorithms. Subgraph isomorphism is still a challenging problem and lots of algorithms have been proposed. Most of them are based on a backtracking method and extend this method according to their optimizing specifics, including matching orders, pruning rules, and auxiliary information constraint. Representative works are briefly surveyed as following.

Ullmann [13] is the first practical algorithm based on backtracking strategy. It chooses a random matching order and prunes out a pattern vertex's candidate vertices with smaller degree than it. VF2 [4] selects pattern vertices connected to matched ones in a matching order and introduces 1-step-neighbor and 2-step-neighbor constraint to stop unpromising matching. Based on global vertex label frequencies, QuickSI [11] proposes a matching sequence by the order of vertices inserted into a pre-defined minimum spanning tree and prunes out candidates not satisfying parent-child relationship. GraphQL [8] proposes neighborhood signature and pseudo subgraph isomorphism test to prune out false candidates. Instead of matching one vertex at a time, SPath [16] matches a path each time and defines neighborhood signature to index and reduce candidates. Recently, TurboISO [7] shows great improvements over all competitors by several orders of magnitude. It exploits duplicate structures in a pattern graph and matches vertices with identical neighborhoods each time.

Different from existing algorithms, we focus on exploring the containment correlation between different pattern graphs and utilizing them to reduce duplicate computation and to accelerate searching performance. It's important that our method is not a single algorithm. It is a framework that can be integrated with any existing subgraph isomorphism algorithm and speed up its performance. To the best of our knowledge, this idea has not been studied in previous work.

Index-Based Matching Algorithms. For processing pattern graphs over large scale datasets, index-based work follows the framework of *filtering* and *verification*. Many effective graph indices have been proposed to filter false candidate graphs and to reduce the workload of verification based on subgraph isomorphism. The index construction process is usually time-consuming, mining discriminant features over the whole dataset, such as paths [12], subgraphs [2, 14], trees [17], and other structures [15]. Though our work doesn't specially study index construction strategies, it inspires further research in two aspects: (1) the matching order defined by our framework allows posterior pattern to search on previous matching results, which naturally achieves the indexing effect without index construction cost; (2) our work can be integrated with any index-based approach in filtering-and-verification framework.

3. PRELIMINARIES

3.1 Problem Statement

A graph is called a *simple graph* if it doesn't have multiple edges nor loops [5]. An *undirected labeled simple graph* is defined as a 4-tuple (V, E, L, l), where V is the vertex set, $E \subseteq V \times V$ is a set of undirected edges, and l is the labeling function that assigns labels belonging to the label set L to vertices and edges.

Subgraph Isomorphism. Given two graphs $p = (V'$, E', L', $l')$ and $d = (V, E, L, l)$, p is subgraph isomorphic to d (denoted as $p \subseteq d$) if there exists an bijective function f: (1) $\forall v \in V'$, $f(v) \in V$ such that $l'(v) = l(f(v))$; (2) $\forall (u, v) \in E'$, $(f(u), f(v)) \in E$ such that $l'(u, v) = l(f(u), f(v))$. Subgraph isomorphism problem is defined as, given a *pattern graph* (*query*) p and a *data graph* d, find all distinct embeddings of p in d.

Graph Pattern Matching. Given a data graph set $D =$

$\{g_1, g_2, ..., g_n\}$ and a pattern graph set $P = \{p_1, p_2, ..., p_m\}$, graph pattern matching is that for each $p_m \in P$, search for a set of graphs D_p where $D_p = \{g \,|\, g \in D \wedge p_m \subseteq g\}$.

In this paper, we only discuss undirected and labeled graphs, but our approach can be applied to directed graphs and unlabeled graphs as well. And in practice, we stop subgraph isomorphism search after finding the first 1000 embeddings, the same setting as mentioned in [7, 8, 16].

3.2 Query Cost Model

In the typical *filtering* and *verification* matching mechanism, the time cost of matching a pattern graph set in a data graph set can be modeled below:

$$T = |P| \times (T_{filter} + |C_p| \times T_{iso}) \qquad (1)$$

where P is the set of pattern graphs and T_{filter} is the average filtering cost based on indexing techniques. C_p is the filtered candidate data graph set and each graph in C_p should be verified by subgraph isomorphism methods. T_{iso} is the average subgraph isomorphism cost. Existing algorithms reduce T mainly in two aspects. One aspect is to apply effective filtering methods to minimize $|C_p|$ while keeping T_{filter} in a reasonable range, which requires index with high quality features. The other aspect is to optimize subgraph isomorphism algorithms to reduce T_{iso} by improving pruning rules, matching orders, and information constraint. In terms of T_{iso}, pattern graph size and data graph size affect its value. Suppose v_d and e_d are respectively the average vertex and edge numbers of each data graph, while v_p and e_p are respectively the average vertex and edge numbers of each pattern graph. Considering the subgraph isomorphism method without any pruning rules, the time complexity of T_{iso} can be modeled as:

$$O(v_d \times (\frac{e_d}{v_d})^{e_p+v_p-1}) \qquad (2)$$

In order to reduce T_{iso}, existing subgraph isomorphism methods try to reduce the number of candidate v_d and e_d.

Our proposed PatternTreeISO focuses on reducing the total number of v_p and e_p since T_{iso} increases exponentially over v_p and e_p. Meanwhile, PatternTreeISO can also reduce T_{filter} and $|C_p|$ without time-consuming index construction. These can be achieved by regarding the pattern set as a whole instead of independent ones. We would explain the time cost of PatternTreeISO after defining *Pattern Tree* in next section.

4. PATTERN TREE

In order to make use of containment correlation in a set of pattern graphs, we propose a method of organizing pattern graphs into a tree-structured order, called *Pattern Tree*, and apply strategies to optimize this tree for best performance. Some specific concepts related to *Pattern Tree* are defined below.

4.1 Pattern Tree Definitions

Definition 1. **Basic Pattern.** In a pattern set P, if $p \in P$ and $\nexists p' \in P$ that p' is subgraph isomorphic to p, p is called a basic pattern. Basic pattern set of P is denoted as P_{bas}.

Definition 2. **Extended Pattern.** In a pattern set P, if $p \in P$ and $\exists p' \in P$ that p' is subgraph isomorphic to p, p is called an extended pattern. Extended patterns set of P is denoted as P_{ext}.

Definition 3. **Residual Graph.** In a pattern set P, if $p_a \in P$, $p_b \in P$, and p_a is subgraph isomorphic to p_b, the residual graph of p_b corresponding to p_a is $res(p_b, p_a) = p_b - p_a$, which contains vertices and edges in p_b excluding those isomorphic corresponding to p_a.

Definition 4. **Pattern Tree.** *Pattern Tree* is a tree representing subgraph isomorphism relations in a pattern set. In a *Pattern Tree*, each node represents a pattern graph. If $p_a \in P$, $p_b \in P$ and p_a is subgraph isomorphism to p_b, an edge e_{ab} of the tree would direct from p_a to p_b. p_a is called a **parent pattern**, and p_b is called a **child pattern**. The weight of e_{ab} represents containment correlation degree (which will be explained in section 4.3). A virtual root node directs to all patterns in P_{bas}.

For example, Figure 2(d) is a *Pattern Tree* of the pattern set in Figure 2(a). p_0 is a virtual root. It directs to all basic patterns $P_{bas} = \{p_4, p_5\}$. The extended pattern set is $P_{ext} = \{p_1, p_2, p_3, p_6, p_7\}$. The residual graph of p_2 corresponding to p_1 is $res(p_2, p_1) = \{v_{25}, e_1\}$. The edge directs from p_5 to p_3 represents that p_5 is subgraph isomorphic to p_3. And p_5 is the parent pattern of p_3, while p_3 is the child pattern of p_5.

4.2 Query Time Saving by Containment Correlation Analysis

By applying *Pattern Tree* in subgraph isomorphism search, the time cost T in equation 1 of Section 3.2 can be saved in three aspects: (1) the average time of filtering $T_{filtering}$, (2) the average number of candidate data graphs for each pattern graph $|C_p|$, (3) the average subgraph isomorphism time cost T_{iso}.

$T_{filtering}$ and $|C_p|$ reduction. Given a pattern graph p and a data graph set D, for a basic pattern, $|C_p| = |D|$. But for an extended pattern, $|C_p| = |M|$, where M is the matched results of its parent pattern. M is quiet a small subset of the whole data graph set proved by experimental results showed in Section 6. Furthermore, a big part of the extended pattern can be directly matched to the embeddings in M and then continue to be searched from the neighbors of already matched vertices. This process helps filter candidate data graphs and candidate vertices while the filtering time cost $T_{filtering}$ is close to zero.

T_{iso} reduction. In PatternTreeISO, the vertex number to be matched is $V_{total} = |V(P_{bas})| + \sum_{p_i \in P_{ext}} |V(res(p_i, parent(p_i)))|$ and the edge number to be matched is $E_{total} = |E(P_{bas})| + \sum_{p_i \in P_{ext}} |E(res(p_i, parent(p_i)))|$, where $parent(p_i)$ indicates the parent pattern of p_i in the *Pattern Tree* derived from P. In practice, $V_{total} \leqslant |V(P)|$ and $E_{total} \leqslant |E(P)|$. The equal sign holds only when there is no subgraph isomorphism between pattern graphs. From equation 2 we observe that T_{iso} increases exponentially over v_p and e_p. *Pattern Tree* helps reduce V_{total} and E_{total}, and thus reduce T_{iso}.

From the above proof we can see that in a pattern graph set, the fewer basic patterns, V_{total} and E_{total}, the less query time cost. Therefore, we built *Pattern Tree* on three widely used graph sets, namely AIDS, NASA and Synthetic, and analyzed pattern containment correlation in them. Statistical analysis is shown in Section 6.

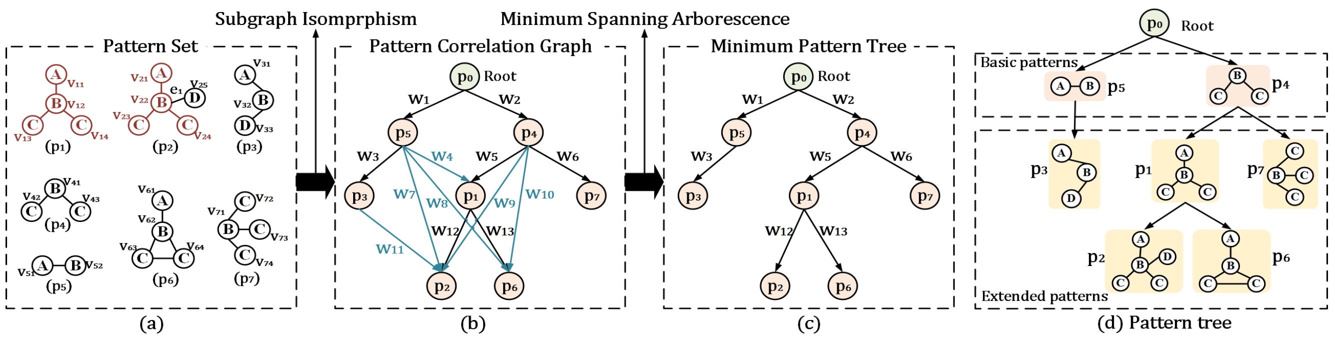


Figure 2: Pattern tree building process.

4.3 Pattern Tree Building

In this section, we describe our method of building the *Pattern Tree* in detail. The process contains two key steps: *Pattern Correlation Graph Building* and *Minimum Pattern Tree Building*.

(1) *Pattern Correlation Graph Building* : Between every two pattern graphs in a pattern set, we search for subgraph isomorphism relations and build a *pattern correlation graph* (*PCG*) to represent all isomorphic relations in the pattern set. As shown in Figure 2(b), *PCG* is a rooted and directed graph and records all isomorphic relations in Figure 2(a). The root of *PCG* is a virtual vertex and has edges pointing to all the basic graphs. Each edge directs from a parent pattern to a child pattern. The edge is weighted by an extra computing cost score $Score(e_{ij})$:

$$Score(e_{ij}) = |V(res(p_j, p_i))| + \partial \times |E(res(p_j, p_i))| \qquad (3)$$

where e_{ij} is an edge directing from p_i to p_j. $V(res(p_j, p_i))$ and $E(res(p_j, p_i))$ respectively represent the vertices and edges in $res(p_j, p_i)$. ∂ $(0 \leqslant \partial \leqslant 1)$ is a user-specified factor for weighting the computation cost of edges. Each score represents computation cost for processing the child pattern p_j based on the matched results of the parent pattern p_i.

(2) *Minimum Pattern Tree Building* : A *minimum pattern tree* (*MPT*) is derived from its *PCG* to get the optimal tree-structured order for best performance in graph pattern matching. We can see that in Figure 2(b) each child pattern may have multiple parent patterns and we only need one that leads to a minimum residual graph for the child pattern. In this way, it costs the least time when searching the child pattern, since there are minimum numbers of vertices and edges left to be searched. Thus the total cost for processing the whole pattern set is reduced to the minimum. Figure 2(c) is a *MPT* derived from Figure 2(b). The formal definition is given below.

Definition 5. **Minimum Pattern Tree.** Given a pattern correlation graph $g_{pattern} = (V, E)$, (p_i, p_j) is an edge in E directing from p_i to p_j. If there exists a subset $T \subseteq E$ while the graph $g'_{pattern} = (V, T)$ is a directed tree rooted at r and satisfying the following equation

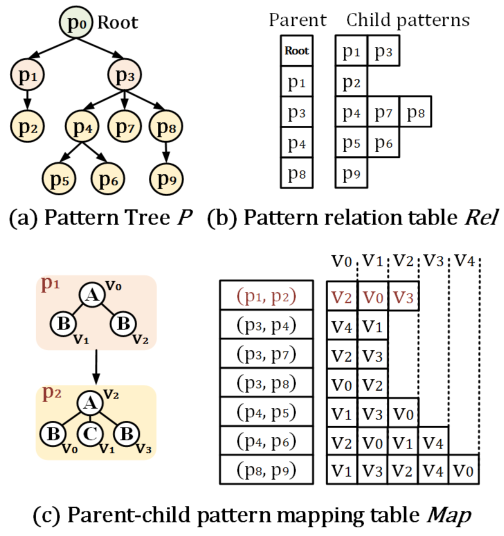
$$Score_{total}(g'_{pattern}) = \underset{T}{argmin} \sum_{(p_j, p_j) \in T} Score(p_i, p_j) \qquad (4)$$

$g'_{pattern}$ is called a *minimum pattern tree*.

Building *MPT* is a typical mathematical problem. The famous Chu-Liu algorithm [3] and the minimum spanning arborescence (MSA)[1] can solve it within $O(VE)$. In our problem, (1) there is no circuit in *PCG* and (2) there is no self-loop in each pattern vertex. According to the Chu-Liu algorithm, except the root vertex, the optimal solution for solving MSA is to select one incoming edge with the minimum weight and remove other edges for each vertex, which is pretty fast in practice. The computing procedure is detailed in Section 5.3. We will introduce how *Pattern Tree* is applied in PatternTreeISO framework for graph pattern matching in the following section.

5. FRAMEWORK OF PATTERNTREEISO

5.1 Data Structure


Root
(a) Pattern Tree *P*
(b) Pattern relation table *Rel*
(c) Parent-child pattern mapping table *Map*

Parent	Child patterns		
Root	p1	p3	
p1	p2		
p3	p4	p7	p8
p4	p5	p6	
p8	p9		

	V0	V1	V2	V3	V4
(p1, p2)	V2	V0	V3		
(p3, p4)	V4	V1			
(p3, p7)	V2	V3			
(p3, p8)	V0	V2			
(p4, p5)	V1	V3	V0		
(p4, p6)	V2	V0	V1	V4	
(p8, p9)	V1	V3	V2	V4	V0

Figure 3: Pattern Tree data structure

As introduced in Section 4, the most essential problem in our framework is to effectively build and utilize the *Pattern Tree*. In this section, we firstly present the data structures for storing the *Pattern Tree*. It mainly consists two data structures: (1) *pattern relation table Rel*. $Rel[a]=\{b \mid b = child(p_a),\ p_a \in P\}$, where $child(p_a)$ represents the child pattern set of p_a and used to index child patterns given a par-

[1]Introduction of minimum spanning arborescence: http://en.wikipedia.org/wiki/Minimum_spanning_tree

ent pattern. For example, Figure 3(a) is a *Pattern Tree* P. And Figure 3(b) is the *pattern relation table* of P; (2) *parent-child pattern mapping table* Map. $Map[c]$={$d \mid c = (p_i, p_j)$, $p_i \in P$, $p_j \in P$, $p_i \subseteq p_j$, d is one p_i-to-p_j vertex correspondence mapping}. For example, Figure 3(c) shows the *parent-child pattern mapping table* of P in Figure 3(a) and how the vertex corresponding relationship between p_1 and p_2 is stored in the *parent-child pattern mapping table*.

5.2 Overview Of PatternTreeISO

Algorithm 1 PatternTreeISO(D, P, ∂)

Input: Pattern graph set $P = \{p_1, ..., p_n\}$, data graph set $D = \{d_1, ..., d_m\}$, user-specified parameter for Pattern Tree building ∂
Output: All embeddings of P in D

```
1: $Rel$, $Map$ = BuildPatternTree($P, \partial$)
2: $basList \leftarrow \{b \mid a = root, b = Rel[a]\}$
3: for each $b$ in $basList$ do
4:     $ansBas \leftarrow$BasPatMatch($b, D$)
5:     $ansExt \leftarrow$ExtPatMatch($Rel$, $Map$, $b, ansBas$)
6:     $R \leftarrow R \bigcup ansBas \bigcup ansExt$
7: end for
8: return $R$
```

Algorithm 1 outlines the framework of PatternTreeISO. It consists two main steps: *Pattern Tree building* and *query processing*. The inputs are the pattern graph set, data graph set and an edge weight factor as we explained in Section 4.3. The outputs are all the embeddings of the pattern graphs in the data graphs. Firstly, a *Pattern Tree* is built according to *BuildPatternTree* (detailed in Algorithm 2) and is stored in the pattern relation table *Rel* and the parent-child pattern mapping table *Map* (line 1). From the pattern relation table *Rel*, we can find all the basic patterns and record them in *basList*. They are the children of the virtual root (line 2). In the query processing step, for each basic pattern b, we firstly perform *BasPatMatch* (detailed in Algorithm 3) to search b in the whole data graph set D and the results are stored in *basRes* (line 3-4). After we get the matched results *basRes*, *ExtPatMatch* (detailed in Algorithm 4) searches descendants of b based on *basRes* in a depth-first search order (line 5). Then, matched results of b and all descendants of b are stored successively (line 6-7).

5.3 Pattern Tree Building

The *Pattern Tree* building procedure is showed in Algorithm 2. The inputs are the pattern graph set and the edge weight factor introduced in Section 4.3. The outputs are a pattern relation table and a parent-child mapping table. We firstly build the *PCG* of the input pattern graphs (line 1-10) and store it in *allMap*. For each pair of patterns, we only need to find one vertex mapping by *GetOneIsoMap* (line 3). Any existing isomorphism algorithm can be applied here. Then we weight edges in the *PCG* by the cost score according to the equation 3 (line 11-15). Finally we compute the *MPT* from the weighted *PCG* (line 16-19). *MPT* stores the *Pattern Tree* and consists of the pattern relation table and the parent-child mapping table.

5.4 Basic Pattern Matching

Algorithm 2 BuildPatternTree(P, ∂)

Input: Pattern set P, edge weight ∂
Output: Pattern relation table Rel, parent-child mapping table Map

```
1: for each $p$ in $P$ do
2:     for each $q$ in $P$ where $|E(p)| \geqslant |E(q)| \, and \, |V(p)| \geqslant |V(q)|$ do
3:         $onemap$ = GetOneIsoMap($p$, $q$)
4:         if $onemap \neq \phi$ then
5:             $a \leftarrow a \bigcup \{q\}$
6:             $allMap[(q, p)] \leftarrow onemap$
7:         end if
8:     end for
9:     $H \leftarrow H \bigcup (p,\ a)$
10: end for
11: for each ($p$, $a$) in $H$ do
12:     for each $k$ in $a$ do
13:         $score \leftarrow |V(k)| + \partial \times |E(k)|$
14:         $S \leftarrow S \bigcup (k,\ score)$
15:     end for
16:     $finalParent \leftarrow k$ with minimum score and $(k,\ score) \in S$
17:     $Rel[finalParent] \leftarrow p$
18:     $Map[(finalParent,\ p)] \leftarrow allMap[(finalParent,\ p)]$
19: end for
20: return $relation$, $mapping$
```

Algorithm 3 BasPatMatch$(p,\ D)$

Input: Basic pattern graph p, data graph set D
Output: All embeddings of p in D $basRes$

```
1: for each $d$ in $D$ do
2:     $isomaps \leftarrow$GetIsoMaps($p$, $d$)
3:     $basRes \leftarrow \{basRes\} \bigcup (d,\ isomaps)$
4: end for
5: return $basRes$
```

Algorithm 3 describes the procedure of *BasPatMatch* function. It searches one basic pattern graph in all the data graphs. The inputs are a basic pattern graph and all the data graphs. The outputs are the matched results of this basic pattern graph. For each data graph, *GetIsoMaps* function aims to search embeddings of the basic pattern p in one data graph d (line 1-2). Any existing subgraph isomorphism algorithm can be applied as *GetIsoMaps* and get integrated in PatternTreeISO framework. In our experiments, we choose a classical algorithm VF2 and a state-of-the-art algorithm TurboISO respectively in *GetIsoMaps* to prove the effectiveness of our framework. The matched results are stored in *basRes* (line 3).

5.5 Extended Pattern Matching

Algorithm 4 describes the procedure of processing extended patterns. It searches the decedents of a parent pattern recursively. The inputs are the parent-child relation table, the parent-child mapping table, the parent pattern, and the matched results of the parent pattern. It outputs all the matched results of the basic pattern's descendants. In *ExtPatMatch*, it firstly searches the child patterns of *parent* in data graphs on the base of *parentRes* computed by function *MatchResidual* (line 1-6). *MatchResidual* searches *res*(*p*, *parent*) in the data graph d based on its parent's em-

Algorithm 4 ExtPatMatch(Rel, Map, $parent$, $parentRes$)

Input: Parent-child relation table Rel, parent-child mapping table Map, parent pattern $parent$, parent's matched result $parentRes$

Output: All embeddings of $parent$'s child patterns in D $extRes$

```
1: for each $p$ in $Rel[parent]$ do
2:   $piso \leftarrow \phi$
3:   for each ($d$, $embed$) in $parentRes$ do
4:     $ans \leftarrow$MatchResidual($p$, $d$, $embed$, $Map$)
5:     $piso \leftarrow piso \bigcup ans$
6:   end for
7:   $extRes \leftarrow extRes \bigcup piso$
8:   if $Rel[p] \neq \phi$ then
9:     $extRes \leftarrow extRes \bigcup$ExtPatMatch($Rel$, $Map$, $p$, $piso$)
10:  end if
11: end for
12: return $extRes$
```

beddings *embed*. The residual graph searching starts from the vertices in the data graph which has been matched by the parent pattern. The matched results of p is stored in *piso*. Then we treat p as a parent pattern and search its child patterns based on *piso* by $ExtPatMatch$ (line 9). This procedure is executive iteratively until all the basic pattern's descendants have been processed. All the results are stored in *extRes* (line 7-9).

6. EXPERIMENTS

The purpose of our experiments was to evaluate (1) the response time and space efficiency of building $Pattern\ Tree$, and (2) the improvement of PatternTreeiso framework compared with existing subgraph isomorphism algorithms. To test the efficiency of PatternTreeiso, we change the pattern graph number and compare the average elapsed time. To test the scalability of PatternTreeiso, we change the data graph size and analyze the speed improvement. The factor ∂ was set as different values in (0, 0.3, 0.5, 0.7, 1) to evaluate its influence on the matching speed.

6.1 Experiment Setup

Implementation and experiment environment. We integrated PatternTreeiso with 2 existing subgraph isomorphism algorithms, VF2 and Turboiso. VF2[4] is a classical algorithm which has been implemented as a baseline in many graph isomorphism work[7, 10]. Turboiso is a state-of-the-art algorithm and has been proved to outperform all competitors by up to several orders of magnitude[7]. All the algorithms were implemented in python 2.7 using iGraph package[2]. We implemented Turboiso based on [7] and implemented VF2 implementation based on [4]. All the experiments were conducted on a 64-bit Windows machine with Intel 3GHz CPU and 4GBytes RAM.

Measurements. **(1) Pattern Tree building analysis**. Firstly, we use the statistics of vertex and edge number to measure the computational reduction based on the $Pattern\ Tree$. The *original total vertices* and *original total edges* respectively correspond to $|V(P)|$ and $|E(P)|$ in Section 4.2, while the *pattern tree vertices* and *pattern tree edges* respectively correspond to V_{total} and E_{total} in Section 4.2. Then, we show the time cost for building the $Pattern\ Tree$ and the space cost for storing all the necessary data structures introduced in Section 5.1. **(2) Efficiency and scalability of query processing**. We measure the performance of an algorithm in two aspects: average elapsed time over increasing pattern set size for efficiency test, and average elapsed time over increasing data set size for scalability test. The average elapsed time for processing each pattern is $T = T_{total}/|P|$, where T_{total} includes the total elapsed time for both building $Pattern\ Tree$ and query processing while $|P|$ is the pattern set size. In practice, for one pattern and one data graph, we stop subgraph isomorphism searching on them once we find the first 1000 embeddings, as in [7, 8, 16].

Data graph sets. We use AIDS, NASA and Synthetic datasets for both efficiency tests and scalability tests. The three datasets have different characters which are showed in Table 1. (1) AIDS consists of 10,000 sparse graphs. For scalability tests, we generate five datasets with equivalent vertex increment by randomly selecting data graphs in AIDS. And the total vertex numbers of the five datasets are (50K, 100K, 150K, 200K, 250K). (2) NASA consists of 36,790 tree-structured graphs and the degree of some vertices is very big (up to 245). We generate five datasets in the same way as AIDS, and their sizes range in (50K, 100K, 150K, 200K, 250K) in terms of vertex number. (3) Synthetic contains five sets with different vertex numbers generated by **GraphGen**[3]. The vertex numbers of synthetic data graph sets are (70K, 140K, 210K, 280K, 350K).

Table 1: Profile of datasets

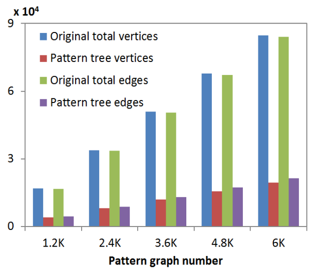
Datasets	$\|V\|$	$\|E\|$	$\|L\|$	Max degree
AIDS	254,156	274,513	51	11
NASA	1,223,194	1,186,404	117,302	245
Synthetic	1,036,868	2,747,163	50	12

Pattern graph sets. We generate pattern graphs by randomly selecting equal numbers and different sizes of pattern graphs from existing pattern sets of AIDS, NASA and Synthetic. Therefore, one pattern graph set consists of different sizes of pattern graphs which is just like what happens in real application scenarios. For example, AIDS contains six existing pattern sets in terms of edge numbers ranging from 4 to 24 (named respectively as $Q4$, $Q8$, ..., $Q24$). Each pattern set contains 1000 graphs. To generate a 1.2K AIDS pattern set, we randomly select 200 graphs from $Q4$, 200 graphs from $Q8$, ..., 200 graphs from $Q24$. Since every pattern set contains varies sizes of pattern graphs, we name them by their graph numbers, such as AIDS pattern sets (1.2K, 2.4K, 3.6K, 4.8K, 6K). Pattern sets of NASA and Synthetic are generated in the same way. The difference in Synthetic is that the pattern scales are (3.6K, 7.2K, 10.8K, 14.4K, 18K). A 3.6K Synthetic pattern set indicates that it selects 600 patterns from each of the pattern subset $Q4$, ..., $Q24$.

6.2 Experiment on AIDS dataset

[2]iGraph: a network analysis package can be downloaded for free from the link http://igraph.org/

[3]GraphGen: a synthetic graph data generator can be downloaded for free from the link http://www.ust.hk/graphgen/.



(a)Vertex and edge reduction

Pattern set	Time (min)	Space (KByte)
1.2K	0.4	52
2.4K	1.9	110
3.6K	4.3	168
4.8K	7.5	227
6K	11.6	286

(b)Building time and space

Figure 4: Pattern Tree building over AIDS

Data graphs in AIDS are ringlike and have small average and maximum degree. Figure 4 shows the experimental results of building the *Pattern Tree* over AIDS patterns. Figure 4(a) compares the number of vertices and edges that need to be searched based on the constructed *Pattern Tree* with that in the original pattern sets. It's obvious that V_{total} and E_{total} are greatly reduced over pattern sets of all the scales compared with $|V(P)|$ and $|E(P)|$. The reduction is nearly 80% on average. In other words, 80% overhead for matching original pattern vertices and edges are saved based on the *Pattern Tree* structure. Figure 4(b) shows the time and space cost of building the *Pattern Tree* over five pattern sets. Since the subgraph isomorphism test is conducted between each two pattern graphs in the pattern sets, the building time increases as the pattern set size increases. The longest time for processing 6K pattern set is 11.6 minutes, which is within acceptable scope. When tuning the parameter ∂, the building time fluctuates by up to 0.3 seconds. In other words, different value of ∂ has quite limited influence on the total building time. That's because the majority time cost is from computing PCG, while MPT computation only takes a few seconds, which is costless.

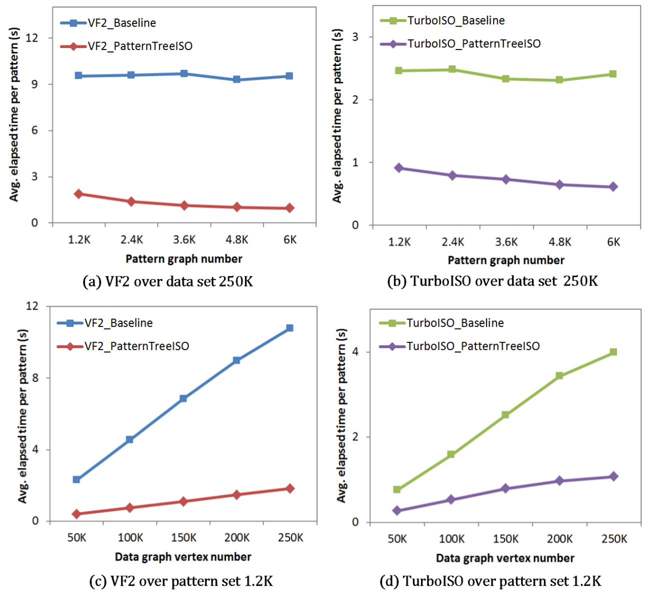


Figure 5: Query processing over AIDS.

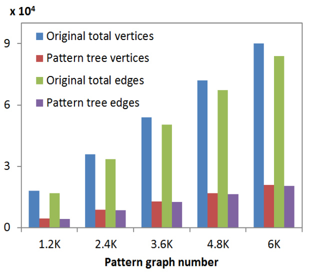
The experimental results of efficiency tests are shown in Figure 5(a)∼(b). We can see that the average elapsed time of VF2 and Turboiso is relatively steady when the pattern set size increases. With the integration of PatternTreeiso, all of VF2 and Turboiso are improved by several times. It's obvious that VF2+PatternTreeiso outperforms VF2 over every pattern set size and the average matching speed is 5∼10 times faster than VF2. Turboiso is the most state-of-the-art subgraph isomorphism algorithm and is nearly 3 times faster than VF2. However, with the integration of our framework, it is accelerated by 2.6∼3.9 times when the pattern set size varies. The improvement of VF2 and Turboiso when integrated into PatternTreeiso is more remarkable when increasing the pattern set size. The average elapsed time of VF2+PatternTreeiso decreases from 1.9 seconds to 0.9 seconds while the average elapsed time of Turboiso+PatternTreeiso decreases from 0.9 seconds to 0.6 seconds as pattern set size increases. That's because that the more patterns, the higher ratio of extended patterns in our containment correlation model as analyzed in section 4.2. And for extended patterns, only their residual graphs need to be processed which are much smaller than their original patterns.

The experimental results for scalability tests are shown in Figure 5(c)∼(d), which represent the average elapsed time for processing 1.2K patterns by varying the data set size. As the data set size grows, it's obvious that both VF2 and Turboiso achieve much better performance after integrated into PatternTreeiso. For small data set sizes, the improvement by PatternTreeiso is not that significant. But for large data set sizes, the performance can be six times faster. This is because for VF2 and Turboiso, all patterns need to be searched independently in the whole dataset while for PatternTreeiso framework, only basic patterns and residual graphs need to be searched. When the dataset size grows, the computation reduction based on the containment correlation is more notable. So this framework is greatly beneficial for searching patterns on large datasets.

In Figure 5, we only show the curves by setting the factor ∂ to 1.0 since other settings achieves tiny fluctuation. This phenomenon is due to the sparsity of the graphs in the pattern sets. For each graph, the number of edges is almost equal to the number of vertices. The pattern being a subgraph of an extended pattern with more vertices in the corresponding residual graph will also have more edges. The factor ∂ which highlights the influence of edges is useless. The following experiments on NASA and Synthetic datasets have similar phenomena and we also show the curves by setting the factor ∂ to 1.0.

6.3 Experiment on NASA dataset

NASA dataset is much bigger than AIDS and consists of star-shaped data graphs with a large number of vertex labels. Figure 6 shows the performance of constructing *Pattern Tree* over NASA. As shown in Figure 6(a), the vertex and edge needed to be searched are greatly reduced by applying *Pattern Tree*. On average, both vertices and edges are reduced by 75%, which means that repetitive computing in query processing can be avoided by up to 75%. Figure 6(b) shows the construction time and space over NASA. When varying the pattern set size, the time and space cost of building *Pattern Tree* is within a reasonable range. It takes 16.3 minutes at most when the pattern set size is 6K. The parameter ∂ also has limited influence on the building time



(a)Vertex and edge reduction

Pattern set	Time (min)	Space (KByte)
1.2K	0.6	55
2.4K	2.5	115
3.6K	5.9	176
4.8K	10.6	236
6K	16.3	299

(b)Building time and space

Figure 6: Pattern Tree building over NASA

over NASA. Building time changes within 2 seconds when tuning ∂.

Figure 7(a)∼(b) show the experimental results of efficiency tests. Figure 7(a) depicts the performance comparison of VF2 and VF2+PatternTreeISO. VF2 couldn't accomplish the searching tasks at any pattern set size within a week because of its relatively weak pruning power. VF2+PatternTreeISO speeds up significantly compared with VF2 and finishes processing in less than 40 seconds on average. The improvement is no less than 4 orders of magnitude. Moreover, the figure shows a trend that with the growth of pattern set sizes, the average elapsed time of VF2+PatternTreeISO decreases, which has the same phenomenon and reason as the performance over AIDS. For TurboISO in Figure 7(b), the improvement by integrated with PatternTreeISO is 4.4∼6 times faster at different pattern set sizes. The time cost of both TurboISO and TurboISO+PatternTreeISO fluctuates slightly. Comparatively, TurboISO+PatternTreeISO performs more stable than TurboISO. At the 6K pattern set, TurboISO+PatternTreeISO shows a tiny increase in time cost due to the prominent increased building time.

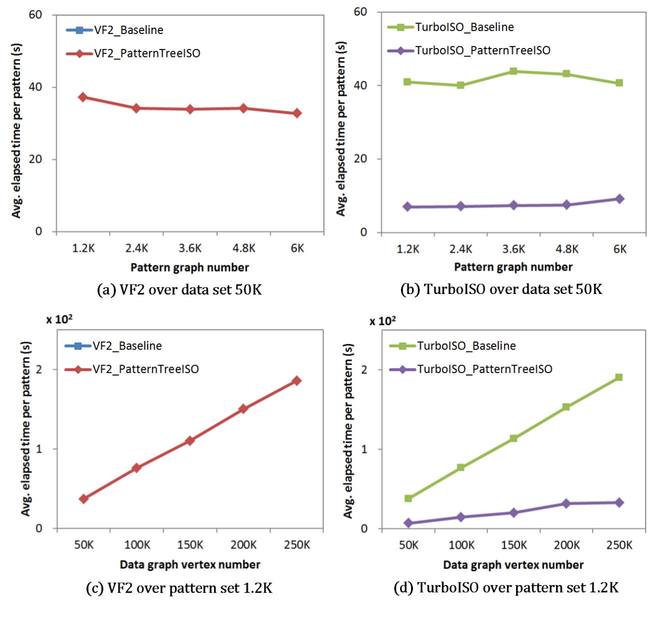
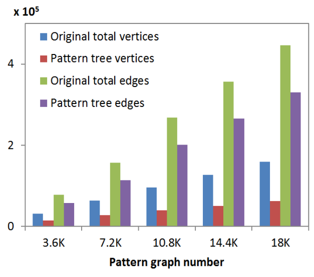


Figure 7: Query processing over NASA.

Figure 7(c)∼(d) show the experimental results of scalability tests. Since the pattern set remains the same, the *Pattern Tree* building time is equal on all the scalability tests. Therefore, the speed improvement of both algorithms integrated with PatternTreeISO becomes more significant with the increase of data set sizes. As shown in Figure 7(c), VF2 still performs too slow to be recorded. However, after integrated with PatternTreeISO, the tasks can be finished in a reasonable time. The time cost increases linearly with the grows of data set size. In Figure 7(d), the average elapsed time of TurboISO+PatternTreeISO outperforms TurboISO by 5.4∼5.9 times when the data set size varies. The curve of TurboISO increases sharply while the curve of TurboISO+PatternTreeISO has a sublinear increase. This is because that TurboISO processes patterns independently while TurboISO+PatternTreeISO avoids repetitive subgraph matching of patterns. The overhead reduction becomes particularly significant when the data graph size is large.

6.4 Experiment on Synthetic dataset

Synthetic is an automatically generated dataset whose graphs have fewer containment correlations than AIDS and NASA. We generated relatively bigger pattern sets over Synthetic than those of AIDS and NASA. Figure 8 shows the performance of building *Pattern Tree* over Synthetic pattern sets. In Figure 8(a), on average, V_{total} is 50% of $|V(P)|$ and E_{total} is 70% of $|E(P)|$ which indicates that in Synthetic pattern sets, nearly 50% vertices and 30% edges are repetitive. Figure 8(b) shows the cost of building *Pattern Tree* over Synthetic. All pattern subsets can be built within an acceptable time which is no more than one hour even for the biggest pattern set. The space cost for storing *Pattern Tree* is very little and increases in a linear trend with the growth of pattern set size. It shows good scalability in terms of time and space cost for building the *Pattern Tree*.



(a)Vertex and edge reduction

Pattern set	Time (min)	Space (KByte)
3.6K	2.3	97
7.2K	9.2	213
10.8K	20.7	332
14.4K	36.8	458
18K	57.6	586

(b)Building time and space

Figure 8: Pattern Tree building over Synthetic

Figure 9(a)∼(b) show the results of efficiency tests on Synthetic. The average elapsed time of all algorithms fluctuates in a small scale. VF2+PatternTreeISO outperforms VF2 on every pattern set. The average matching speed is two times faster than that of VF2. However, the average elapsed time of TurboISO changes quite differently from that of VF2 after integrated with PatternTreeISO. In Figure 9(b) for TurboISO, there are slight improvements when the pattern set size is less than 10.8K. However, when the pattern set size increases to 14.4K and larger, TurboISO+PatternTreeISO becomes slightly slower. This phenomenon is mainly due to two reasons. One is the fact that the containment correlations in Synthetic is less than that in AIDS and NASA, thus the speedup is not that significant. The other reason is that the pattern set size of Synthetic is much larger than that of AIDS and NASA, which costs more time in building

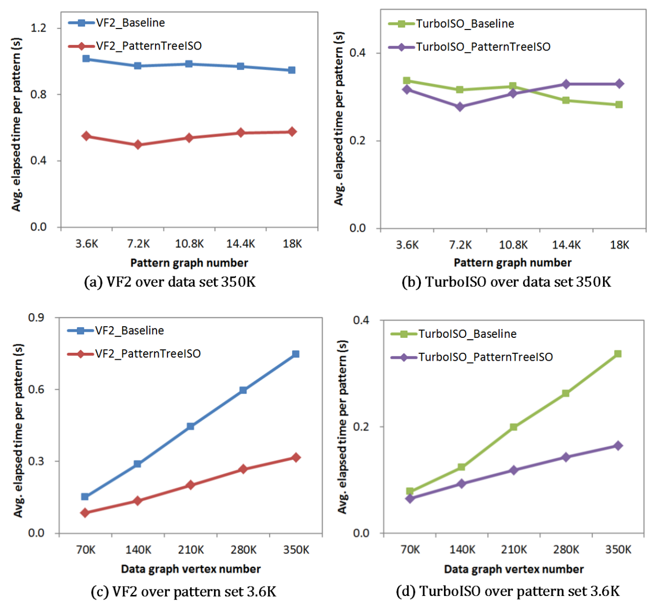


Figure 9: Query processing over Synthetic.

Pattern Tree. Since we include the initialization time when calculating the average elapsed time, the processing speed decreases when the pattern set size increases.

Figure 9(c)∼(d) are scalability experiments on Synthetic. For both VF2 and TurboISO, the speed improvement is obvious on all the tests. After integrated with PatternTreeISO, VF2 is speeded up by 2 times on average over all the datasets. For TurboISO, the improvement is slight when the data graph size is small while the advantage becomes more remarkable with the growth of the dataset, which is consistent with the performance of VF2. What's more, the average elapsed time of both VF2 and TurboISO increases much sharper than that of PatternTreeISO. It implies that the advantage of applying PatternTreeISO is especially remarkable when the data graph size is large.

7. CONCLUSIONS

In this paper, we propose a novel framework, PatternTreeISO, for accelerating subgraph isomorphism over massive graphs. The main contribution of our approach is that it makes best use of the containment correlations among pattern graphs and avoids massive duplicate computation. Any existing subgraph isomorphism algorithm can be integrated into this framework to speed up their matching performance. Extensive experiments on real and synthetic datasets prove that, by integrating with PatternTreeISO, both the typical subgraph isomorphism algorithm and the state-of-the-art one can be accelerated by up to several times.

PatternTreeISO is proved to be effective and efficient based on the hypothesis that pattern containment exists among pattern graphs. If such relation cannot be provided, auxiliary and concise graphs need to be adaptively constructed to bridge the pattern graphs and form the *Pattern Tree*. Besides, the incremental method for building the *Pattern Tree* is another important focus to avoid re-computing in the whole pattern graph set and reduce the response time when the pattern graphs update. These problems will be further studied in an extended version of this paper.

8. REFERENCES

[1] D. A. Bader and K. Madduri. A graph-theoretic analysis of the human protein-interaction network using multicore parallel algorithms. *Parallel Computing*, 34(11):627–639, 2008.

[2] J. Cheng, W. N. Yiping Ke, and A. Lu. Fg-index: Towards verification-free query query processing on graph databases. *SIGMOD*, pages 857–872, 2007.

[3] Y. Chu and T. Liu. On the shortest arborescence of a directed graph. *Scientia Sinica*, 14:1396–1400, 1965.

[4] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. A (sub)graph isomorphism algorithm for matching large graphs. *PAMI*, 26(10):1367–1372, 2004.

[5] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. *Introduction to algorithms.* MIT Press, Cambridge, 2001.

[6] W. Fan, X. Wang, and Y. Wu. Expfinder: Finding experts by graph pattern matching. *ICDE*, pages 1316–1319, 2013.

[7] W. S. Han, J. Lee, and J. H. Lee. Turboiso: Towards ultrafast and robust subgraph isomorphism search in large graph databases. *SIGMOD*, pages 337–348, 2013.

[8] H. He and A. K. Singh. Graphs-at-a-time: query language and access methods for graph databases. *SIGMOD*, pages 405–418, 2008.

[9] J. Leskovec, A. Singh, and J. Kleinberg. Patterns of influence in a recommendation network. *Springer Berlin Heidelberg*, 3918:380–389, 2010.

[10] X. Ren and J. Wang. Exploitng vertex relationships in speeding up subgraph isomorphism over large graphs. *Proceedings of the VLDB endowment*, 8(5):617–628, January 2015.

[11] H. Shang, Y. Zhang, X. Lin, and J. X. Yu. Taming verification hardness: an efficient algorithm for testing subgraph isomorphism. *PVLDB*, 1(1):364–375, 2008.

[12] D. Shasha, J. T. Wang, and R. Giugno. Algorithmics and applications of tree and graph searching. *SIGMOD*, pages 39–52, 2002.

[13] J. R. Ullmann. An algorithm for subgraph isomorphism. *JACM*, 23(1):31–42, 1976.

[14] J. Yu, Y. Liu, Y. Zhang, M. Liu, J. Tan, and L. Guo:. Survey on large-scale graph pattern matching. *Journal of Computer Research and Development*, 52(2):391–409, 2015.

[15] D. Yuan and P. Mitra. Lindex: A lattice-based index for graph databases. *VLDB Journal*, 22(2):229–252, 2013.

[16] P. Zhao and J. Han. On graph query optimization in large networks. *PVLDB*, 3(1):340–351, 2010.

[17] P. Zhao, J. X. Yu, and P. S. Yu. Graph indexing: Tree+delta>=graph. *VLDB*, pages 938–949, 2007.