Data & Knowledge Engineering 68 (2009) 1034–1058

Contents lists available at ScienceDirect

Data & Knowledge Engineering

journal homepage: www.elsevier.com/locate/datak

Mining globally distributed frequent subgraphs in a single labeled graph

Xing Jiang [a,*,1], Hui Xiong [b], Chen Wang [c], Ah-Hwee Tan [a]

[a] School of Computer Engineering, Nanyang Technological University, Singapore 639798
[b] Management Science and Information Systems Department, Rutgers University, NJ 07102, USA
[c] IBM China Research Laboratory, Beijing 100094, China

ARTICLE INFO

Article history:
Received 14 January 2008
Received in revised form 15 April 2009
Accepted 17 April 2009
Available online 5 May 2009

Keywords:
Frequent subgraph mining
G-Measure
G-Pattern

ABSTRACT

Recent years have observed increasing efforts on graph mining and many algorithms have been developed for this purpose. However, most of the existing algorithms are designed for discovering frequent subgraphs in a set of labeled graphs only. Also, the few algorithms that find frequent subgraphs in a single labeled graph typically identify subgraphs appearing regionally in the input graph. In contrast, for real-world applications, it is commonly required that the identified frequent subgraphs in a single labeled graph should also be globally distributed. This paper thus fills this crucial void by proposing a new measure, termed G-Measure, to find globally distributed frequent subgraphs, called G-Patterns, in a single labeled graph. Specifically, we first show that the G-Patterns, selected by G-Measure, tend to be globally distributed in the input graph. Then, we present that G-Measure has the downward closure property, which guarantees the G-Measure value of a G-Pattern is not less than those of its supersets. Consequently, a G-Miner algorithm is developed for finding G-Patterns. Experimental results on four synthetic and seven real-world data sets and comparison with the existing algorithms demonstrate the efficacy of the G-Measure and the G-Miner for finding G-Patterns. Finally, an application of the G-Patterns is given.

1. Introduction

Graph structure data, such as Resource Description Framework (RDF) files, Extensible Markup Language (XML) files, and protein–protein interaction data, are being accumulated in many application domains. The need is now critical to develop efficient and effective graph mining algorithms for analyzing such data. Towards this direction, there have been attempts for mining frequent substructures from a set of labeled graphs [1–9], discovering important/frequent subgraphs from a network [10,11], and mining frequent subtrees from a forest [12–16]. These identified frequent subgraphs are further used for a variety of applications such as designing database schemas [17], building database indexes [18], and modeling user profiles [19].

In this paper, we study a particular graph mining problem, i.e., *mining frequent subgraphs in a single labeled graph*. In fact, when we apply the existing graph mining algorithms to real-world applications, they would face the following practical challenges.

- First, most of the existing algorithms are developed for finding frequent subgraphs in a set of labeled graphs only. These algorithms are able to effectively and efficiently discover all the frequent subgraphs and be scaled to large data sets.

* Corresponding author.
E-mail addresses: jiangxing@pmail.ntu.edu.sg (X. Jiang), hxiong@rutgers.edu (H. Xiong), chwang@cn.ibm.com (C. Wang), asahtan@ntu.edu.sg (A.-H. Tan).
[1] This work was partially done by the first author during his internship at the IBM China Research Laboratory. Also, this research was partially supported by the Singapore National Research Foundation Interactive Digital Media R&D Program under research Grant NRF2008IDM-IDM004-037 and the National Science Foundation (NSF) of USA via Grant number CNS 0831186.

0169-023X/$ - see front matter © 2009 Elsevier B.V. All rights reserved.
doi:10.1016/j.datak.2009.04.008

However, they cannot be directly used to mine in a single labeled graph, despite the fact that the problem of finding frequent subgraphs in a single labeled graph is more general and applicable [11].

- Second, the reason why we conduct frequent pattern mining is that the frequent patterns found are supposed to be the common structures in the data, which can clearly describe the general relationships among items of different classes in the data and effectively analyze the data. However, due to the difference between the graph model and other data models, simply finding frequent subgraphs in a single labeled graph may not fulfil our requirement. The identified subgraphs of high frequency could just include a small set of items of a particular class, which can be interpreted as that these found patterns appear only in a regional area of the input graph (*i.e., located regionally*). In contrast, some less frequent but *globally* distributed subgraphs are more useful, as they provide global information about the whole graph. An illustration of this point is given in Fig. 1, wherein we have found the two 2-edge graph patterns PA (*located globally*) and PB (*located regionally in the specified subarea*) in the input graph. If we merely consider the importance of the frequency, PB should be much more important than PA ($freq(PB) = 10 > freq(PA) = 3$). However, in real-world applications, for example, building graphical indices [18], PA would be more useful, as PA is the typical structure that all the instances of the class A have while PB is only held by one instance of A.
- Third, being a critical property, the downward closure property enables the frequent pattern mining algorithms to effectively prune the search space. However, when mining a single labeled graph, simply counting the occurrence/frequency of a graph pattern in the input graph may not have the downward closure property. For example, given the two graph patterns P and Q shown in Fig. 2, while Q is a superset of P ($P \subseteq Q$), Q has a higher frequency than that of P in the input graph G. This indicates that the downward closure property no longer holds. As a result, the graph mining algorithms cannot utilize the downward closure property to prune the search space. They have to search the whole candidate space for finding all the frequent subgraphs. The mining process becomes extremely time-consuming.

In view of the above challenges, we introduce a new graph mining problem in this paper, i.e., *finding globally distributed frequent subgraph patterns in a single labeled graph*. For simplicity, we call such graph patterns as **G-Patterns** in rest of this paper. Those identified G-Patterns can provide global information about the input graph and are extremely valuable for a variety of applications such as web usage mining [20], hypertext classification [21,22], and biological data analysis [23–26].

To find G-Patterns, the straightforward approach is *divide and conquer*. That is, to partition the input graph into smaller manageable segments, and to find frequent patterns within those segments. However, a large amount of noise may be created by splitting the input graph, rendering this approach ineffective.

Another approach of finding G-Patterns is to adopt a post-processing step which eliminates spurious patterns from the results of the existing mining algorithms. However, as finding such globally distributed subgraphs normally requires setting a lower support threshold, it often leads to the identification of many uninteresting graph patterns and the computation cost can be prohibitively high. For example, given the small synthetic graph V2E1N500L20A55 used in our experiments, Table 1 presents the frequency distribution of the multi-edge graph patterns found in this graph. We could see the distribution is rather skewed. Many patterns are with low frequency while only a few have frequency greater than 100. If we set a lower minimum support threshold, many more patterns would be extracted. It consequently requires much more time to validate whether these found subgraphs are globally distributed.

A better approach is to have a measure that can efficiently identify G-Patterns and automatically remove spurious patterns during the mining process. In this paper, we propose a new measure, named **G-Measure**, for mining G-Patterns in a single labeled graph. The basic idea behind the G-Measure takes the distribution of the graph patterns in the input graph

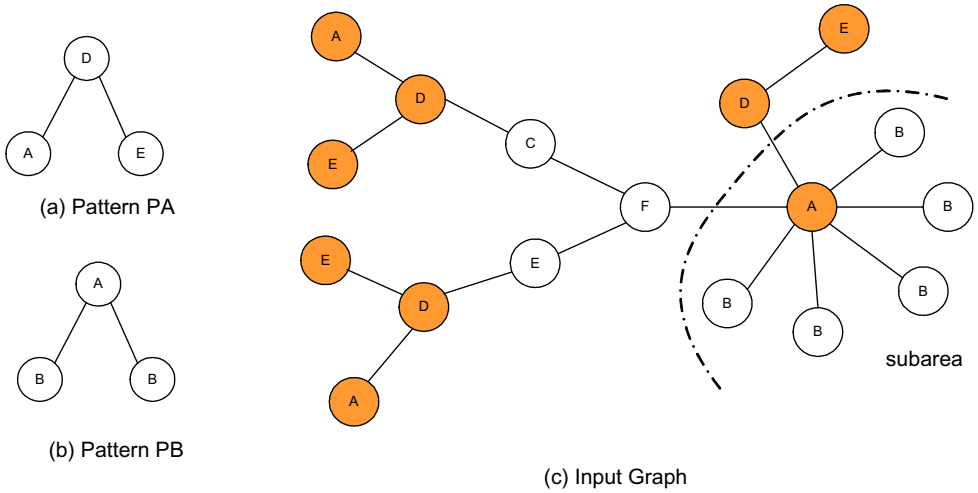

Fig. 1. An illustration of the distribution of the two subgraph patterns found in the input graph, where A–F are the labels of the vertices.

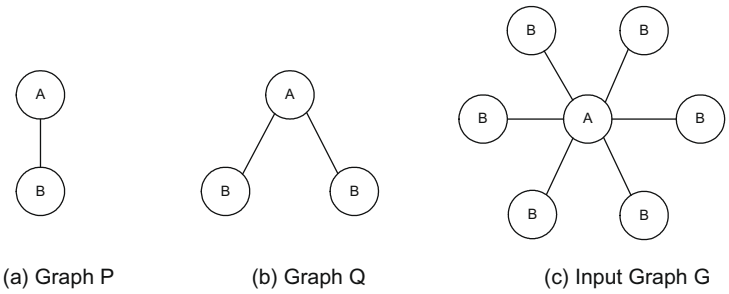

Fig. 2. An illustration of the case that the downward closure property cannot hold. Here, Q is a superset of P but Q has a higher frequency than P in graph G.

Table 1
The frequency distribution of the multi-edge subgraphs found in the synthetic graph. These patterns are divided into seven disjoint groups according to their frequencies.

Group	G1	G2	G3	G4	G5	G6	G7
Frequency	3–10	11–20	21–30	31–40	41–50	50–100	100–1080
#Patterns	76,259	12,536	4070	1247	1170	1559	520

into consideration. Graph patterns with a G-Measure value no less than the user specified minimum support threshold are the ones we are looking for.

Based on the proposed G-Measure, we develop an algorithm called G-Miner for finding G-Patterns. Specifically, G-Miner is implemented following the Depth-First Search (DFS) approach, since this approach has shown to have a computational advantage over the Breadth-First Search (BFS) for mining a single labeled graph [27,11] and works efficiently with the DFS code [3] used for graph isomorphism testing and candidate generation. We evaluate the performance of the G-Miner on both synthetic and real-world data sets and compare with the state-of-the-art graph mining algorithms. As demonstrated by our experiments, G-Miner can efficiently and effectively find globally distributed G-Patterns in a single labeled graph. Indeed, G-Miner runs several orders of magnitude faster than the existing approaches. Finally, we show an application of the G-Patterns.

Overview. The rest of this paper is organized as follows. Section 2 provides the basic concepts related to this graph mining problem. Section 3 introduces the G-Measure. Section 4 elaborates the G-Miner algorithm. The experimental results are given in Section 5. Related work is presented in Section 6. Finally, we draw conclusions in Section 7.

2. Preliminary concepts

In this section, we define the graph model used in this paper and introduce some related concepts.

Definition 1 (Labeled graph). A labeled graph can be represented by a 4-tuple, $G = (V, E, L, l)$, where

- V is a set of vertices,
- $E \subseteq V \times V$ is a set of edges,
- L is a set of labels,
- $l : V \cup E \rightarrow L$, l is a function assigning labels to the vertices and edges.

A subgraph of a graph G, SG, is a graph such that $V_{SG} \subseteq V_G$ and $E_{SG} \subseteq E_G$, and the assignment of endpoints to edges in SG is the same as in G.

Note that we focus on undirected labeled simple graph in this paper. However, our developed algorithm can easily be modified for processing other kinds of graphs. For example, we can treat the direction of the edges as a label for processing directed graphs.

A labeled graph extracted from the Biozon database (http://www.biozon.org) and one of its subgraphs is shown in Fig. 3. *Protein*, *Uni_encodes*, *Unigene*, *DNA*, *Uni_contains*, *Uni_encodes*, and *Encodes* are the labels of the vertices and edges. ID:xxx represents the ID of the corresponding vertex or edge stored in the database. With this graph, researchers can easily explore relationships between these biological resources. Note that the IDs of the vertices and the edges in the labeled graph are not considered during mining. They are presented here for illustration purpose. The frequent graph patterns[2] that we search for are like the one shown in Fig. 3c.

[2] The frequent graph patterns to be found by our algorithm are those containing at least one edge and without disconnected vertices.

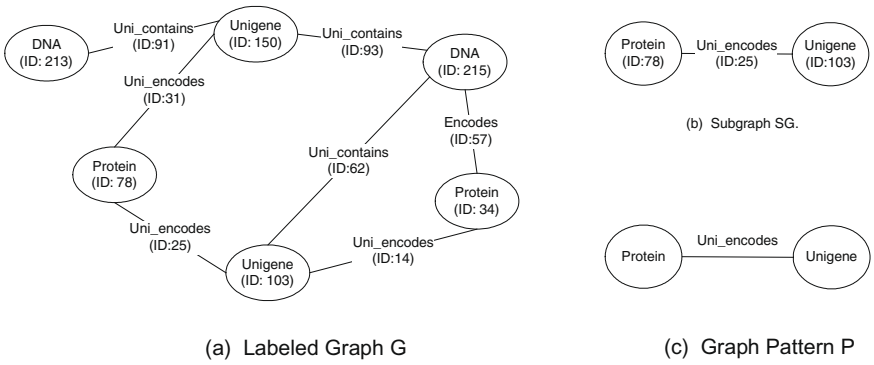

Fig. 3. The illustration of a single labeled graph *G*, its subgraph *SG*, and a graph pattern *P* identified in *G*.

Definition 2 (*Graph instance*). Let *G* be a single labeled graph and *P* be a graph pattern we search for in *G*, a subgraph of *G*, *SG*, is an *instance* of *P* in *G* if there exists an isomorphism between *P* and *SG*.

For example, we search for the graph pattern *P* shown in Fig. 4a in the labeled graph *G* presented in Fig. 3a. We can see there are four subgraphs of *G* isomorphic to *P* (see Fig. 4b). They are called the instances of *P* in *G*.

Definition 3 (*Edge-disjoint based instance graph*). Given the instances of a graph pattern *P* found in an input graph *G*, we can construct a new graph for *P* where the vertices represent *P*'s instances in *G* and edges are added between two vertices if their corresponding instances share an edge in *G*. This new graph is called **edge-disjoint based instance graph**, or **instance graph** for simplicity.

For the example pattern *P* shown in Fig. 4, as its instance *i*1 shares the edge *DNA(ID:215)–Uni_contains(ID:62)–Protein(ID:103)* with *i*2 and *i*3 shares the edge *Protein(ID:78)– Uni_encodes(ID:31)–Unigene(ID:150)* with *i*4, its edge-disjoint based instance graph is constructed as the one shown in Fig. 5a. Note that the instance graph is an unlabeled graph. The labels *i*1, *i*2, *i*3, and *i*4 shown in the example are only used to help indicate the relations between the vertices with their corresponding instances.

Besides the edge-disjoint based instance graph, we can also form other types of instance graphs, for example, vertex-disjoint based instance graph, i.e., edges are added between two vertices of the instance graph if their corresponding instances share vertices in the input graph (see Fig. 5b), or even a distance *l* based instance graph, i.e., edges are added between two vertices if their corresponding instances are reachable with paths of length *l* in the input graph. Forming different instance graphs would be seen as a way of defining the *global distributed* property, since how patterns are called globally distributed are changed in different applications. In other words, the G-Pattern mining problem can be expressed as a function $f(l,\theta)$, where *l* is the preferred distance between the instances and θ is the minimum support threshold. Given the same support threshold, setting a larger *l* will lead to fewer patterns to be found. Therefore, fewer patterns are found with the vertex-disjoint based instance graph than those found with the edge-disjoint based instance graph.

In this paper, we only consider the edge-disjoint based instance graph. This setting specifies that *graph patterns are called globally distributed when their instances do not share common edges in the input graph*. However, it is trivial to fit all our observations, methods, and conclusions to other conditions.

Definition 4 (*The downward closure property*). Given a measure *f*, downward closure property, also called the admissible property [28,29] or frequency anti-monotone property [30], requires that for every pair of patterns *p* and *q* in the data set such that *p* is a subset of *q*[3] ($p \subseteq q$), the value of *p* computed by *f* is not lower than that of *q*, $f(p) \geqslant f(q)$. This property is essential for the computational tractability of most frequent pattern discovery algorithms, since it can be used to quickly narrow the search space.

3. G-Measure

In this section, we present the motivation behind the G-Measure, the definition of the G-Measure, and its downward closure property.

3.1. Motivation

We aim to find frequent subgraphs which are also globally distributed in a single labeled graph. To achieve our goal, the straightforward approach is *divide and conquer*. That is, to partition the input graph into smaller manageable segments, and to find frequent patterns within those segments. However, this approach will suffer from the following problems.

[3] We also call *q* a child of *p* in this paper for simplicity.

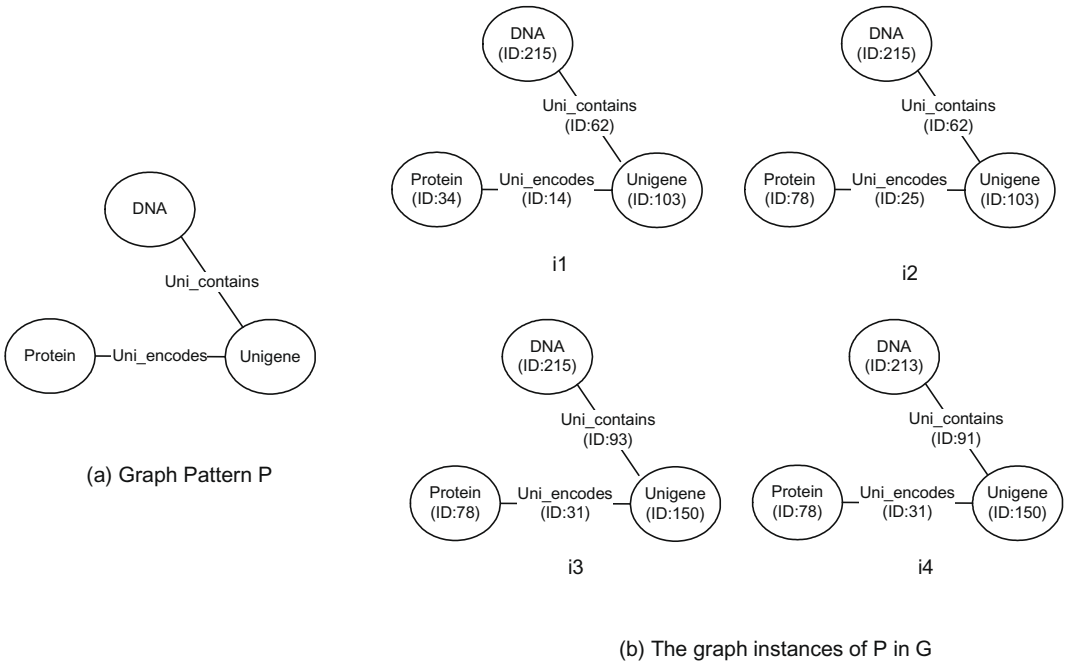

Fig. 4. An illustration of a graph pattern *P* and its four instances in the input graph *G* shown in Fig. 3a.

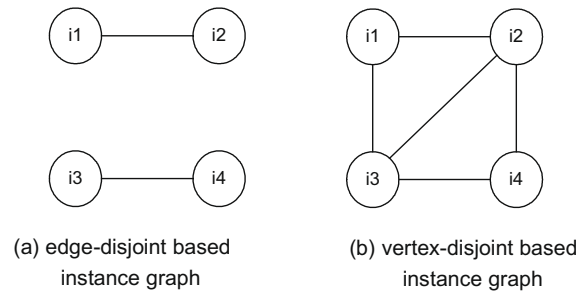

Fig. 5. The edge-disjoint and vertex-disjoint based instance graphs of *P*.

1. The quality of the obtained segments is not ensured, as we have no knowledge of the input graph in advance and may not set an optimal segment number. As a result, the subgraphs found are not the required G-Patterns. For example, if we adopt the *k*-way partition method [31] to split the input graph, too many regionally distributed patterns would be extracted due to a high *k* value.
2. A certain G-Patterns would be lost as their instances happen to occur across different segments. To guarantee the completeness of the G-Patterns to be found, additional methods are required to recover these lost graph instances, which would make the mining task even harder.

Besides the *divide and conquer* approach, we would adopt a post-processing step which eliminates spurious patterns from the results of the existing mining algorithms for finding G-Patterns. But as shown in the introduction section, this approach is computationally inefficient. It requires much time to validate whether these found frequent subgraphs are also globally distributed.

A better approach is to have a measure that can efficiently identify G-Patterns and automatically remove spurious patterns during the mining process. In this paper, we propose a new measure, called **G-Measure**, that can efficiently identify G-Patterns and automatically remove spurious patterns during the mining process.

The basic idea behind the G-Measure is based on the observations of forming the instance graph: when the instances of a graph pattern appear in the same regional area of the input graph, sharing edges, their corresponding vertices in the instance graph would be directly connected with edges. On the other hand, if these instances are located in different areas, their corresponding vertices in the instance graph are less likely to be linked. For the extreme cases, the instance graph could only be composed of a set of disconnected vertices, since no common edges are shared by these instances.

An illustration of our observations is shown in Fig. 6, where we will build the instance graph of *P* with *G*. We can see that *P*'s instances are located in two particular areas of *G*, namely *subarea* 1 and *subarea* 2. Following the definition, we build *P*'s instance graph as shown in Fig. 7. For instances located in the same areas, their corresponding vertices in the instance graph are closely connected, even forming clique, while for instances distributed in different areas, their corresponding vertices are with a few connections.

Our observations are similar to those happening in the social network where humans of the same class closely connect while humans of different classes are with a few interactions. Therefore, we could change the G-Pattern mining problem to a community detection problem. By partitioning the instance graph, we could test whether a pattern is a G-Pattern, where the G-Measure is to count the number of partitions obtained from the instance graph.

Note that an incorrect impression could be made that we can count the number of disconnected components in the instance graph for finding G-Patterns. However, referring to the above example, we can see this approach may not correctly separate instances located in different areas from those in the same areas given some particular distance requirements, for example, edge-disjoint.

3.2. Definition

Before giving the formal definition of the G-Measure, we first present the criterion used for partitioning the instance graph.

Basically, to effectively find community structure in a social network, we assume that for the result obtained, there is a higher density of edges within the same partition than between different partitions. To quantify this requirement, people

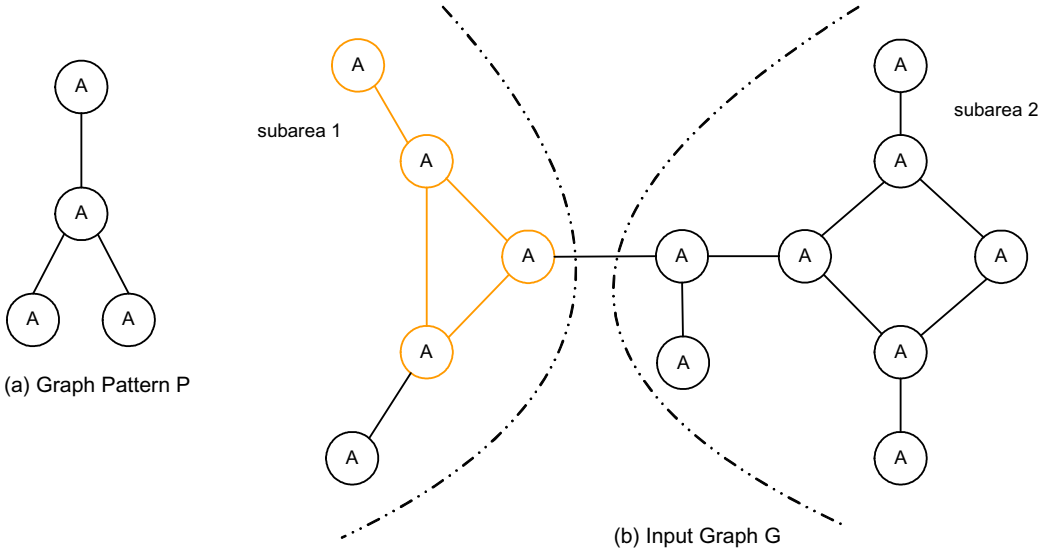

Fig. 6. The graph pattern *P* and the input graph *G*.

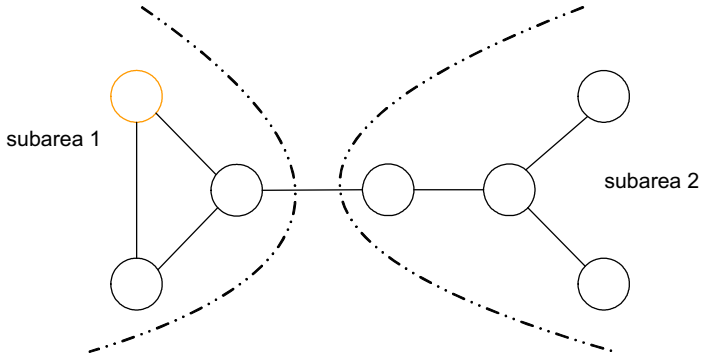

Fig. 7. The built *P*'s instance graph, where the node marked with orange color corresponds to *P*'s instance in Fig. 6 with the same color. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

have proposed different quality measures such as the min cut [32] and the ratio cut [33]. In this paper, we adopt the *modularity* measure [34], which is commonly used in the social network research field to tackle this problem.

The general principle of the *modularity* measure is presented as follows. Let A_{vw} be an element of this network's adjacency matrix where

$$A_{vw} = \begin{cases} 1 & \text{if vertices } v \text{ and } w \text{ are connected,} \\ 0 & \text{otherwise.} \end{cases} \tag{1}$$

and suppose this network has been divided into several partitions such that vertex v is in partition P_v. Thus, the fraction of edges that connect vertices in the same partition, is

$$\frac{\sum_{vw} A_{vw}\delta(P_v, P_w)}{\sum_{vw} A_{vw}} = \frac{1}{2m}\sum_{vw} A_{vw}\delta(P_v, P_w), \tag{2}$$

where $\delta(P_v, P_w)$ is 1 if $P_v = P_w$ and 0 otherwise, and $m = \frac{1}{2}\sum_{vw} A_{vw}$ is the number of edges in the network.

Note that for a random graph, the probability of an edge existing between vertices v and w is $k_v k_w/2m$, where k_v and k_w are the degrees of the vertices v and w. If the social network is correctly partitioned, its value computed by Eq. (2) should be greater than that of a random graph. Otherwise, the two values will be the same. Therefore, the *modularity* measure, M, is defined by

$$M = \frac{1}{2m}\sum_{vw}\left[A_{vw} - \frac{k_v k_w}{2m}\right]\delta(P_v, P_w). \tag{3}$$

A higher M value indicates the result is good.

With the *modularity* measure, we give the formal definition of the G-Measure as follows:

Definition 5 (*G-Measure*). The G-Measure is a metric that reflects the distribution of a graph pattern's instances in the input graph. Given a graph pattern P, the G-Measure $G(P)$ is computed as the number of partitions found on P's instance graph with the modularity measure. If the instance graph is composed of disconnected components, the G-Measure is computed as the sum of the different partitions found in each component.

The definition of the G-Pattern is as follows:

Definition 6 (*G-Pattern*). Given a graph pattern P and a minimum threshold θ, P is a G-Pattern if and only if $G(P) \geqslant \theta$.

3.3. The downward closure property of G-Measure

After defining the G-Measure, we now present its downward closure property. Particularly, we first introduce three types of operations occurring in the instance graph. We show that the instance graph of a graph pattern Q is constructed and can only be constructed under these operations on that of its ancestor P. Therefore, an instance graph based support measure, e.g., the G-Measure, is shown to have the downward closure property if its value is non-decreasing under the three types of operations.

Note that our defined operations are not the same as those given in [29]. Our operations are discovered independently by analyzing the process of forming Q's instance graph on that of its ancestor P. Furthermore, our defined operations are able to test the downward closure property of an instance graph based support measure on all kinds of instance graphs (i.e., edge-disjoint based instance graph, vertex-disjoint based instance graph, and distance based instance graph) while Vanetik et al. have claimed their operations are only proved correct for evaluating the downward closure property of a support measure on the edge-disjoint based instance graph [29].

3.3.1. Operations on the instance graph

Given a graph pattern P and its child Q that is with a growth of one edge[4], Q's instances cannot occur in the input graph suddenly. They all should be generated from P's instances. The corresponding operations shown on P's instance graph belong to three categories, namely forming clique, adding edge, and removing vertex.

Definition 7 (*Forming clique*). Given a pattern P's instance graph $G_P = (V_P, E_P)$ and an arbitrary vertex k on G_P, the forming clique operation on k adds a clique $K = (V_k, E_k)$ into G_P, which results in the instance graph $G_Q = (V_Q, E_Q)$ where

$$V_Q = V_P \setminus \{k\} \bigcup V_k$$

and

$$E_Q = (\{u, v\} | u, v \in V_P \setminus \{k\}, \{u, v\} \in E_P) \bigcup (\{u', v'1\} | u' \in V_P \setminus \{k\},$$
$$v' \in V_k, \{u', k\} \in E_P) \bigcup (\{u'', v''\} | u'', v'' \in V_k, \{u'', v''\} \in E_k).$$

[4] For simplicity, we only give examples that Q is with a growth of one edge from P. However, it is trivial to consider the growth of many edges.

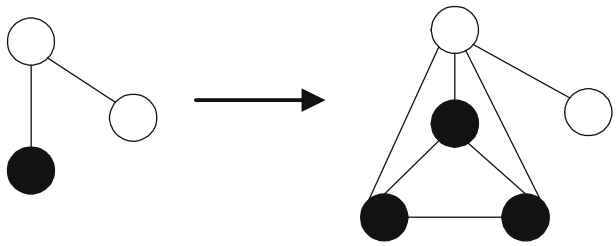

Fig. 8. An example of the forming clique operation, where the vertices marked with black color are used for highlight purpose.

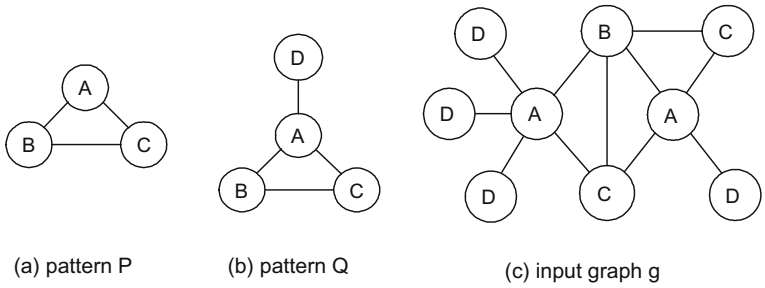

Fig. 9. The sample patterns and the input graph corresponding to the forming clique operation shown in Fig. 8.

Fig. 8 presents an example of the forming clique operation, where a clique K_3 is added into the instance graph. It happens when several instances of Q are generated from the same instance of P. The sample patterns and the input graph corresponding to this example are given in Fig. 9.

Definition 8 (*Removing vertex*). Given the instance graph $G_P = (V_P, E_P)$, removing vertex operation removes a vertex v from the instance graph, which results in the instance graph $G_Q = (V_Q, E_Q)$, where

$$V_Q = V_P \setminus \{v\},$$

and

$$E_Q = E_P \setminus \{\{u, v\} | u \in V_P \setminus \{v\}, \{u, v\} \in E_P\}.$$

Definition 9 (*Adding edge*). Given the instance graph $G_P = (V_P, E_P)$, adding edge operation adds a new edge $\{u, v\}$ into the instance graph, resulting in the graph $G_Q = (V_P, E_P \bigcup \{\{u, v\}\})$.

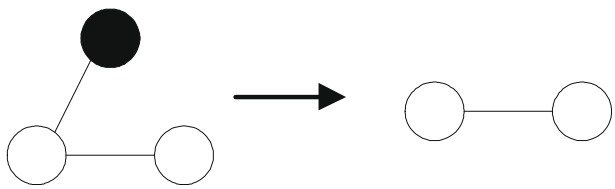

Fig. 10. An example of the removing vertex operation, where the vertex marked with black color is used for highlight purpose.

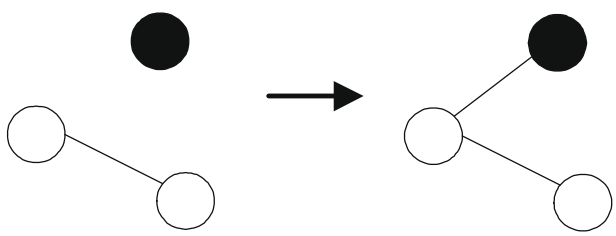

Fig. 11. An example of the adding edge operation, where the vertices marked with black color are used for highlight purpose.

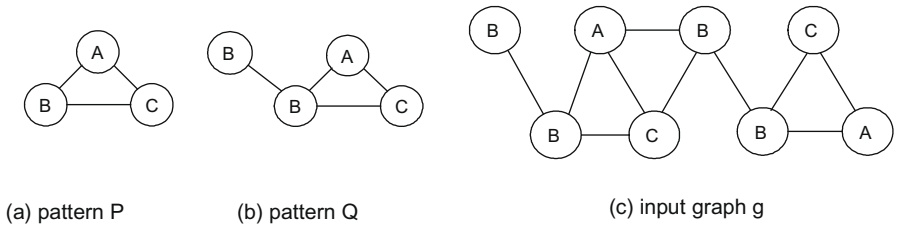

Fig. 12. The sample patterns and the input graph corresponding to the adding edge operation shown in Fig. 11.

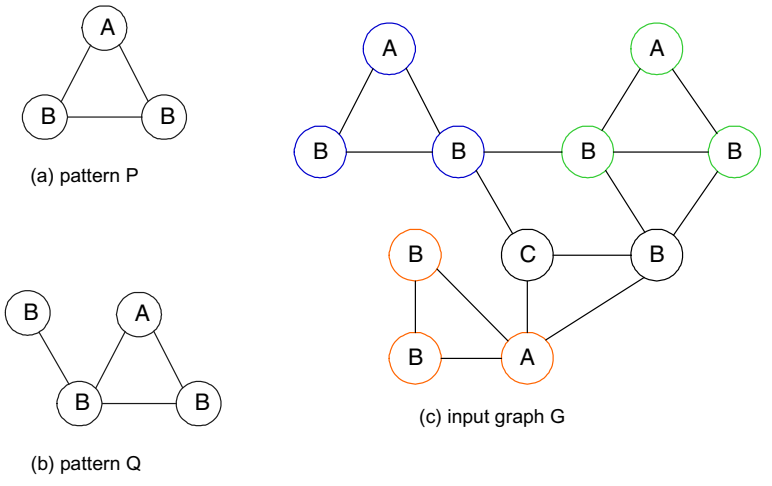

Fig. 13. The sample patterns and the input graph for illustrating how the instance graph of Q is constructed from that of P with the three operations.

The examples corresponding to the removing vertex operation and the adding edge operation are given in Figs. 10 and 11, respectively. The former operation occurs if an instance of P cannot generate Q's instance with one edge growth and the latter happens when the instances of Q share edges in the input graph. The sample patterns and the input graph corresponding to the adding edge operation shown in Fig. 11 are presented in Fig. 12.

3.3.2. Constructing instance graphs with the three operations

After defining the three operations, we now present how the instance graph of Q can be and only be constructed from that of P under these operations.

First, we show that the instance graph of Q can be constructed from that of P with the three operations. An example is given in Fig. 13, where we will build the instance graph of Q based on that of P (see Fig. 14a). Firstly, for P's instances that cannot generate instances of Q with one edge growth, we remove their corresponding vertices from the instance graph (see Fig. 14b). Then, the forming clique operation is performed on the instance graph (Fig. 14c). Finally, we add a new edge to the instance graph, obtaining the instance graph of Q (see Fig. 14d). Note that the orders of these operations do not affect the final instance graph obtained. They can be arranged freely when building Q's instance graph.

Second, we prove that the instance graph of Q can only be constructed from that of P with the three operations. For demonstration, we first introduce two other operations[5], namely deleting an edge from P's instance graph (Fig. 15) and adding completely new vertices (Fig. 16) into Q's instance graph whose corresponding instances are not generated from P's instances. The two operations may also be supposed by people to happen when forming Q's instance graph. Therefore, if the two operations do exist, our statement is incorrect.

Then, we show that the two operations cannot exist. For example, if the deleting edge operation happens (see Fig. 15), the corresponding instances of $v1$ and $v2$ in Q's instance graph, Q_{v1} and Q_{v2}, will not share a common edge in the input graph.[6] However, Q's instances are generated based on P's instances. If Q_{v1} and Q_{v2} do not share an edge, its ancestors P_{v1} and P_{v2} do not share an edge either. It conflicts with the truth that P_{v1} and P_{v2} share edges in the input graph. The same conclusion can be

[5] Given an instance graph, the possible operations on it can only be adding edge, adding vertices, deleting vertices, and deleting edges.

[6] This proof given here is based on the edge-disjoint based instance graph only. However, it is trivial to demonstrate its correctness on other kinds of instance graphs.

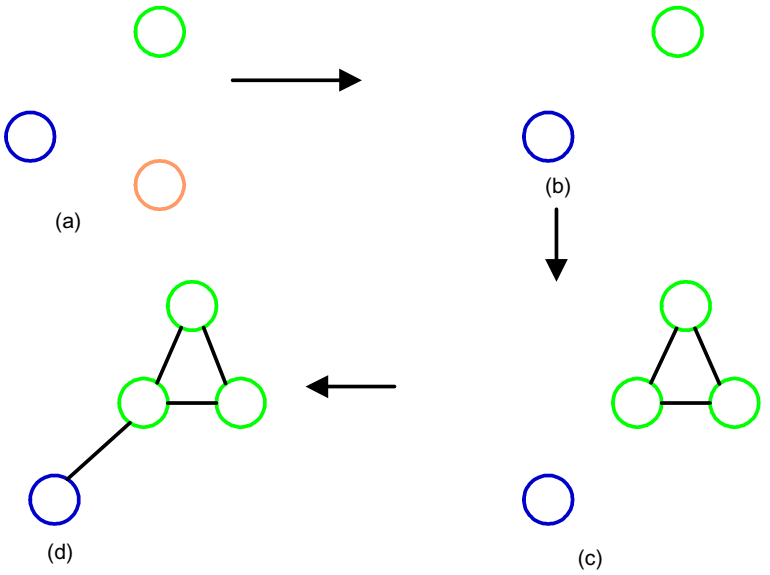

Fig. 14. The detailed steps of building Q's instance graph.

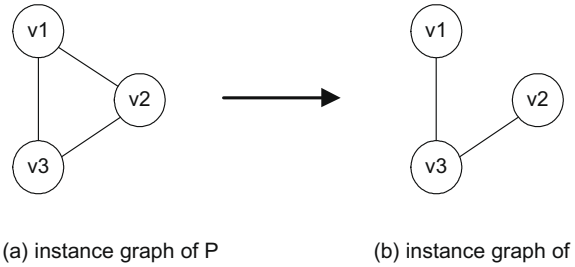

Fig. 15. Deleting an edge from P's instance graph for forming Q's instance graph.

obtained for the operation of adding vertices into the instance graph. Therefore, the instance graph of Q can only be built from that of P with the above three defined operations.

Theorem 1. *An instance graph based support measure for mining frequent subgraphs in a single labeled graph has the downward closure property if its value is non-decreasing under the forming clique, removing vertex, and adding edge operations on the instance graph.*

Proof. As demonstrated above, given a graph pattern P and its child Q, Q's instance graph can be and only be constructed under the three types of operations on that of P. If a measure f is non-decreasing under the three type of operations, the value of P computed by f will not be lower than that of Q, $f(P) \geqslant f(Q)$, satisfying the definition of the downward closure property. This measure therefore has the downward closure property. □

3.3.3. Conditions for the downward closure property

To evaluate the downward closure property of G-Measure, the standard approach is to test whether $G(P) \geqslant G(Q)$. However, it is well known that the problem of deciding the optimal partition number of a graph has no standard solutions yet [34]. Although we have many quality measures, the results are still with uncertainness because of many factors, which will further affect the evaluation. For example, two different partition results may be obtained given the same input graph (see Fig. 17), which can further ruin the downward closure property. Therefore, we design a new approach to compute the G-Measure values.

The basic idea of the new approach is based on the fact that all the instances of Q are generated from those of its ancestor P with edge growth. If we have partitioned P's instance graph, we may not directly partition that of Q again. The vertices of Q's instance graph can be placed into different partitions according to its ancestors. Specifically, if a vertex of P's instance graph, v_p, is placed into a partition Par_i, all the vertices of Q's instance graph that are generated from v_p can also be placed

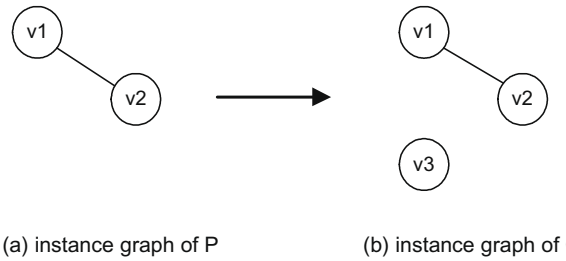

Fig. 16. Adding a new vertex into Q's instance graph whose corresponding instance is not generated from *P*'s instance.

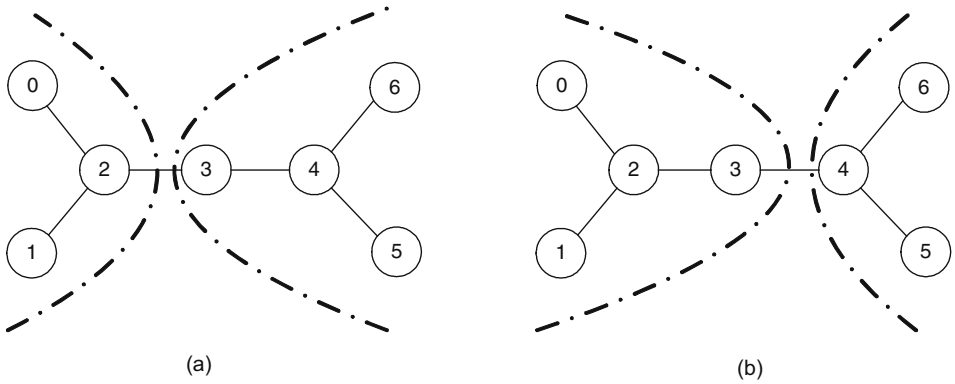

Fig. 17. An illustration of the problem we suffer from, where two different partition results obtained from the same instance graph.

into Par_i. Such an approach can keep the downward closure property of the G-Measure with the following lemmas and Theorem 1.

Lemma 1. *Given instance graph of a graph pattern P and that of its child Q, if Q's instance graph is partitioned according to the partition result of P's instance graph, i.e., the vertices of Q's instance graph are placed together if their ancestors in P's instance graph are in the same partition, the G-Measure of P will equal to that of Q, $G(P) = G(Q)$, when forming clique operation happens on the instance graph.*

Proof. Given a vertex of *P*'s instance graph, v_p, that has been placed into a partition Par_i, when a clique *K* is generated from v_p, all the vertices of *K* will also be placed into Par_i. No new partition is formed on the instance graph and no existing partition is deleted, i.e., $G(P) = G(Q)$. □

Lemma 2. *Given instances of a graph pattern P and those of its child Q, if Q's instance graph is partitioned according to the partition result of P's instance graph, i.e., the vertices of Q's instance graph are placed together if their ancestors in P's instance graph are in the same partition, the G-Measure of P will equal to that of Q, $G(P) = G(Q)$, when adding edge operation is conducted on the instance graph.*

Proof. Given two vertices of *P*'s instance graph, v_{p1} and v_{p2}, and two vertices of *Q*'s instance graph, v_{q1} and v_{q2}, where v_{q1} is generated from v_{p1} and v_{q2} is generated based on v_{p2},

- if v_{p1} and v_{p2} both are in a partition Par_1, v_{q1} and v_{q2} will be in Par_1 too. Adding one edge between v_{q1} and v_{q2} does not form a new partition or delete an existing partition of the instance graph, i.e., $G(P) = G(Q)$.
- if v_{p1} is in partition Par_1 and v_{p2} is in partition Par_2, v_{q1} will be in Par_1 and v_{q2} is in Par_2 subsequently. Adding one edge between v_{q1} and v_{q2} does not form a new partition or remove an existing partition either, i.e., $G(P) = G(Q)$. □

Lemma 3. *Given instances of a graph pattern P and those of its child Q, if Q's instance graph is partitioned according to the partition result of P's instance graph, i.e., the vertices of Q's instance graph are placed together if their ancestors in P's instance graph are in the same partition, the G-Measure of P will equal to or be greater than that of Q, $G(P) \geqslant G(Q)$, when removing vertex operation is conducted on the instance graph.*

Proof. Given a vertex of P's instance graph, v_p, that is placed in a partition Par_1 and an instance p corresponding to v_p, if no instance of Q can be generated based on p, v_p should be removed from the instance graph. Specifically,

- If Par_1 has more vertices besides v_p, Par_1 will still exist after removing v_p, i.e., $G(P) = G(Q)$.
- If Par_1 only contains v_p, removing v_p will also lead to deleting Par_1, i.e., $G(P) > G(Q)$.

Therefore, $G(P) \geqslant G(Q)$ when removing vertex from the instance graph. □

Note that directly adopting above method can only produce an approximate G-Measure value. However, as all the instances of the graph pattern Q are generated from those of its ancestor P, the exact modularity value can actually be computed by merging these partitions obtained. Merging two neighbor partitions simply as if the modularity value increases[7], which can be seen as a local optimization of the modularity value. A global optimization has to analyze all the partitions for maximizing the modularity value. Because the local optimization value is not less than the exact value, we can quickly prune many spurious patterns by doing local optimizations only.

With above observations, the new approach is designed for mining G-Patterns. Specifically, we only partition a few patterns' instance graphs directly with the partition algorithm (in this paper only the 2-edge patterns). The instance graphs of the descendant patterns are firstly partitioned according to their ancestors. Then, the G-Measure value is computed by merging these possible partitions. If a pattern's value is already lower than the threshold during local optimizations, it can be discarded directly.

If we map this approach to the input graph, we could find it actually follows the idea of *divide and conquer*. However, the new approach is based on the instance graphs to partition the input graph, which can be seen as a type of prior knowledge of the input graph. It thus avoids the problems mentioned in Section 3.1.

4. The G-Miner algorithm

In this section, we present the G-Miner algorithm which is designed for mining G-Patterns with G-Measure values not less than a user specified minimum threshold. Specifically, we first introduce the Depth-First Search (DFS) code based approach [3] for candidate generation and graph isomorphism testing. Then, we present the details of the G-Miner algorithm.

4.1. The DFS code

Graph isomorphism testing, subgraph isomorphism testing, and candidate generation are costly steps in the frequent subgraph mining problems. In G-Miner, we use the DFS code based approach to solve these problems.

The basic ideas behind the DFS code based approach are given as follows. For a labeled graph G, for example, the one shown in Fig. 18a, we have different DFS trees of this graph (see T1 in (b) and T2 in (c)). We call edges included in the DFS tree are the forward edges and edges not in the DFS tree are the backward edges. A linear order $\prec$ of the edges in G is then defined. Particularly, given a DFS tree T and two edges $a = (vi_a, vj_a)$ and $b = (vi_b, vj_b)$ in G where vi_a is the start node of a and vj_a is the end node of a according to T,

$$a < b = \begin{cases} a \text{ and } b \text{ are forward edges}, & (vj_a < vj_b) \| (vi_a > vi_b \&\& vj_a = vj_b), \\ a \text{ and } b \text{ are backward edges}, & (vi_a < vi_b) \| (vi_a = vi_b \&\& vj_a < vj_b), \\ a \text{ is a forward edge and } b \text{ is a backward edge}, & vj_a \leqslant vi_b, \\ a \text{ is a backward edge and } b \text{ is a forward edge}, & vi_a < vj_b. \end{cases} \tag{4}$$

With the $\prec$, the edges in G can be arranged into a sequence for individual DFS trees, called DFS code of G. For example, the DFS code for T1 is $(\langle v0, v1, B, y, A\rangle, \langle v1, v2, A, x, A\rangle, \langle v2, v3, A, y, C\rangle, \langle v3, v1, C, x, A\rangle)$ and that of T2 is $(\langle v0, v1, A, x, A\rangle, \langle v1, v2, A, x, C\rangle, \langle v2, v0, C, y, A\rangle, \langle v1, v3, A, y, B\rangle)$.

As a linear order over the labels in the graph can further be defined, for example, $A < B < C$ for the input graph G[8], we are able to sort these different DFS codes obtained. For example, as $\langle v0, v1, A, x, A\rangle$ is smaller than $\langle v0, v1, B, y, A\rangle$ with the linear order over the labels, the DFS code of T2 is thus smaller than that of T1, which can also be proved to be the minimum DFS code of G.

Because two graphs are isomorphic if they have the same minimum DFS code, the problem of mining frequent subgraphs can be changed to mining frequent minimum DFS codes. Particularly, the edge sequence for generating a subgraph must be the minimum DFS code of this subgraph. As a result, only a few candidates would be explored, which reduces the total cost of subgraph isomorphism testing and candidate generation.

[7] Note that merging two partitions will not ruin the downward closure property, since the number of partitions can only decrease.

[8] Note that we can order the labels using other approaches besides the lexicographic order. For example, in [3], the labels are ordered according to their frequencies in the data set.

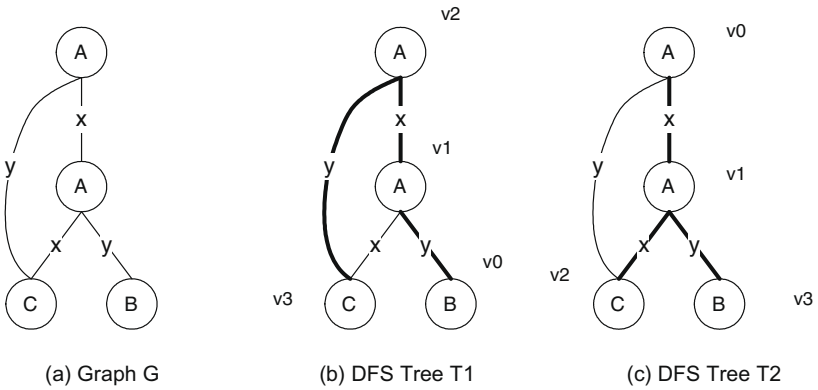

Fig. 18. An illustration of the DFS code based approach. The input graph *G* has different DFS trees, which are shown in (b) and (c).

4.2. Algorithm details

G-Miner utilizes a Depth-First Search method to find G-Patterns, since this approach has shown to have a computational advantage over the Breadth-First Search for mining a single labeled graph [27,11] and works efficiently with the DFS code. Specifically, G-Miner starts with a set of frequent 2-edge graph patterns. Then, it recursively generates potential descendants with a growth of one edge. Graph patterns with G-Measure value not less than the minimum support threshold are selected.

Fig. 19 outlines the pseudo-code of the G-Miner algorithm. It is similar to the `gSpan` algorithm. The major differences between the two algorithms are that G-Miner will partition the instance graphs and use the G-Measure for finding frequent subgraphs. Explanations of this algorithm are presented as follows.

Step 1 (line 1–4): Scan the whole input graph *G* into the memory. Remove infrequent vertices and edges from the input graph. The remaining vertices and edges are relabeled in descending frequency for forming 1-edge graph patterns. These frequent 1-edge graphs are added into S^1 and sorted in the DFS lexicographic order. Note that in this paper we only consider the problem that the whole data set can be held into the main memory. For large scale data sets, we could adopt our developed technique on mining a set of graphs [4] with certain motivations. That will be our future work.
Steps 2–4: Loops to generate qualified G-Patterns. It stops after all the elements of S^1 are explored.
Step 2 (line 6): For each 1-edge graph pattern, enumerate its potential 2-edge children. Note that only graph patterns satisfying the minimum DFS code are built.
Step 3 (line 8–10): For each 2-edge graph pattern *P*, record *P*'s instances in the input graph *G*, build *P*'s instance graph and partition this graph. Each instance will be assigned into a particular partition according to the partition result.
Step 4 (line 11): Subgraph_mining recursively generates *P*'s potential descendants with a growth of one edge (satisfying the minimum DFS code). Only graph patterns with G-Measure value greater than the minimum support are added to *S* (**line 17**) and further explored (**line 18**). Their instances are firstly placed into different partitions according to the ancestors (**line 21**). Then, neighbor partitions are merged as new partitions (similar to the agglomerative clustering) if the modularity value increases (**line 13**). For patterns whose G-Measure values are already lower than the threshold before finishing the merge process, they are discarded immediately.
Step 5 (line 12): Add the frequent 1-edge graph set S^1 to *S*.

Note that the reason why G-Miner starts from 2-edge patterns but not 1-edge patterns is because the instance graphs of the 1-edge patterns are all composed of a set of disconnect vertices. As a result, the G-Miner algorithm cannot separate instances that are located in the same area from those appearing in different areas, making poor performance for finding G-Patterns.

5. Experiments

In this section, we evaluate the performance of the G-Miner algorithm and the usefulness of the G-Patterns. Specifically, we evaluate: (1) the effectiveness of the G-Miner algorithm for mining G-Patterns. (2) the computational efficiency of the G-Miner algorithm for mining G-Patterns.[9] (3) the application of G-Patterns for text categorization.

[9] For evaluation, we only compare G-Miner with algorithms in which a post-processing step can be adopted, as we do not find an effective way of defining the *K* value for partitioning the input graph and make it difficult to compare with the *divide and conquer* approach.

G-Miner Algorithm

Input: G: a labeled graph.
θ: a user-specified minimum support threshold.
Output: S: frequent subgraph set.

main(G, θ, S) // the main function
1. sort the labels of the vertices and edges in G by their frequency
2. remove infrequent edges and vertices
3. relabel the remaining vertices and edges in descending frequency
4. sort the 1-edge graphs into S^1 referring to the DFS lexicographic order
5. **foreach** 1-edge graph $sg \in S^1$
6. enumerate sg's potential 2-edge children
7. **foreach** 2-edge graph P, P is a child of sg
8. record P's instances in G
9. build and partition P's instance graph
10. assign each instance of P into its corresponding partition
11. **subgraph_mining**(G, S, P)
12. $S \leftarrow S \cup S^1$

subgraph_mining(G, S, P) //the sub procedure
13. merge different partitions into a new partition if it is possible
14. compute the G-Measure value of P, $G(P)$
15. **if** $G(P) < \theta$ **then**
16. **return** NULL
17. $S \leftarrow S \cup \{P\}$
18. generate all P's potential children with one edge growth
// only patterns with minimum DFS code are generated
19. **foreach** P^*, P^* is a child of P
20. record the instances of P^* in G
21. assign each instance into a particular partition according to its parent
22. **subgraph_mining**(G, S, P^*)

Fig. 19. The pseudo-code of the G-Miner algorithm.

5.1. The performance of the G-Miner algorithm

5.1.1. The experimental setup

We evaluate the performance of the G-Miner algorithm on both synthetic and real-world data sets.

Synthetic Data. We develop a synthetic data generator which is based on Barabasi's evolving scale-free random graph model [35] to produce labeled power-law graphs, as we can easily observe the difference between the patterns mined by the existing frequent subgraph mining algorithms and those discovered by G-Miner on such graphs.

The procedure of this generator for producing a labeled graph is as follows: Firstly, by setting the initial vertex seed number $|V|$, the number of edges to be added at each step $|E|$, and the number of steps for evolution N, an unlabeled power-law graph is generated. Then, each vertex is assigned a particular label, which is controlled by the number of labels in the graph (L) and the probability of neighbor vertices sharing a same label (α). The details how the parameters L and α affect the frequent patterns mined are given in Section 5.1.2. The summary of the parameter settings used to create the synthetic graphs is given in Table 2.

Real-world Data. Seven real-world data sets are used in the experiments. The basic characteristics of these data sets are shown in Table 3.

The credit and aviation data sets are downloaded from SUBDUE's web site[10]. Among all, the aviation data set is the largest used in the experiments, with more than 100,000 vertices and 90,000 edges.

The citation50_15, citation50_20, and citation50_25 data sets are generated from the citation graph used in the KDD Cup 2003[11], the vertices of which represent research papers and the edges of which indicate the citation relation between papers.

[10] http://www.subdue.org.
[11] http://www.cs.cornell.edu/projects/kddcup/.

Table 2
Parameter settings for the synthetic data sets used in the experiments.

Data set	$\|V\|$	$\|E\|$	N	L	α
V2E1N500L20A55	2	1	500	20	0.55
V2E1N500L20A05	2	1	500	20	0.05
V2E1N550L10A65	2	1	550	10	0.65
V2E1N550L100A65	2	1	550	100	0.65

Table 3
The characteristics of the real-world data sets.

Data set	Total number of		Total number of	
	Vertices	Edges	Vertex label	Edge label
Credit	14,700	14,000	59	20
Aviation	101,185	98,576	6173	51
Citation50_25	18,114	47,776	50	1
Citation50_20	15,766	34,494	50	1
Citation50_15	12,760	21,219	50	1
PPI	4931	10,003	28	1
Chemical	9189	9317	66	4

Because the original citation graph does not have any meaningful labels for the vertices and edges, we firstly apply a clustering algorithm to group the abstracts of the papers into 50 topic clusters. The cluster IDs are then assigned to the vertices as their labels. For the edges, we give them a common label, since they all represent the citation relation. As our experiments are assumed for processing undirected graphs only, the directions of the edges are ignored. Note that the original citation graph is very dense, containing 29,555 vertices and 352,807 edges. A simple graph pattern can have many instances in the data set (for example, the one shown in the Fig. 20 even has a frequency of 1,843,068), which can lead to the programs quickly run out of memory even just starting mining. Therefore, we only use a subgraph of the original graph for evaluation, which is created by removing vertices with degree greater than a threshold θ and their associated edges from the original graph. In our experiments, we set the thresholds as 15, 20, and 25, respectively.[12] Such settings enable us to conduct a more comprehensive study and allow the computation to be finished in a reasonable amount of time.

The PPI data set is a protein–protein interaction network[13], in which each vertex represents a protein and an edge is added between two vertices if the corresponding proteins are detected to have an interaction in the experiments. For each vertex, we assign the protein's functional class as its label, where the functional class scheme used is the one established by the MIPS database[14]. Since all the edges represent the interaction relation, we only give one label to them. Similar to the citation graphs, we remove vertices whose degrees are greater than 10 and the incident edges from the original graph to guarantee the computation to be done in a reasonable mount of time.

The chemical data set[15], provided for the Predictive Toxicology Evaluation Challenge [36], contains 340 chemical compounds (i.e., 340 labeled graphs) that has been used for evaluating the performance of FSG [2] and gSpan [3]. It is used by us to test whether G-Miner can also effectively find frequent subgraphs from a set of small graphs as we claim. Note that the 340 small graphs have to be merged into a large graph as the input and each becomes a disconnected component of the large graph.

Baseline Algorithms. In our experiments, we implement two algorithms, namely baseline-1 and baseline-2, as the baselines for comparison. The details of the two algorithms are given as follows.

The baseline-1 algorithm is a standard algorithm for finding frequent subgraphs in a single labeled graph. The support value of a graph pattern is computed as the exact number of its instances in the input graph. Its pseudo-code is given in Fig. 21.

As mentioned previously, the standard support measure, i.e., the one used in baseline-1, may not have the downward closure property for mining a single labeled graph. To overcome this problem, a downward closure property holding measure is proposed by Vanetik et al. [28] that computes the maximum independent set of a pattern's instance graph and assigns the size as its support value (for example, for the pattern P shown in Fig. 4a, its support value would be 2 with this support measure). Programs with this support measure are supposed to quickly discover a set of frequent subgraphs from the input graph, although it cannot find all the frequent subgraphs. Here, the baseline-2 algorithm is implemented with Vanetik's support measure for finding frequent subgraphs. Its pseudo-code is given in Fig. 22.

[12] When setting a higher threshold θ, the resulted graph would be denser, making more subgraphs extracted with more time.
[13] The *S. cerevisiae* data set is available at http://dip.doe-mbi.ucla.edu/dip/Download.cgi.
[14] This scheme is available in http://mips.gsf.de/projects/funcat, containing 28 classes.
[15] http://www.comlab.ox.ac.uk/activities/machinelearning/PTE/.

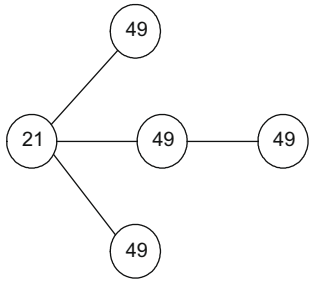

Fig. 20. A 4-edge pattern with frequency of 1,843,068 in the original citation graph.

Baseline-1 Algorithm
Input: G: a labeled graph.
θ: a predefined support threshold.
Output: S: frequent subgraph set.

main(G, θ, S) // the main function
1. sort the labels of the vertices and edges in G by their frequencies
2. relabel the remaining vertices and edges in descending frequency
3. sort the 1-edge graphs into S^1 according to the DFS lexicographic order
4. **foreach** 1-edge graph $sg \in S^1$
5. **subgraph mining**(G, S, sg)
6. $S \leftarrow S \cup S^1$

subgraph mining(G, S, P) //the sub procedure
7. compute the support value of P, $S(P)$
8. **if** $S(P) \geq \theta$ **then**
9. $S \leftarrow S \cup \{P\}$
// *Cannot stop mining here because of missing downward closure property.*
10. generate all P's potential children with one edge growth
11. **foreach** P^*, P^* is a child of P
12. **subgraph mining**(G, S, P^*)

Fig. 21. The pseudo-code of the baseline-1 algorithm.

All the experiments are conducted on an Intel 64-bit Xeon CPU (2.0 GHz) PC with 4G main memory, running linux. When partitioning the instance graph, we use a fast community detection algorithm [34] with time complexity of $O(md \log n)$, where n is the number of vertices, m is the number of edges, and d is the depth of the dendrogram. When computing the MIS of the instance graph, we use a fast exact maximum clique algorithm *wclique* [37]. Note that the reason of using a 64-bit PC is because the 32-bit JVM can only support up to 1.5 Gigabyte memory while some data sets require more memory. In our experiments, the maximum memory heap size is 3G. Also, as our program is implemented using JAVA, it gives us a disadvantage when comparing with programs using C/C++.

5.1.2. Illustration of the global distribution of G-Patterns

Firstly, we evaluate the effectiveness of the G-Miner algorithm for mining G-Patterns. As the real-world data sets are too large for visualization, we only use the synthetic data sets here. For data sets V2E1N500L20A55 and V2E1N500L20A05, we set a lower α value to the latter so that the probability of two linked vertices sharing a common label is more random, which leads to fewer frequent patterns to be found. For data sets V2E1N550L10A65 and V2E1N550L100A65, assigning more labels to the latter also leads to fewer frequent patterns to be found.

The experimental results of the G-Miner algorithm on the synthetic data sets are given in Figs. 23–26, respectively. We have highlighted vertices which are contained in the top 10 frequent multi-edge G-Patterns found in these data sets. Also, we mark the coverage of the top 10 frequent multi-edge patterns found by baseline-1 whose edge number is not greater than the maximal edge number of the frequent G-Patterns discovered.

We can see patterns found by baseline-1, as we expect, occur only in several particular areas of the input graphs (particularly, these found patterns all occur around the vertices with high degrees), while patterns found by G-Miner distribute

Baseline-2 Algorithm
Input: G: a labeled graph.
θ: a predefined support threshold.
Output: S: frequent subgraph set.

main(G, θ, S) // the main function
1. sort the labels of the vertices and edges in G by their frequencies
2. remove infrequent edges and vertices
//baseline-1 cannot conduct this operation
3. relabel the remaining vertices and edges in descending frequency
4. sort the 1-edge graphs into S^1 according to the DFS lexicographic order
5. **foreach** 1-edge graph $sg \in S^1$
6. **subgraph mining**(G, S, sg)
7. $S \leftarrow S \cup S^1$

subgraph mining(G, S, P) //the sub procedure
8. compute maximum independent set of P's instance graph, $MIS(P)$
9. **if** $MIS(P) < \theta$ **then**
10. **return** NULL
11. $S \leftarrow S \cup \{P\}$
12. generate all P's potential children with one edge growth
13. **foreach** P^*, P^* is a child of P
14. **subgraph mining**(G, S, P^*)

Fig. 22. The pseudo-code of the baseline-2 algorithm.

Table 4
The number of unique vertices contained in the frequent patterns found by G-Miner and Baseline-1.

Data sets	G-Miner	Baseline-1
V2E1N500L20A55	107	52
V2E1N500L20A05	59	20
V2E1N550L10A65	185	54
V2E1N550L100A65	44	22

globally in the synthetic graphs. Table 4 lists the number of unique vertices found by the two algorithms. Although the number of frequent subgraphs in each synthetic graph is different, G-Miner can always effectively find G-Patterns which cover more unique vertices of the input graphs.

To save space, we do not present the coverage of the frequent patterns by baseline-2 in this paper. The interested readers can find that patterns mined by baseline-2 seem globally distributed in the input graph as well. However, the G-Patterns cannot be effectively identified by baseline-2. An illustration of this point is given in Fig. 27, where the instance graph of a particular graph pattern is shown. If we use the G-Measure, the value of this pattern would be 1, since the vertices of the instance graph are closely connected. However, if we compute the maximum independent set of the instance graph, this pattern's value will be 2. These closely connected instances cannot be identified as a whole.

5.1.3. Computational efficiency of mining G-Patterns

We evaluate the computational performance of the G-Miner algorithm for finding G-Patterns on the first six real-world data sets.[16] The baseline-1 and baseline-2 algorithms are used for comparison, as a post-processing step can be applied to the results of the two algorithms for finding G-Patterns. If G-Miner outperforms baseline-1 and baseline-2 in the experiments, it certainly runs faster than the approaches that further adopt a post-processing step.

Table 5 shows the runtime (in seconds), the minimum support threshold used, and the number of frequent patterns found by the three algorithms. Entries marked with "-" represent experiments aborted for running more than one day or out of memory. We can see G-Miner runs greatly faster than baseline-1 and baseline-2 on these data sets. Given the same minimum support threshold and 24-hour slot, baseline-1 cannot work on any data sets and baseline-2 works only on three. Such results clearly demonstrate the computational efficiency of the G-Miner algorithm for finding G-Patterns.

[16] The chemical data set is only used to compare the performance of G-Miner with those of gSpan and FSG shown later.

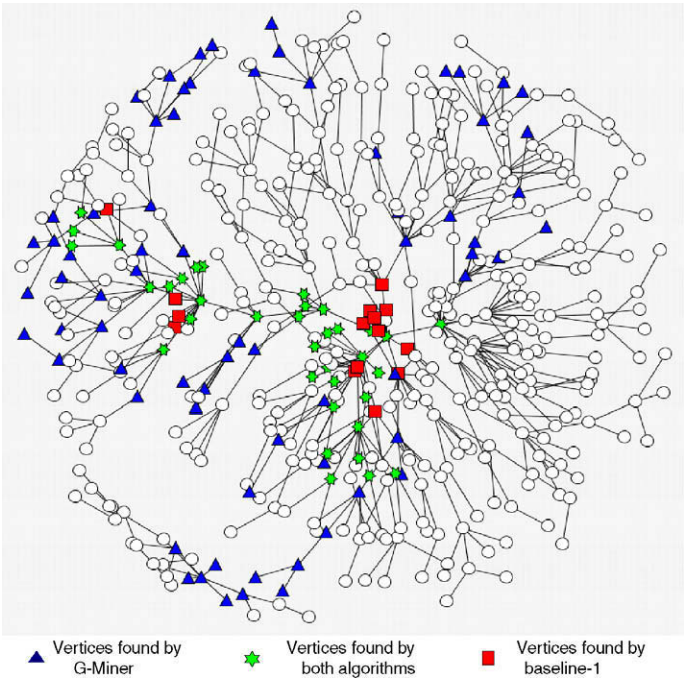

Fig. 23. The coverage of the top 10 frequent multi-edge patterns extracted by G-Miner compared with that of baseline-1 on graph V2E1N500L20A55.

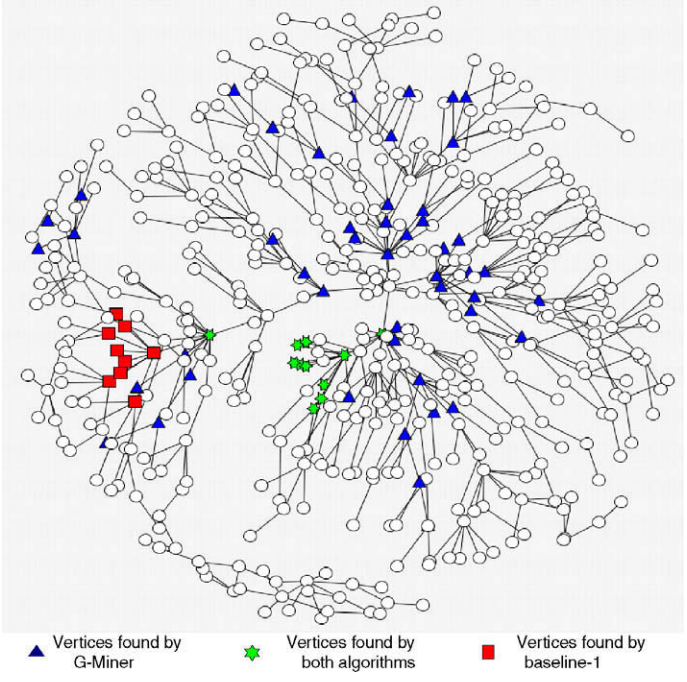

Fig. 24. The coverage of the top 10 frequent multi-edge patterns extracted by G-Miner compared with that of baseline-1 on graph V2E1N500L20A05.

5.1.4. Example G-Patterns

We use the G-Patterns found in the PPI data set to illustrate the type of subgraphs that G-Miner can discover. Recall that the PPI data set is a network recording the interactions between proteins. Therefore, the G-Patterns found are the common structures among proteins in the interaction network, which can be used to analyze these proteins.

Fig. 28 shows two representative G-Patterns found in the PPI data set. It can be immediately seen that vertices of the found G-Patterns belonging to the same functional class, same as the observations of the previous experiments that proteins of a frequent pattern perform the same biological function [25]. Furthermore, we see that proteins of *Cell Transport* function

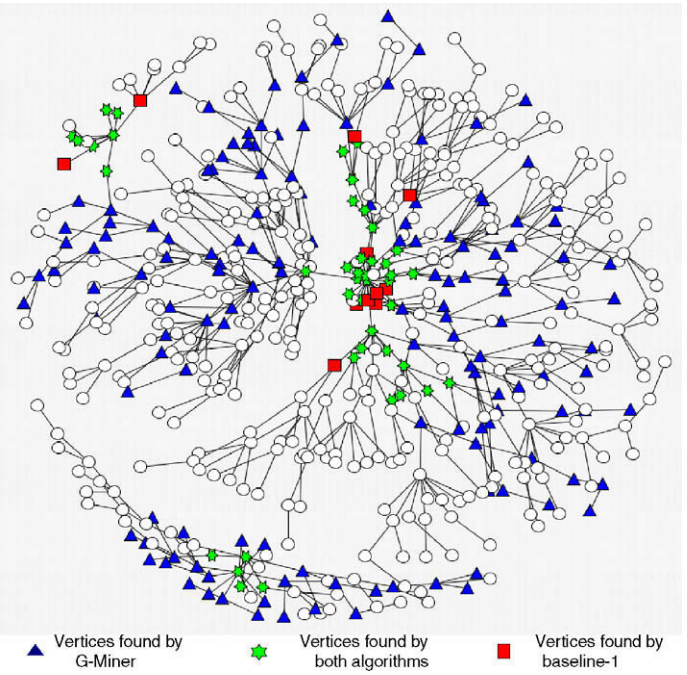

Fig. 25. The coverage of the top 10 frequent multi-edge patterns extracted by G-Miner compared with that of baseline-1 on graph V2E1N550L10A65.

have a line structure (see Pattern A) while proteins of *Cell cycle and DNA processing* have a tree structure (see Pattern B). Therefore, the two patterns can be used to help classify unknown proteins of the two categories.

5.1.5. Performance comparison with existing algorithms

Comparison with SEuS. The SEuS algorithm [38] is designed for mining frequent subgraphs in a single labeled directed graph. Particularly, it first builds a data structure called *data summary*, which is a compressed representation of the input graph. Then, frequent subgraphs are found based on the data summary, in which the support values of the patterns are computed as their exact number of instances in the input graph, same as baseline-1. The advantage of such an approach is that the most frequent patterns in the data set can be easily found. However, it still needs to explore the whole candidate space for finding frequent patterns with support values no less than a predefined threshold.

Here, we only compare the computational performance of G-Miner with that of SEuS on the credit data set, as the other real-world data sets either cannot be treated as a directed graph or be processed by SEuS. Given the same minimum support threshold $\theta = 100$, 11,696 patterns mined by G-Miner using 47 seconds, that is 36 frequent subgraphs per second. However, as SEuS sets a beam width parameter[17] to limit the number of possible candidates to be tested, only 5 frequent graph patterns found by SEuS with 2.1 seconds, that is 2.4 frequent patterns per second. We can see G-Miner runs much faster than SEuS for mining this graph.

Comparison with SUBDUE. Different from G-Miner or SEuS which find frequent subgraphs in a single labeled graph, SUBDUE [10,39] is to find subgraphs that can compress a single graph according to the minimum description length principle. Specifically, SUBDUE searches for patterns Ps in the graph G that minimize

$$I(P) + I(G|P),$$

where $I(P)$ represents the number of bits required to encode the pattern P and $I(G|P)$ is the number of bits required to encode G with respect to P. Such a requirement makes the patterns found by SUBDUE may not be the most frequent patterns. Therefore, we cannot directly compare G-Miner with SUBDUE because of the inherent difference between the two algorithms.

Here, we only evaluate the computational performance of the SUBDUE system on the first six real-world data sets. Because SUBDUE employs a heuristic beam search to keep limited candidates for further exploration, only a few patterns are mined from these data sets.[18] The results are given in Table 6. We can see SUBDUE generally requires more time than G-Miner for mining these real-world data sets. For the aviation data set, the program even does not finish the computation after spending one day. Such results clearly indicate the computational inefficiency of SUBDUE for processing large graphs.

Comparison with gSpan and FSG. One of the reasons why the problem of mining a single labeled graph is more general and applicable is that its solution can be used for mining frequent subgraphs in a set of labeled graphs directly. For demonstration,

[17] The default beam width in SEuS is 5 for version 1.0 and we cannot set a larger value, which will make the program crash.
[18] The default beam width is 4 for SUBDUE version 5.2.1.

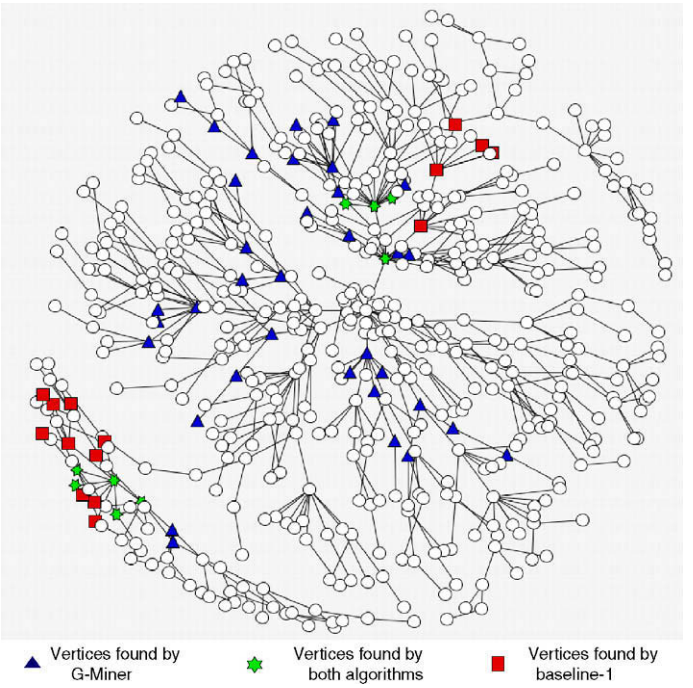

Fig. 26. The coverage of the top 10 frequent multi-edge patterns extracted by G-Miner compared with that of baseline-1 on graph V2E1N550L100A65.

Table 5
Experimental results of the G-Miner on the real-world data sets compared with those of baseline-1 and baseline-2.

Data set	θ	G-Miner		Baseline-1		Baseline-2	
		Runtime	# Patterns	Runtime	# Patterns	Runtime	# Patterns
Credit	50	155.2	73,992	–	–	432.6	73,992
	100	33.4	11,696	–	–	217.9	11,696
Aviation	1500	80.7	5231	–	–	88,623	5231
	2,000	16.6	843	–	–	24,988	843
Citation50_15	10	2.1	1099	–	–	2.9	1140
	20	1.1	260	–	–	1.1	260
Citation50_20	30	–	–	–	–	–	–
	40	1.45	108	–	–	–	–
Citation50_25	70	1.57	65	–	–	–	–
	80	1.34	53	–	–	–	–
PPI	250	–	–	–	–	–	–
	275	286.5	12	–	–	–	–

we employ the G-Miner to find frequent subgraphs from the chemical data set, where gSpan[19] and FSG[20] are used for comparison.

Note that we modify the setting of the G-Miner algorithm by forming reachable based instance graphs instead of edge-disjoint based instance graphs in this comparison. That is, an edge will be added to two vertices of the instance graph if their corresponding instances are reachable through certain paths in the input graph. In this case, a pattern's G-Measure value can actually be computed by counting the number of disconnected components in the instance graph. G-Miner's output will thus be the same as those of gSpan and FSG.

The experimental results of G-Miner, gSpan, and FSG, in terms of the runtime (in seconds) and the number of frequent subgraph patterns found on the chemical data set, are given in Table 7. The reasons why G-Miner runs slowly may be attributed to the difference between JAVA and C/C++ for computation and the time required for identifying whether two instances are reachable in the input graph. Nevertheless, we can see G-Miner extracts the same number of frequent subgraphs from the chemical data set as gSpan and FSG.

[19] http://www.xifengyan.net/software/gSpan.htm.
[20] http://glaros.dtc.umn.edu/gkhome/pafi/overview.

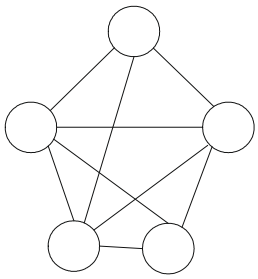

Fig. 27. An example instance graph on which different support values are obtained by using the G-Measure and the MIS based support measure.

Table 6
The performance of SUBDUE on the real-world data sets.

Data set	Number of patterns	Runtime (second)	Frequency of found patterns		
Credit	3	65.8	341	395	253
Aviation	3	–	–	–	–
Citation50_25	3	89.26	6	5	2
Citation50_20	3	61.6	10	9	8
Citation50_15	3	35.29	10	9	9
PPI	3	4104	96	112	113

Table 7
The experimental results of G-Miner, gSpan, and FSG on the chemical data set.

θ	G-Miner		gSpan		FSG	
	Runtime	# Pattern	Runtime	# Pattern	Runtime	# Pattern
50	7.5	434	0.2	434	0.6	434
100	5.8	71	0.1	71	0.2	71
150	5.6	58	0.1	58	0.2	58

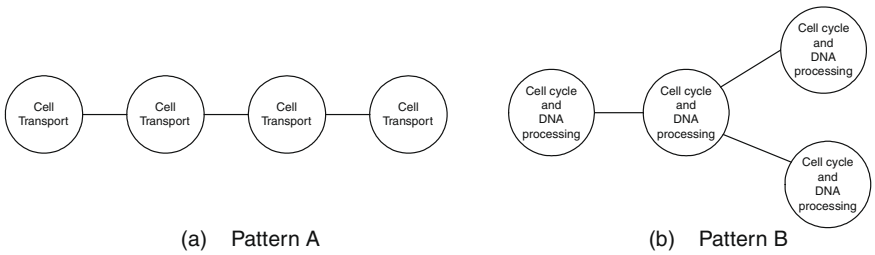

Fig. 28. Two example G-Patterns discovered by G-Miner from the PPI data set.

5.2. The application of G-Patterns for text categorization

Here, we present an application of G-Patterns for classifying text documents.

5.2.1. Basic principle

Different from the common text classification methods that only utilize the document content, approaches of classifying hypertext could further use the links between the documents to tackle this problem. For example, a *relaxation labeling* problem [21,22] is introduced that assigns a particular class label to an un-classified document by considering both the content of the document and the labels of its neighbors. An illustration of such a problem is given in Fig. 29, where we have known certain documents' labels (i.e., nodes presented with rectangle and triangle) and want to infer the left documents' labels (i.e., nodes marked with ?).

The basic solution of the *relaxation labeling* problem is presented as follows. Given a document d, the probability of assigning a label c to d is computed by

$$\Phi_{c,d} = Pr[\lambda(d) = c|\tau(d), \lambda(d_1), \ldots, \lambda(d_n)], \quad (5)$$

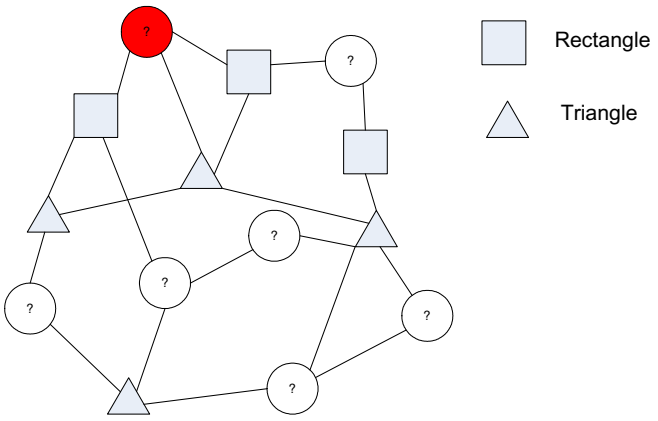

Fig. 29. An illustration of the relaxation labeling problem.

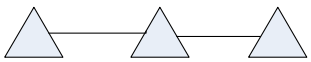

Fig. 30. A sample G-Pattern used to label documents.

where $\Phi_{c,d}$ is the probability of assigning c to d, $\lambda(d)$ represents the label of d, $\tau(d)$ represents the content of d, and d_1 through d_n are the documents linked to d in the data set. As there is no direct dependence between the content of a document and the labels of its neighbors, we can simplify the solution as:

$$\Phi_{c,d} = Pr[\lambda(d) = c|\tau(d)] \cdot Pr[\lambda(d) = c|\lambda(d_1), \cdots, \lambda(d_n)]. \tag{6}$$

Finally, a document d will be assigned a label c^* if $\Phi_{c^*,d}$ is the maximal value.

Here, we propose using G-Patterns to resolve this *relaxation labeling* problem. Specifically, we assign the label to an unknown node based on the possible G-Patterns it matches. For example, the unknown node marked with red color in Fig. 29 will be assigned label "rectangle" by the existing approach as most of its neighbors are labeled with "rectangle". However, as this node could satisfy the G-Pattern shown in Fig. 30, it may be assigned the label *triangle* as well.

The G-Pattern based approach of the *relaxation labeling* problem can be presented as:

$$\Phi_{c,d} = Pr[\lambda(d) = c|\tau(d)] \cdot Pr[\lambda(d) = c|\lambda(P_1(d)), \ldots, \lambda(P_n(d))], \tag{7}$$

where $P_n(d)$ represents a G-Pattern P_n the document d matches. The document d will be assigned a label c^* if $\Phi_{c^*,d}$ gets the maximal value.

5.2.2. Results

We conduct experiments to evaluate the performance of the G-Pattern based solution for the *relaxation labeling* problem. For evaluation, we would compare the performance of labeling documents using Eq. (7) with that of using Eq. (6). However, as the content of the documents would not affect this evaluation, we simplify the evaluation as the comparison between

$$\Phi_{c,d} = Pr[\lambda(d) = c|\lambda(d_1), \ldots, \lambda(d_n)] \tag{8}$$

and

$$\Phi_{c,d} = Pr[\lambda(d) = c|\lambda(P_1(d)), \ldots, \lambda(P_n(d))]. \tag{9}$$

The Wikipedia[21] data set is used for this experiment, which contains 5360 documents crawled from the Wikipedia dump file. These documents have been classified into 7 classes, including "Politics", "Computer Science", "Physics", "Chemistry", "Biology", "Mathematics", and "Geography". Because the graph of the original data set is very dense, containing 83,956 links, we removed links if the content similarity of their linked documents is larger than 0.05 to compress the graph. Finally, there are 4727 links kept for the experiment.

The details of this experiment are as follows: First, a set of documents are selected from the data set as the testing documents. Then, G-Patterns for estimating $Pr[\lambda(d) = c|\lambda(P_1(d)), \ldots, \lambda(P_n(d))]$ and the transition matrix for estimating $Pr[\lambda(d) = c|\lambda(d_1), \ldots, \lambda(d_n)]$ are built based on the graph formed with the left documents. Finally, these testing documents are labeled with Eqs. (8) and (9), respectively.

[21] http://www.mpi-inf.mpg.de/angelova/.

Table 8
The experimental results of labeling the 200 testing documents using Eq. (8) compared with that of using Eq. (9).

Method	Eq. (8)	Eq. (9)
Precision	0.53	0.61
F-Measure	0.69	0.75

Two hundred documents that have at least one link with other documents are selected for testing. The performance of the two methods is evaluated by the precision and the F-measure of the testing documents labeled, which is presented in Table 8. We can see the G-Pattern based approach indeed increases the performance of classifying documents. The improvement in precision obtained is 15% (from 0.53 to 0.61) and the improvement in F-Measure obtained is 7%. Such results validate our approach of using G-Patterns to resolve the *relaxation labeling* problem.

6. Related work

Given the population of the graph structure data, a large number of graph mining algorithms have been developed for analyzing and managing such data. Based on the difference of the inputs to the algorithms, we can classify the existing graph mining algorithms into two categories, namely mining frequent/important patterns in a set of labeled graphs and mining frequent/important patterns in a single labeled graph. The G-Miner algorithm presented in this paper belongs to the second category.

Between the two classes of algorithms, algorithms belonging to the first category [1–7] are more mature. Most existing graph mining algorithms are of this category. These algorithms can efficiently and effectively discover all the frequent subgraphs in the data and be scaled to large data sets. However, algorithms of the second category have received much less attention, despite the fact that this problem is more general and applicable. A simple illustration for this point is that algorithms that mine a single labeled graph can be directly used to find subgraphs in a set of labeled graphs. However, algorithms that mine in a collection of graphs have to be modified before working on a single labeled graph.

Besides the G-Miner algorithm, there are only a few attempts for finding frequent/important subgraphs in a single labeled graph [10,40,28,27,38]. As a major difference, G-Miner targets on finding globally distributed frequent subgraphs while these existing mining algorithms usually lead to the appearance of the identified frequent subgraphs regionally in the input graph. In addition to this key difference, we compare and contrast some algorithmic ideas of the related mining algorithms as follows.

The SUBDUE system [10,39] is a well-known algorithm which selects qualified patterns for compressing the input graph under the minimum description length principle. Specifically, SUBDUE searches for patterns Ps in the input graph G that minimize

$$I(P) + I(G|P),$$

where $I(P)$ represents the number of bits required to encode P and $I(G|P)$ is the number of bits required to encode G with respect to P. This requirement makes the patterns found by SUBDUE may not be the most frequent patterns in the graph. Furthermore, to avoid the problem of exploring the whole search space, a heuristic beam search is employed to narrow the search space. Only limited candidates are kept for further exploration. As a result, SUBDUE cannot find the complete set of important/frequent subgraphs in the input graph. The similar approach is adopted by GBI [40] for finding typical subgraphs in a single labeled graph.

The SEuS algorithm [38] aims at finding frequent subgraphs in a directed labeled graph. In particular, SEuS first uses a data structure called *data summary* to construct a compressed representation of the input graph, similar to DataGuides [41] for semi-structured data. Then, the frequencies of the graph patterns are estimated based on the built *data summary*. SEuS can quickly find the most frequent patterns in the input graph. However, it still needs to explore the whole search space for finding all the frequent patterns with support value not less than a predefined threshold. As the authors indicate [38], the SEuS algorithm is not efficient if the input graph contains a large number of frequent subgraphs with low frequency and is useful only when there are a few frequent subgraphs with high frequency in the input graph. This limitation greatly affects its ability for mining G-Patterns, since G-Patterns are normally with relatively low frequencies.

Given the fact that the standard support measure does not guarantee the downward closure property for finding frequent subgraphs in a single labeled graph, Vanetik et al. [28] propose a downward closure property holding measure instead. Specifically, this proposed measure computes the maximum independent set (MIS) of a graph pattern's instance graph. Only patterns with MIS size not less than the minimum support threshold are selected. Because of the downward closure property, algorithms implemented with this support measure may quickly find a set of frequent subgraphs. However, computing the maximum independent set of a graph has proved to be a NP-hard problem [42]. Algorithms with this support measure can even perform worse than those with the standard approach.[22] Also, as shown in [27,11], algorithms implemented with this measure are only efficient for mining large sparse graphs due to the high cost of computing the maximum independent set. As a result, the new measure is not a general solution for quickly mining a single labeled graph. Note that although some variants of Vanetik's support measure are proposed [43,44], these new measures only reduce the number of MIS to be computed and do not change the problem substantially.

[22] If the maximal edge number of the frequent pattern to be found is also predefined, the algorithm with Vanetik's support measure may run slower than algorithms with the standard support measures on certain graphs, e.g., the credit data set we have used in the experiments.

7. Conclusion

In this paper, we have formalized the problem of finding G-Patterns, a type of globally distributed frequent subgraphs, in a single labeled graph. As a measure of association for graph data, G-Measure is proposed to find G-Patterns. It is consequently exploited to develop a G-Miner algorithm for efficiently identifying G-Patterns in a single labelled graph. Experimental results from both synthetic and real-world data sets demonstrated the efficacy of the G-Miner for discovering G-Patterns. Finally, we present an application of the G-Patterns.

References

[1] A. Inokuchi, T. Washio, H. Motoda, An apriori-based algorithm for mining frequent substructures from graph data, in: Principles of Data Mining and Knowledge Discovery, 4th European Conference (PKDD 2000), 2000, pp. 13–23.
[2] M. Kuramochi, G. Karypis, Frequent subgraph discovery, in: Proceedings of the 2001 IEEE International Conference on Data Mining (ICDM 2001), 2001, pp. 313–320.
[3] X. Yan, J. Han, gspan: Gaph-based substructure pattern mining, in: Proceedings of the 2002 IEEE International Conference on Data Mining (ICDM 2002), 2002, pp. 721–724.
[4] C. Wang, W. Wang, J. Pei, Y. Zhu, B. Shi, Scalable mining of large disk-based graph databases, in: Proceedings of the Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD 2004), 2004, pp. 316–325.
[5] J. Huan, W. Wang, J. Prins, Efficient mining of frequent subgraphs in the presence of isomorphism, in: Proceedings of the Third IEEE International Conference on Data Mining (ICDM 2003), 2003, pp. 549–552.
[6] J. Wang, W. Hsu, M.-L. Lee, C. Sheng, A partition-based approach to graph mining, in: Proceedings of the 22nd International Conference on Data Engineering (ICDE 2006), 2006, p. 74.
[7] M. Koyutürk, Y. Kim, S. Subramaniam, W. Szpankowski, A. Grama, Detecting conserved interaction patterns in biological networks, J. Comput. Biol. 13 (7) (2006) 1299–1322.
[8] S. Nijssen, J.N. Kok, A quickstart in frequent structure mining can make a difference, in: KDD, 2004, pp. 647–652.
[9] M.J. Zaki, M. Peters, I. Assent, T. Seidl, Clicks: An effective algorithm for mining subspace clusters in categorical datasets, Data Knowl. Eng. 60 (1) (2007) 51–70.
[10] L.B. Holder, D.J. Cook, S. Djoko, Substucture discovery in the subdue system, in: Knowledge Discovery in Databases: Papers from the 1994 AAAI Workshop, 1994, pp. 169–180.
[11] M. Kuramochi, G. Karypis, Finding frequent patterns in a large sparse graph, Data Min. Knowl. Discov. 11 (3) (2005) 243–271.
[12] M.J. Zaki, Efficiently mining frequent trees in a forest, in: Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD 2002), 2002, pp. 71–80.
[13] T.-S. Chen, S.-C. Hsu, Mining frequent tree-like patterns in large datasets, Data Knowl. Eng. 62 (1) (2007) 65–83.
[14] Q. Zhao, L. Chen, S.S. Bhowmick, S.K. Madria, Xml structural delta mining: issues and challenges, Data Knowl. Eng. 59 (3) (2006) 627–651.
[15] S. Zhang, J.T.-L. Wang, Discovering frequent agreement subtrees from phylogenetic data, IEEE Trans. Knowl. Data Eng. 20 (1) (2008) 68–82.
[16] A. Termier, M.-C. Rousset, M. Sebag, K. Ohara, T. Washio, H. Motoda, Dryadeparent, an efficient and robust closed attribute tree mining algorithm, IEEE Trans. Knowl. Data Eng. 20 (3) (2008) 300–320.
[17] A. Deutsch, M.F. Fernandez, D. Suciu, Storing semistructured data with stored, in: Proceedings ACM SIGMOD International Conference on Management of Data (SIGMOD 1999), 1999, pp. 431–442.
[18] X. Yan, P.S. Yu, J. Han, Graph indexing: a frequent structure-based approach, in: Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD 2004), 2004, pp. 335–346.
[19] C. Domshlak, R.I. Brafman, S.E. Shimony, Preference-based configuration of web page content, in: Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI 2001), 2001, pp. 1451–1456.
[20] Y. Fu, M. Creado, C. Ju, Reorganizing web sites based on user access patterns, in: Proceedings of the 2001 ACM CIKM International Conference on Information and Knowledge Management, 2001, pp. 583–585.
[21] S. Chakrabarti, B. Dom, P. Indyk, Enhanced hypertext categorization using hyperlinks, in: Proceedings of ACM SIGMOD International Conference on Management of Data (SIGMOD 1998), 1998, pp. 307–318.
[22] R. Angelova, G. Weikum, Graph-based text classification: learn from your neighbors, in: Proceedings of the 29th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 2006), 2006, pp. 485–492.
[23] L. Guo, J. Shanmugasundaram, G. Yona, Topology search over biological databases, in: Proceedings of the 23rd International Conference on Data Engineering (ICDE 2007), 2007, pp. 556–565.
[24] S. Su, D.J. Cook, L.B. Holder, Knowledge discovery in molecular biology: identifying structural regularities in proteins, Intell. Data Anal. 3 (6) (1999) 413–436.
[25] H. Xiong, X. He, C.H.Q. Ding, Y. Zhang, V. Kumar, S.R. Holbrook, Identification of functional modules in protein complexes via hyperclique pattern discovery, in: Biocomputing 2005, Proceedings of the Pacific Symposium, 2005.
[26] M. Kirac, G. Özsoyoglu, Protein function prediction based on patterns in biological networks, in: Research in Computational Molecular Biology, 12th Annual International Conference, RECOMB 2008, Singapore, March 30–April 2, 2008. Proceedings, 2008, pp. 197–213.
[27] M. Kuramochi, G. Karypis, Finding frequent patterns in a large sparse graph, in: Proceedings of the Fourth SIAM International Conference on Data Mining (SDM 2004), 2004.
[28] N. Vanetik, E. Gudes, S.E. Shimony, Computing frequent graph patterns from semistructured data, in: Proceedings of the 2002 IEEE International Conference on Data Mining (ICDM 2002), 2002, pp. 458–465.
[29] N. Vanetik, S.E. Shimony, E. Gudes, Support measures for graph mining, Data Min. Knowl. Discov. 13 (2006) 243–260.
[30] H. Xiong, P.-N. Tan, V. Kumar, Mining strong affinity association patterns in data sets with skewed support distribution, in: Proceedings of the Third IEEE International Conference on Data Mining (ICDM 2003), 2003, pp. 387–394.
[31] A. Abou-Rjeili, G. Karypis, Multilevel algorithms for partitioning power-law graphs, in: 20th International Parallel and Distributed Processing Symposium (IPDPS 2006), 2006.
[32] M. Stoer, F. Wagner, A simple min-cut algorithm, J. ACM 44 (4) (1997) 585–591.
[33] S. Wang, J.M. Siskind, Image segmentation with ratio cut, IEEE Trans. Pattern Anal. Mach. Intell. 25 (6) (2003) 675–690.
[34] A. Clauset, M.E.J. Newman, C. Moore, Finding community structure in very large networks, Phys. Rev. E 70 (2004) 066111.
[35] A.-L. Barabasi, R. Albert, Emergence of scaling in random networks, Science 286 (5439) (1999) 509–512.
[36] A. Srinivasan, R.D. King, S. Muggleton, M.J.E. Sternberg, The predictive toxicology evaluation challenge, in: IJCAI (1), 1997, pp. 4–9.
[37] P.R.J. Östergård, A fast algorithm for the maximum clique problem, Discrete Appl. Math. 120 (1–3) (2002) 197–207.
[38] S. Ghazizadeh, S.S. Chawathe, Seus: structure extraction using summaries, in: Discovery Science, Fifth International Conference, DS 2002, 2002, pp. 71–85.
[39] D.J. Cook, L.B. Holder, Substructure discovery using minimum description length and background knowledge, J. Artif. Intell. Res. (JAIR) 1 (1994) 231–255.

[40] K. Yoshida, H. Motoda, Clip: concept learning from inference patterns, Artif. Intell. 75 (1) (1995) 63–92.
[41] R. Goldman, J. Widom, Dataguides: Enabling query formulation and optimization in semistructured databases, in: Proceedings of 23rd International Conference on Very Large Data Bases (VLDB1997), 1997, pp. 436–445.
[42] M.R. Garey, D.S. Johnson, Computer and Intractability: A Guide to the Theory of NP-Completeness, W.H. Freeman, 1979.
[43] M. Fiedler, C. Borgelt, Support computation for mining frequent subgraphs in a single graph, in: Mining and Learning with Graphs (MLG 2007), Proceedings, 2007.
[44] M. Wörlein, A. Dreweke, T. Meinl, I. Fischer, M. Philippsen, Edger: the embedding-based graph miner, in: Mining and Learning with Graphs (MLG 2006), Proceedings, 2006.

Xing Jiang is currently a Ph.D. Student in School of Computer Engineering, Nanyang Technological University, Singapore. He received the B.M. degree in Information Management and Information System and the B.E. degree in Computer Science and Technology from the University of Science and Technology of China, China. His research interests include text mining, information retrieval, and Semantic Web.

Hui Xiong is currently an Associate Professor in the Management Science and Information Systems department at Rutgers, the State University of New Jersey. He received the B.E. degree in Automation from the University of Science and Technology of China, China, the M.S. degree in Computer Science from the National University of Singapore, Singapore, and the Ph.D. degree in Computer Science from the University of Minnesota, USA. His general area of research is data and knowledge engineering, with a focus on developing effective and efficient data analysis techniques for emerging data intensive applications. He has published over 60 technical papers in peer-reviewed journals and conference proceedings. He is a co-editor of Clustering and Information Retrieval (Kluwer Academic Publishers, 2003) and a co-Editor-in-Chief of Encyclopedia of GIS (Springer, 2008). He is an Associate Editor of the Knowledge and Information Systems journal and has served regularly in the organization committees and the program committees of a number of international conferences and workshops. He was the recipient of the 2008 IBM ESA Innovation Award, the 2007 Junior Faculty Teaching Excellence Award and the 2008 Junior Faculty Research Award at the Rutgers Business School. He is a senior member of the IEEE, and a member of the ACM.

Chen Wang has been working at IBM Research China since 2005. He has specialized in RDF storage, index and querying, cloud computing based analytics platform, graph index, frequent graph mining, graph clustering, master data management, database, semantic web, and healthcare applications. He received his Ph.D. degree in computer science from Fudan University in 2005 when he won the China Computer Federation best doctoral dissertation award. He has published more than 20 publications in the areas of database, semantic web, and data mining, including the premium conferences like SIGMOD, VLDB, ICDE, WWW, SIGKDD, and ISWC.

Ah-Hwee Tan received the B.S. degree (First Class Honors) and the M.S. degree in Computer and Information Science from the National University of Singapore and the Ph.D. degree in Cognitive and Neural Systems from Boston University. He is currently an Associate Professor and Head, Division of Information Systems at the School of Computer Engineering, Nanyang Technological University. He was also the founding Director of the Emerging Research Laboratory, a research centre for incubating interdisciplinary research initiatives. Prior to joining NTU, he was a Research Manager at the A*STAR Institute for Infocomm Research (I2R), responsible for the Text Mining and Intelligent Agents programmes. His current research interests include biologically-inspired cognitive systems, information mining, machine learning, knowledge discovery, document analysis, and intelligent agents. Dr. Tan holds several patents and has published over eighty technical papers in books, international journals, and conferences. He is an editorial board member of Applied Intelligence, a senior member of IEEE, and a member of ACM.