International Journal of Grid Distribution Computing
Vol.7, no.3 (2014), pp.233-250
http://dx.doi.org/10.14257/ijgdc.2014.7.3.24

Recommending Optimal API Orchestration with Mining Frequent Mashup Patterns

Dunlu Peng[1], Lei Xie[1], Duan Kai[1] and Feitian Li[2]

[1]Shanghai Key Lab of Modern Optical System,
School of Optical-Electrical and Computer Engineering,
University of Shanghai for Science and Technology, Shanghai, 200093, China
[2]Shanghai BroadText Iswind Software Co. Ltd, Shanghai, 200072, China
dlpeng@usst.edu.cn; leixie2007@126.com; duankai@yeah.net,
lifeitian@broadtext.com.cn

Abstract

As more and more organizations publish their data or services through Open APIs on the Internet, mashup applications have captured a lot of attention in recent years. However, as the number and categories of Open APIs grow rapidly, efficiently creating optimal mashup applications becomes a crucial issue for making the technology of mashup more applicable. In this work, we present a Mashup Directed Orchestration Model (MDOM) to depict the mashup patterns with a graph-based model on the basis of mashup orientation. According to the features of MDOM, by taking advantage of the theory of directed graph and the strategies used in the algorithms for discovering frequent sub-graphs, an algorithm named as FSOMM is presented to efficiently mine the frequent orchestration patterns hidden in the MDOMs. These discovered frequent orchestration patterns provide us a promising way to create optimal mashup applications. In addition, the performance of the proposed approach is verified by implementing a series of experiments on both synthetic and real datasets.

Keywords: *Mahsup application, API orchestration, Frequent mashup pattern, Web data integration*

1. Introduction

In the past decades, Web-related technologies have been developed rapidly; especially wireless computing has made Web applications everywhere. Open APIs encapsulate Web services as a series of interfaces that can be identified and accessed by computer, and these Open APIs are published on the Internet that makes them easier to be invoked by third-party applications. Different from the traditional Web pages which are fit for browsing, Open APIs provide a convenient way for data exchanging between the applications of service providers' and those of the users' on the Internet. Users compose Open APIs from different websites to form new applications in order to fulfil their business requirements. This makes the so-called mashup applications come into being and advance quickly on the Web in recent years.

The term *mashup*, originally comes from the field of pop music, where people seamlessly combine music from one song with the vocal track from another—thereby mashing them together to create something new[1]. In Web applications, there is still no unified definition for mashup, even there have been a lot of typical mashup applications. For example, Housingmaps.com combines Google Maps API with those provided by Craigslist; a company

[1] http://en.wikipedia.org/wiki/Mashup_(web_application_hybrid)

ISSN: 2005-4262 IJGDC

offers information for house renting. Since it went online, mashup application has gained much attention from both industrial and academic fields. In most cases, it is suggested that Web mashup be a process that combines two or more different Open APIs to form a new Web application with a certain model in the literature [1-3]. Nowadays, mashup service has been considered as an important way to enhancing their marketing competitiveness by a large number of companies, such as Flickr, Amazon, Yahoo, etc. In addition, mashup technology has been widely used in lots of areas, such as network maps, videos, images, shopping searching, news, microblog services and so on. As early as 2008, Gartner reported in their Internet Report that mashup is becoming one of the most noteworthy technologies on the Internet.

As more and more organizations publish their core business as Open APIs on the Web to provide users better services, both the number and categories of Open APIs have increased at great rapidity in recent years. For example, by January 14, 2014, there are more than 10000 Open APIs and 7000 mashup applications registered at the Programmable.com, and they still keep quickly increasing. The abundant Open APIs provide a realistic basis for the development of mashup services. However, it also brings big challenges on selecting proper Open APIs for building efficient mashup applications, selecting effective mashup model and simplifying the constructing process of mashup application for users because of the large number and varieties of Open APIs available on the Web.

In view of the above challenges, researchers have studied mashup technology from different perspectives. O. Greenshpan proposed an automated model for building mashup applications [4]. That model tries to match the user's selected Open APIs with the ones which have already been in the mashup patterns. If there is a pattern that contains the user's selected Open APIs, it will be recommended to the user. In [5], the authors presented a platform to support users to create, use and manage mashup applications. On this platform, users can easily build a complete mashup application even without having sufficient knowledge of Web. During the process of building a mashup application, the similarity between user's behaviors has been fully considered for creating the applications in an efficient way. It also takes advantage of the relationships between Open APIs built by experienced users. These relationships are often utilized to eliminate the inconsistency and incompleteness among the Open APIs' augments. These studies show that the relationships between Open APIs are very valuable for automatically building mashup applications. Unlike the previous studies, in this work, after deeply analyzing existing Open API orchestrations of mashup applications, we try to facilitate users to choose appropriate Open APIs and the composite models for building optimal mashup applications by mining the frequent mashup patterns hidden in the existing mashup applications.

During the study of Open API combination mode, it not only needs to learn the co-occurrence probabilities of different Open APIs, but also needs to give a deep discussion on the logic of combination between them. In order to accurately identify the combination relationship between Open APIs, this paper proposes a model named as Mashup Directed Orchestration Model (or MDOM) to represent the relationship between Open APIs contained in a mashup application. Namely, apart from depicting the Open APIs appeared in a certain mashup application, MDOM describes the combination sequence of the Open APIs. The sequence is called as the mashup pattern or orchestration pattern of the application. Besides, we propose an algorithm——FSOMM (Frequent Sub-graph Orchestration Mining of Mashup) to discover frequent mashup patterns hidden in the MDOMs. The discovered frequent patterns can be recommended in the future for making optimal mashup applications.

The main contributions of this paper are list as follows:

- After analyzing the possible combination modes between Open APIs, which include serial, parallel and hybrid modes, we present a model, named as Mashup Directed Orchestration Model (MDOM), to represent relationships between Open APIs in mashup applications with directed graphs. All the combination modes can be described with MDOM.
- Through investigating the real mashup applications, we find that the vertices in most of the MDOMs are unique and less than 4. Taking this feature into consideration, an algorithm—FSOMM is proposed for mining frequent mashup patterns hidden in MDOM collection in an efficient way. The discovered frequent mashup patterns can be used for creating optimal mashup applications.
- To verify the performance of the proposed approach, we implement it with experiments on both the synthetic dataset and real dataset. The synthetic dataset we generated it with ourselves developed tool, while the real dataset is the corresponding information about the real mashups downloaded from ProgrammableWeb.com. The experimental results demonstrate that our approach can discover frequent mashup patterns with higher performance compared with other existing sug-graph mining algorithms, such as gSpan and FSG.

The rest of the paper is organized as follows. Section 2 presents the related work; Section 3 discusses the combining modes of Open APIs, and Mashup Directed Orchestration Model – MDOM is also proposed concretely in the same section. In Section 4, we describe how to mining frequent mashup patterns hidden in MDOMs with our algorithm FSOMM. Section 5 analyzes the performance of FSOMM with experiments. Finally, we conclude our work in Section 6.

2. Related Work

In recent years, researchers have paid much attention to finding the optimal patterns in mashup applications, and a lot of work has been done on analyzing the existing mashup applications registered at Programmable Web.com. Cao, *et al.*, recommended optimal data sources to users for building up mashup applications more quickly by using project-based top-N algorithm [6]. Based on analyzing the similarity of user's behavior, Wang, *et al.*, proved that users are willing to employ the Open APIs which are used more frequently [7]. In [8] the authors constructed 13 kinds of diagrams to describe homogeneous or heterogeneous networks of mashups, Open APIs and tags. These diagrams are useful for classification and recommendation of mashup applications. Some other studies focus on the fusion analysis of data layers of mashup [9], performance optimization for data flow in mashup applications [10], mashup of UI components [11, 12], semantics-based Open API recommendation [13], mashup service model [14] , automated recommendation of Open APIs during construction of mashup applications [15-18] and so on. These studies perfect the mashup technology in varies degrees and improve the development of mashup applications.

In this work, we try to apply the theory of graph model to optimize mashup orchestration patterns. A graph is a data structure that describes complex constructions and the relationship between objects in the structures. Graphs are widely used in compound classification [19], biological network analysis [20], and Web fields [21]. In a variety of graph patterns, frequent sub-graphs are often utilized to describe the characteristics of the graphs and as a basis for graph classification [22]. Therefore, mining frequent sub-graphs have been highlighted in the research area for many years. Generally, there are two basic methods to mine sub-graphs: Apriori-based algorithms [23] and pattern-growth-based algorithms [24]. In Apriori-based algorithms, a k+1-edged sub-graph is generated by connecting two frequent k-edged sub-graphs with only one node (or edges) in difference, such as FSG [26] Pattern-growth-based algorithms recursively extend a frequent graph until all of the frequent sub-graphs nested in the graph are discovered, such as gSpan [27].

In this paper, we present MDOM to describe this relationship between Open APIs on the basis of analyzing the combinational relationship between Open APIs in mahsup applications. A MDOM is a directed graph in which vertices represent the Open APIs invoked in the mashup application, and the directed edges represent the API's at the start point executed prior to that at the end point. In terms of MDOM, a mashup application can be expressed as a directed graph. Thus, mining frequent mashup patterns can be treated as mining frequent sub-graphs concealed in a collection of directed graphs. By combining the advantages of the two methods for mining frequent sub-graphs, this paper proposes a special algorithm—— FSOMM to mine frequent mashup patterns.

1. Open API Orchestration Modes and MDOM

As we aforementioned, a mashup application employs one or more Open APIs to accomplish some certain tasks. In other words, Open APIs are the basic elements composing mashup applications. In this section, apart from studying the possible orchestration modes of Open APIs, we present MDOM to describe the relationship between Open APIs with directed graphs.

3.1. Possible Open API Orchestration Modes

As Open APIs are conveniently accessible and programmable to users, publishing data or services as Open APIs onto the Web becomes the mainstream way for service providers to share their information or services. For conciseness, in this paper, we assume that the data or services used in the mashups are obtained through Open APIs. In the previous sections, we discussed that a mashup application aggregates Open APIs from different sources, and each mashup should fulfill some specific requirements that demands the invoked Open APIs being arranged in a certain logic order. These logic orders are called as mashup patterns. Prior to further discussing mashup patterns, we need to look at the possible orchestration modes of Open APIs in mashup applications. The concept of Open API is described in Definition 1.

Definition 1: An ***Open API*** is accessible and programmable interface provided at a particular website. Users can access data or services through this interface while the internal implementation of the Open API is transparent to users.

Open API has been widespread on the Web, such as Flickr APIs, Google Maps APIs. Flickr is a photo-sharing site and its Open APIs allow users to access photo resources with authorized key. Google Maps APIs provide interfaces to access or customize maps supplied by Google.com. All the Open API can be accessed by the third-party applications on the Web.

Definition 2: A mashup application arranges multiple Open APIs with a certain model to accomplish a particular task. The orchestration of Open APIs in mashup application forms the ***mashup patterns***.

Definition 2 shows that a mashup pattern reflects the set of Open APIs composed in a certain mashup and is independent of the Open APIs in other mashups. Suppose there are three Open APIs A, B and C in Mashup M, and the orchestration mode of A, B and C construct the mashup pattern of M which is irrelevant to the mode of them in other mashup applications.

As described in Definition 1, each Open API provides an interface for accessing the data or services behind it. Even through the internal implementation is invisible to users; each interface offers input/output arguments and related description to help users to call the API correctly. Thus, formally, an Open API can be represented by a four-tuple {*des*, *inputs*, *outputs*, *funs*}, in which, *des* is the interface type (data or service), *inputs* is the list of input parameters, *outputs* is the list of output parameters and *funs* indicates the set of available calling operations of the Open API.

In this paper, we derive the possible orchestration modes of two Open APIs by taking advantage of the logic order deduced from the relationship between their inputs and outputs. As known to us, some Open APIs must be executed after others because they need the outputs of the latter as their inputs. Furthermore, in a certain mashup application, the orchestration mode of any two Open APIs is fixed, which means that the Open APIs should be executed one after another or in parallel, but never crosswise.

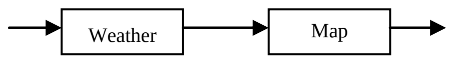
With the above definitions and assumptions, suppose there are two Open APIs *A* and *B* called by mahsup *M*. The sets of invoked operations of *A* and *B* are $OA=\{OA_1, OA_2,\dots,OA_n\}$ and $OB=\{OB_1, OB_2, \dots, OB_m\}$, respectively. The corresponding inputs and outputs are listed as $OA.ins=\{OA_1.in, OA_2.in,\dots,OA_n.in\}$, $OB.ins=\{OB_1.in,OB_2.in,\dots,OB_n.in\}$, $OA.outs=\{OA_1.\ out, OA_2.\ out, \dots, OA_n.\ out\}$ and $OB.\ outs=\{OB_1.\ out,OB_2.\ out,\dots,OB_n.\ out\}$. When composing together the two Open APIs, the following two rules need to be conformed:

r$_1$: If $OA.ins \cap OB.\ outs \neq \emptyset$ or $OA.outs \cap OB.ins \neq \emptyset$, then the operations of *A* are called after all those of *B* that are performed.

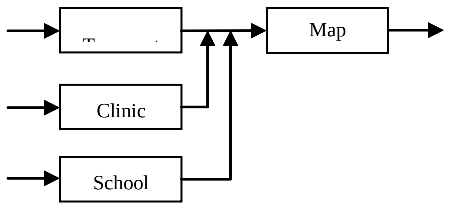
r$_2$: if $OA.ins \cap OB.\ outs = \emptyset$ and $OA.outs \cap OB.ins = \emptyset$, then the execution of the operations of *A* and those of *B* is independent and can be performed in any order except being overlapped.

The two rules indicate that there are two different orchestration modes for Open API *A* and *B*: serial orchestration or parallel orchestration. We define them as follows:

Definition 3: Suppose *A* and *B* are two Open APIs called in a mashup *M*. If *B* must be executed after *A* (or otherwise), then *A* and *B* are in **serial orchestration**, denoted by $B \rightarrow A$ (or $A \rightarrow B$ for the otherwise). Otherwise, if it need not run *A* and *B* in order, then *A* and *B* are in **parallel orchestration** in *M*, denoted by *A//B*.



(a) Serial Orchestration



(b) Parallel Orchestration

Figure 1. Orchestration Modes of Open APIs

Figure 1 depicts both of the orchestration modes with example mashup applications. In Figure 1(a), Weather and Map are two Open APIs which are organized in series orchestration mode. It illustrates, in that mashup application, Weather is called to return weather report as input for Map which is used to show weather information on the map. Figure1(b) brings us an instance of the parallel orchestration mode of Open APIs. In the application, School, Clinic and Transport could respectively return information about schools, clinics and transportation around a house after inputting its geographic position to the three APIs. However, there is no data exchanging among the three Open APIs, and also they need not be invoked in certain sequence. Namely, they are organized in parallel orchestration in the example mashup application.

3.2. Mashup Directed Orchestration Model - MDOM

Through analyzing more than 7000 mashup applications registered at ProgrammableWeb.com, we found that the orchestration of any two Open APIs in a same mashup application can be described with a serial or parallel mode. However, if a mashup application calls more than two Open APIs, the situation becomes complicated. Frequently, there is a hybrid mode which contains both of two orchestration modes in practice. Here, we develop a directed graph as a model to visually represent the orchestration modes of Open APIs contained in a mashup application. The graph is called Mashup Directed Orchestration Model (or MDOM).

Definition 4: The **MDOM** of a mashup application is a directed graph, whose vertices are the Open APIs invoked in the application, and the directed edge between two vertices represents the Open API at the source performed prior to the one at the end.

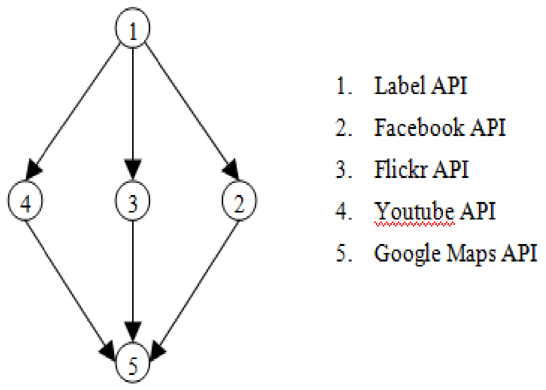


Figure 2. An Example of MDO

For simplicity, the Open APIs are labeled with a serial number as its ID in the graph. In a MDOM graph, if there is a path from Open API *A* to Open API *B*, it indicates that *B* is executed after *A*, which means the orchestration mode of *A* and *B* is serial and there exists data exchanging or logic calling between them. Otherwise, *A* and *B* is in parallel orchestration and there is no data exchanging or logic relationship between them. Figure 2 is an example of MDOM. As shown in the figure, Label API (①) and Facebook API (②) are in serial orchestration mode, while Facebook API, Flickr API (③) and Youtube API (④) are in parallel orchestration mode. All of the Open APIs together form a MDOM in a hybrid orchestration mode.

Based on the investigation of the real mashup applications registered at ProgrammableWeb.com, we summarize the following features of real MDOMs.

Diversity of vertices: A certain Open API can only appear once in a certain MDOM. This property guarantees that there are no two or more vertices with a same ID in a given MDOM.

Only one start vertex and one end vertex: a start vertex in an MDOM is a node which has zero in-degree and stands for the first invoked Open API in the mashup application. An end vertex is a node which has zero out-degree and the corresponding Open API is the last one called in the mashup application.

The number of vertices is generally 2~ 5: Statistics suggests that over 90% of the mashup applications combine 2~5 Open APIs in total. Figure 3 shows the distribution of the number of Open APIs invoked in mashup applications registered at ProgrammableWeb.com. Correspondingly, in our work, the number of vertices in a MDOM is considered to be from 2 to 5 by default.

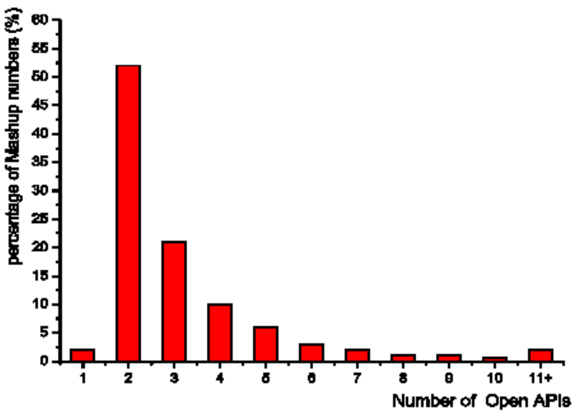


Figure 3. Distribution of the Number of Open APIs Invoked in Mashup Applications Registered at Programmable Web.com

2. Mining Mashup Frequent Patterns

In the previous section, we described the definition of MDOM and summarized the properties a MDOM has. Our frequent pattern mining algorithm, which is called FSOMM, is based on the definition and properties of MDOM.

2.1 Mashup Directed Orchestration Patterns

For convenience, each Open API is assigned an ID before modeling the application with MDOM. Each ID is unique across the Open API library. Thus, the formal description of MDOM, named as Mashup Directed Orchestration Pattern (MDOP), is defined as follows:

Definition 5: The **MDOP** of a mashup application is a directed graph $G=\{V(G),E(G),L(V(G))\}$, where $V(G)=\{API_1,API_2,\ldots, API_n\}$ is the collection of Open APIs invoked by the mashup application, and $E(G)=\{(API_i, API_j)|API_i, API_j\in V(G)\}$ is the set of directed edges demonstrating the orchestration relationship between two Open APIs. That is, if API i is called before API j, there will be an edge directed from API i to API j. Besides, the nodes and labels can be mapped as $L(V(G))=\{l(API_i)|\ API_i\in V(G)\}$, where $L(\ )$ is the label mapping function, and the size of graph G is denoted as $|E\ (G)|$.

In a practical application, the set of vertices is corresponding to the IDs of the Open APIs revoked by the mashup applications, and the label mapping function maps the Open APIs to their IDs. The values of the function have been given as the Open APIs are added into the library. For example, in the MDOP of the MDOM shown in Figure 2, the set of its vertices is {1-5} and the set of edges is {⟨1, 2⟩,⟨1, 3⟩, ⟨1, 4⟩, ⟨2, 5⟩, ⟨3,5⟩, ⟨4, 5⟩}, and 1-5 are the IDs for Label API, Facebook API, Flickr API, YouTube API and Google Maps API respectively and assigned to the Open APIs as they are registered successfully.

MDOP provides a formal expression for MDOM, however, in a large mashup application library， there always exists different formal expressions for a same orchestration relationship between Open APIs. For instance, in Figure 4, it depicts the MDOP of two different mashup applications M_1 and M_2. Although the formal expressions of M_1 and M_2 are not the same, their vertex set and edge set are equal, which are {1-4} and {⟨1, 2⟩, ⟨1, 3⟩, ⟨2, 4⟩, ⟨4, 3⟩}. Here, M_1 and M_2 are referred as an isomorphism of MDOP. Formally, a pair of isomorphic MDOPs is defined as:

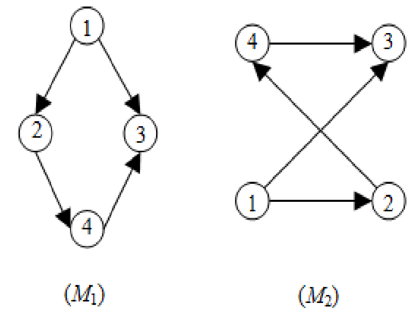


Figure 4. An Example of a Pair of Isomorphic MDOPs

Definition 6: Given two MDOPs G and H, if $V(G)=V(H)$, $E(G)=E(H)$, $L(V(G))=L(V(H))$, then G and H are a pair of **isomorphic MDOP**s.

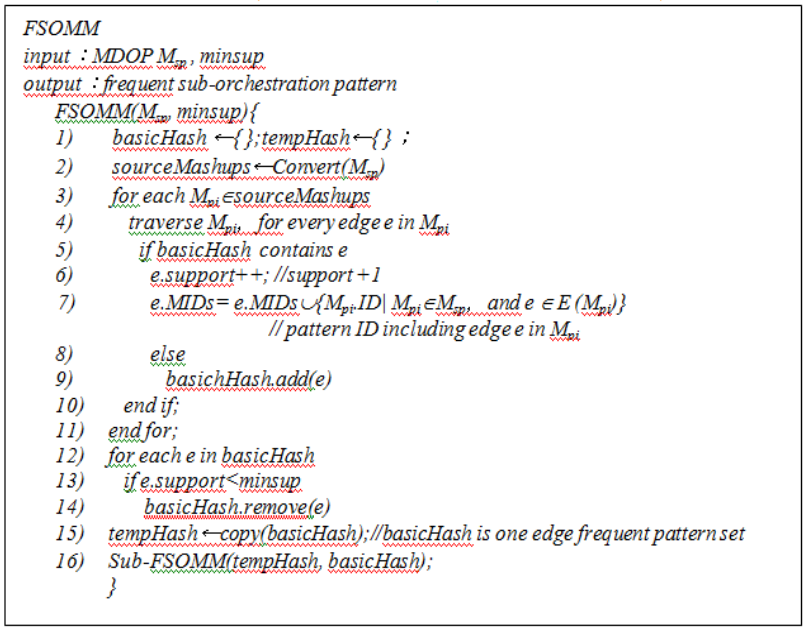
According to Definition 6, two isomorphic MDOPs satisfy the following two conditions: (1) they should call the same set of Open APIs; (2) Their orchestration modes of the Open APIs are also equal. After being mapped onto MDOPs, two MDOPs are isomorphic only when the IDs of the vertices, number of edges and edge directions are all identic correspondingly.

Definition 7: Let MDOP $M=\{V(M),E(M)\}$. If MDOP $M'=\{V(M'),E(M')\}$ satisfies $V'\subseteq V$, $E'\subseteq E$ and $E(M')=\{(API_i, API_j)|\ API_i, API_j \in V(M')\}$, then M' is **a sub-pattern** of M.

Based on Definition 5~7, the problem of mining frequent mashup patterns is formally described as follows: Suppose M_s is a collection containing n mashup applications, its corresponding MDOP set $M_{sp}=\{M_{pi}|M_{pi}$ is the MDOP of mashup application M_i, and i=0, 1,2,…,$n\}$, the minimum support is *minsup* and $f(m_p, M_{pi})$ represents if there is an isomorphism of m_p in M_{pi}. The value of $f(m_p, M_{pi})$ is determined as follows: if m_p is isomorphic with a sub-pattern m_{pi} of M_{pi}'s, then $f(m_p, M_{pi})$=1; otherwise, $f(m_p, M_{pi})$=0. Let $\sigma(m_p, M_{sp}) =\sum_{M_{pi}\in M_{sp}} f(m_p, M_{pi})$, so that $\sigma(m_p, M_{sp})$ indicates the frequency of m_p appeared in M_s, i.e the support of m_p in M_{sp}。 Thus, mining frequent mashup patterns is to develop an applicable algorithm to discover all the sub-patterns m_p in M_{sp} which satisfy $\sigma(m_p, M_{sp})\geq$ *minsup*.

2.2 Mining Algorithm---FSOMM

Before mining the frequent mashup patterns, we need to focus on two fundamental issues: how to get 1-edge frequent sub-patterns, and how to extend the frequent sub-patterns in order to find the frequent patterns with more edges. In [7], Sub-graph mining algorithms are divided into Apriori-based algorithms and pattern-growth-based algorithms. These two kinds of algorithms differ from each other in their pattern expanding strategies. For Apriori-based algorithms, they generate frequent *k*-edge sub-pattern candidates by joining two frequent (*k*-1)-edge sub-patterns that have one and only one different edge. For pattern-growth-based algorithms, the candidates are created by recursively adding 1-edge to frequent patterns starting from frequent 1-edge sub-graphs. Therefore, the former algorithms need to obtain the frequent (*k*-1)-edge patterns before computing the frequent *k*-edge candidates. The latter algorithms are in much the same way as the depth-first traversal of a graph, and its' frequent *k*-edge candidates do not depend on discovered frequent (*k*-1) –edge patterns.



```
FSOMM
input ：MDOP $M_{sp}$ , minsup
output ：frequent sub-orchestration pattern
    FSOMM($M_{sw}$, minsup){
    1)      basicHash ←{ };tempHash←{ } ；
    2)      sourceMashups←Convert($M_{sp}$)
    3)      for each $M_{pi}$ ∈sourceMashups
    4)        traverse $M_{pi}$,  for every edge e in $M_{pi}$
    5)         if basicHash contains e
    6)           e.support++; //support+1
    7)           e.MIDs= e.MIDs∪{$M_{pi}$.ID| $M_{pi}$∈$M_{sp}$,  and e ∈E ($M_{pi}$)}
                               //pattern ID including edge e in $M_{pi}$
    8)           else
    9)             basichHash.add(e)
    10)      end if;
    11)    end for;
    12)    for each e in basicHash
    13)      if e.support<minsup
    14)        basicHash.remove(e)
    15)    tempHash←copy(basicHash);//basicHash is one edge frequent pattern set
    16)    Sub-FSOMM(tempHash, basicHash);
           }
```

Figure 5. Algorithm FSOMM

Considering the features of MDOM described previously, we propose the algorithm FSOMM to mine the frequent mashup patterns hidden in mashup applications. The expanding strategy employed in FSOMM is as follows: the *k*-edge frequent patterns are obtained by adding one frequent edge pattern onto the frequent (*k*-1)-edge patterns. This strategy can be regarded as a combination of those used by the two kinds of algorithms for mining frequent sub-graphs, however, there are also some differences between them. The main distinction between Apriori-based algorithms and FSOMM is that the candidate *k*-edge patterns produced with FSOMM are not generated through joining two frequent (*k*-1)-edge patterns. Unlike pattern-growth-based algorithms that calculate the frequent sub-patterns by extending 1-edge sub-patterns in depth, FSCOMM expands all the frequent *k* -patterns($k \geq 2$) after doing that to all the frequent 1-edge patterns.

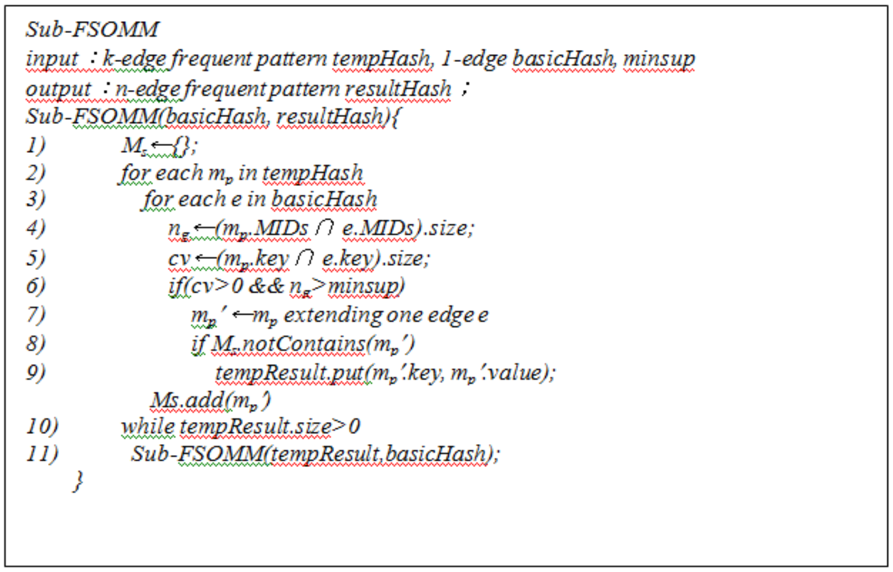
During the computation, after increasing one edge to a frequent (*k*-1)-edge sub-patterns m_p and generating a *k*-edge sub-patterns m'_p, FSOMM immediately examines if *m'ₚ* is a frequent *k*-edge sub-pattern. If m_p' is a frequent sub-pattern, m_p will be extended. Otherwise, m_p will be kept unchanged. This strategy is efficient because it saves time by reducing the detecting times of isomorphic sub-patterns. Figure 5 gives the steps of FSOMM algorithm. The description shows the computing process of FSOMM can be divided into two parts: the first part is to obtain the frequent 1-edge patterns by statistics, and the second part computes the frequent patterns more than one edge by recursively extending frequent 1-edge patterns. The detail of FSOMM is described as follows:

Step 1 (line 2) converts the adjacency matrix representing a mashup pattern M_{pi}'s into a hash table. The key of the hash table is a list composed by the IDs of the Open APIs at the both ends of a directed edge in MDOP, and its value includes three parts: 1) times of current edge appeared in M_{sp}, which is expressed as *support*; 2) the IDs of all MDOPs that contain edge *e*; 3) the direction of *e*, which is indicated as *dir*. For example, if an edge *e* whose vertices are Open APIs 2 and 5 is directed from 2 to 5, then e.*dir*={2_5}。 All the hash tables are stored in a collection named as *sourceMashups*.

Step 2 (line 3-11) calculates the support of edge *e* in M_{sp} and adds the IDs of MDOPs which contain edge *e* into the result set.

Step 3 (line 12-14) removes the edges whose support is less than *minsup* from *basicHash*. Then, the left patterns in the *basicHash* are the frequent mashup patterns.

Step 4（line 15-16）assigns *basicHash* to *tempHash*. Even the two hash tables have same the values, their purposes are different and will be reflected in algorithm Sub-FSOMM (see Figure 6). *tempHash* is composed of the candidates of frequent *k*-edge patterns which will be extended recursively, while *basicHash* is the set of frequent 1-edge patterns which will be added into the frequent (*k*-1)-edge patterns in order to generate the frequent *k*-edge patterns. The value of *basicHash* maintains constantly during the whole process of recursively revoking algorithm Sub-FSOMM. Both *basicHash* and *tempHash* are parameters for revoking Sub-FSOMM, which will generate all the frequent sub-patterns. The description of Sub-FSOMM is presented in Figure6.



```
Sub-FSOMM
input ：k-edge frequent pattern tempHash, 1-edge basicHash, minsup
output ：n-edge frequent pattern resultHash ；
Sub-FSOMM(basicHash, resultHash){
1)        $M_s$←{};
2)        for each $m_p$ in tempHash
3)          for each e in basicHash
4)            $n_g$←($m_p$.MIDs ∩ e.MIDs).size;
5)            cv←($m_p$.key ∩ e.key).size;
6)            if(cv>0 && $n_g$>minsup)
7)              $m_p$' ←$m_p$ extending one edge e
8)              if $M_s$.notContains($m_p$')
9)                tempResult.put($m_p$'.key, $m_p$'.value);
          Ms.add($m_p$')
10)       while tempResult.size>0
11)        Sub-FSOMM(tempResult,basicHash);
     }
```

Figure 6. Algorithm Sub-FSOMM

Sub-FSOMM adds the frequent 1-edge patterns to one frequent *k*-edge pattern each time when it is called, and the frequent edges are picked from *basicHash*. The Sub-FSOMM will be stopped until all frequent patterns have been found. In Line 4, n_g is the number of common vertices between frequent pattern m_p and frequent 1-edge pattern *e*. In Line 5, *cv* represents the number of mashup applications containing m_p and *e* simultaneously. Line 6 decides if there are some common vertices between m_p and *e*. If they have common vertices and the number of mashup applications containing m_p and *e* is not less than *minsup*, *e* will be add into m_p to generate a candidate frequent pattern m_p'. Checking whether there are frequent *k-edge* patterns isomorphic with m_p' is accomplished by Line 8. If there are any isomorphic patterns with m_p', terminate this iteration; otherwise, m_p' will be added into *tempResult* and then continue adding the next edge. After all frequent patterns are traversed completely, if the number of frequent sub-patterns is not zero, then go on calling Sub-FSOMM until all the frequent patterns are obtained.

Notice that M_s is a hash table whose key is the list expressing the direction of sub-pattern m_p. For example, m_p contains three edges <2, 4>、<4, 5>、<1, 6>, then in M_s, the key of m_p is a list representing all the three edges, that is, {1_6, 2_4, 4_5}. Therefore, every time before adding a new frequent pattern into the result set, it only needs to check if the result set contains a key which is same to that of the new one. The experimental results show that storing frequent patterns with hash table could reduce the checking time so that improves the efficiency of mining frequent mashup patterns.

3. Experimental Analysis

3.1 Experimental settings

The experiments are conducted on two datasets: a real dataset and a synthetic dataset. The real dataset consists of information of the mashup applications downloaded from ProgrammableWeb.com, and each mashup application calls at least 2 Open APIs. There are 3318 mashup applications that form the same number of different MDOPs and 1224 Open APIs in the dataset. Some preprocessing work, such as assigning an ID to each of the Open APIs, is done on the real dataset before conducting the experiments. The synthetic dataset is generated by our self-developed algorithm that produces simulated mashup applications according to the features of MDOM discussed in Section 3.2. The dataset contains 10,000 simulated mashup applications and each simulated mashup application is formed with 8~10 Open APIs. As we know, in the real mashup applications, the number of invoked Open APIs is often between 2 and 5. Note that the number of Open API in our synthetic mashup applications is 8~10 which is different from that of the real mashup applications. That is because that our aim for using then synthetic data is to test if the performance of our proposed algorithm is also good when it is conducted on complex data. All experiments are carried out on a PC with Windows 7 Ultimate 64-bit operating system, 4G memory and 1TB hard disk.

Before mining the frequent mashup patterns from the real dataset, we verify the efficiency and the effectiveness of FSOMM with synthetic dataset. Two groups of experiments are designed to achieve this goal. The first group is to investigate the performance of FSOMM by comparing it with FSG and gSpan on executing time and integrity of mining results with synthetic dataset. This group of experiments is trying to test if FSOMM algorithm is more suitable for mining frequent MDOM compared with some existing algorithms. The second group is to mine the frequent sub-patterns hidden in real dataset with FSOMM. The frequent sub-patterns are useful to create optimal mashup applications.

3.2 Experimental Analysis

Experiment Group 1: Investigating the Performance of FSOMM

Experiment 1: examining the efficiency of FSOMM. Figure7 shows the corresponding time cost under different *minsup* when mining frequent patterns with FSOMM, gSpan and FSG. The dataset is synthetic dataset which was kept constantly during experimenting. The horizontal axis represents the threshold of support *minsup*, and the vertical axis is the time taken by the algorithms for mining the frequent sub-patterns. According to the experimental results, it is observed that FSOMM spends the least time mining among the three algorithms when the support threshold *minsup* is small (less than 6%). As *minsup* increases, the running time of the three algorithms become closer, especially when *minsup* is greater than 10%, the running time trends to be equal. That is because as the increase of *minsup*, the number of frequent patterns decreases obviously. That means the scale is also reduced for further computing. Simutaneously, it demonstrates the performances of the three algorithms are guaranteed when the computational scale is small.

Experiment 2: validating the integrity of mining results with FSOMM. Figure8 illustrates the number of frequent patterns mined with three algorithms varies with the minimal threshold of support *minsup*. In Kuramochi and Karypis (2001) and Yan and Han (2002), the authors pointed out that FSG and gSpan could mine all the frequent patterns, and Figure 8 indicates the number of frequent patterns that were found by FSOMM is nearly the same as that were found by FSG and gSpan. Thus, FSOMM algorithm is able to discover frequent mashup patterns completely.

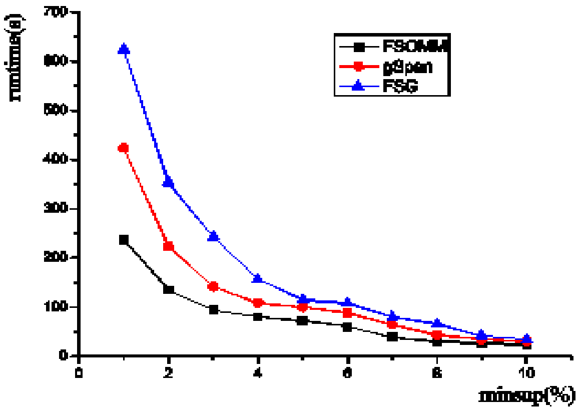


Figure 7. Executing Time of Different Algorithms Varies with Support Threshold Minsup

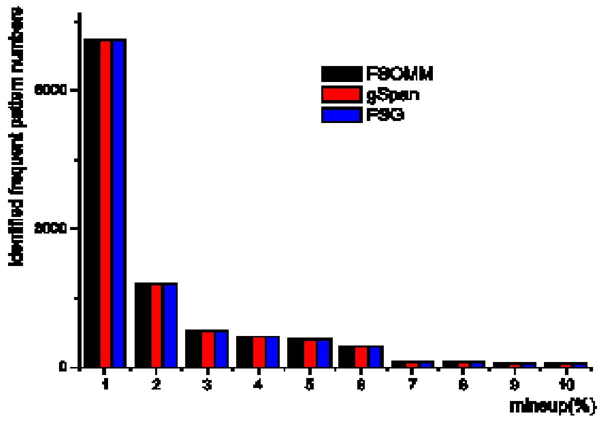


Figure 8. Number of Identified Frequent Patterns Varies with Different Minsup

Experiment Group 2: Mining Frequent Orchestration Patterns on Real Datasets.

The experiments performed on synthetic dataset demonstrate that FSOMM can discover the frequent orchestration patterns completely and efficiently. Now, we try to investigate the performance of mining frequent patterns implied in real dataset with FSOMM from three aspects with Experiment 3~5.

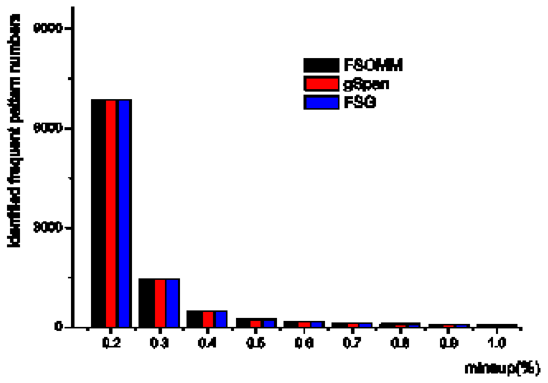


Figure 9. Identified Frequent Patterns with Different Minsup

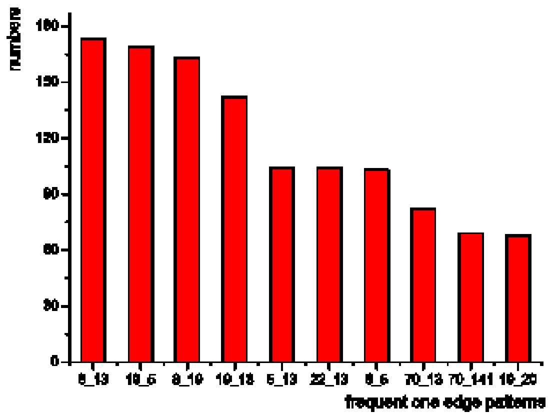


Figure 10. Top-10 Frequent 1-edge Patterns

Experiment 3: investigating the number of frequent patterns changing with different minimal threshold of support *minsup*. According to the statistics, we know there are 3,318 mashup applications and 1,224 Open APIs contained in the real dataset. Each mashup application has 2~5 vertices on average. Thus, the emerging frequency for each Open API ranges from 5 to 14, and the corresponding average support is between 0.2% and 0.4%. Generally, only the patterns whose support is greater than the average one are useful for optimizing mashup applictions. Therefore, during the experiments, *minsup* is set to 0.2%, 0.3% …, 0.5%… 1%. Figure 9 depicts the number of discovered frequent patterns varies with the *minsup*. It displays when the minimal threshold of support ranges from 0.2% to 0.4%, the number of discovered frequent patterns decreases sharply. While when *minsup* is greater than 0.4%, the decrease becomes slower. This implies the dispersion and diversity of Open APIs revoked in real mashups.

Experiment 4: mining top-10 1-edge frequent patterns. The frequency of a mashup pattern reflects the degree of its reliability and popularity to the users. Recommending the top-*k* frequent patterns to users could improve the efficiency of building mashups. Figure 10 gives the experimental results. The figure proves some frequent invoked Open APIs can be integrated with many other Open APIs for creating new mashup applications. For example, Google Maps API can not only be combined with Youtube API to form a video mahsup application, but also can be orchestrated with Twitter or Flickr to build other special mashup applications. That is why there are so many 1-edge frequent patterns with the same vertex, like “8” appeared in both “8_13” and “8_19”. It deserves to be noted that for a given Open API, it may have different relationships with other Open APIs in different mashup applications. For instance, for the mashup labeled as “19_5” and “5_13”, even though the Open API with ID of “5” is involved in the two patterns, “19_5” and “5_13”, it has a different mashup pattern with “19”and “13” because the invoking sequences are different.

Experiment 5: investigating the distribution of frequent *k*-edge patterns under different values of *minsup*. The value of *minsup* impacts greatly on the mining result. If it is too small, the frequent patterns will grow exponentially and make it hard to filter the patterns. On the contrary, if the value is too large, fewer frequent patterns will be found and it may lead the valuable ones not be able to be recommended. Therefore, to set an appropriate value for *minsup* is very important for keeping the frequent patterns in a proper domain.

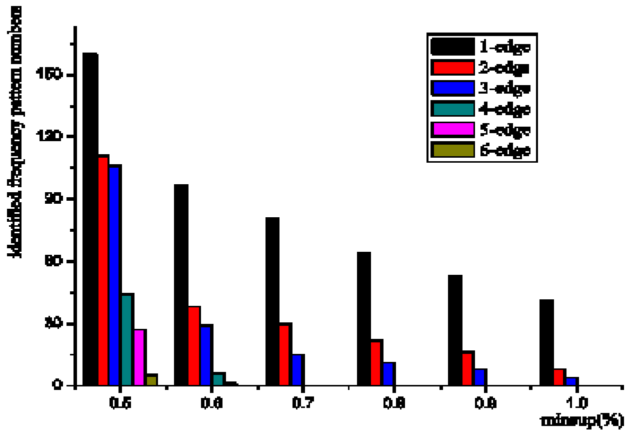


Figure 11. Distribution of Frequent Patterns Under Different Minsup

Figure 11 shows the distribution of frequent patterns under different value of *minsup*. The figure indicates that the number of frequent k-edge patterns decreases as *minsup* increases. While when *minsup* keeps constantly, the number of frequent patterns decreases as the number of edges becomes greater. That is caused by the sparsity of MDOP because the vertices in MDOPs are very few (normally, there are only 2~5 Open APIs contained in a mashup application). In practical, the value of *minsup* can be determined according to the requirements of application.

4. Conclusion

Mining frequent mashup patterns is significant for building optimal mashup applications efficiently. The MDOM presented in this paper can visually describe the orchestration mode among Open APIs invoked in mashup applications. On the basis of MDOM features, a directed graph which can express MDOM in formal description – MDOP is also proposed. An algorithm, called FSOMM, is developed specially for discover frequent mashup patterns by combining the strategies employed in Apriori-based and pattern-growth-based algorithms for mining frequent sub-graphs, and the features of MDOM as well. Experimental results show FSOMM is more suitable to mine the frequent MDOP patterns because most of the real mashups are very small (3~5 Open APIs). In our future work, we will develop an effective structure to index the discovered frequent mashup patterns so that they can help users to create optimal mashup applications in an efficient way.

Acknowledgement

This work is partially supported by the National Natural Science Foundation of China (NSFC) under Grant No. 61202376 and No.61003031, and Shanghai Pudong New Area Science and Technology Development Foundation Innovative Funding for Scientific and Technological Cooperation Project under Grant No.PKJ2013-Z14.

References

[1] J. Yu, B. Benatallah, F. Casati and F. Daniel, “Understanding mashup development”, Internet Computing, vol. 12, no. 5, **(2008),** pp. 44-52.
[2] A. Thor, D. Aumueller and E. Rahm, “Data Integration Support for Mashups”, Proceedings of the 6th International Workshop on Information Integration on the Web, Vancouver, Canada, **(2007),** pp.104-109.
[3] A. Koschmider, V. Torres and V. Pelechano, “Elucidating the Mashup Hype: Definition, Challenges, Methodical Guide and Tools for Mashups”, Proceedings of WWW2009 Workshop on Mashups, Enterprise Mashups and Lightweight Composition on the Web, Madrid, Spain, **(2009).**
[4] O. Greenshpan, T. Milo and N. Polyzotis, “Autocompletion for Mashups”, Proceedings of the VLDB Endowment, vol. 2, no. 1, **(2009),** pp. 538-549.
[5] A. Bouguettaya, S. Nepal, W. Sherchan, X. Zhou, J. Wu, S. Chen, D. Liu, L. Li, H. Wang and X. Liu, “ End-to-end service support for mashups”, IEEE Transactions on Services Computing, vol. 3, no. 3, **(2010),** pp. 250-263.
[6] J. Cao and C. Xing, “Data Source Recommendation for building mashup applications”. Proceedings of the 7th Web Information Systems and Applications Conference, Hohhot, China, **(2010),** pp. 220-224.
[7] J. Wang, H. Chen and Y. Zhang, “Mining user behavior pattern in mashup community”, Proceedings of the 10th IEEE International Conference on Information Reuse and Integration, Las Vegas, NV, USA, **(2009),** pp. 126-131.
[8] G. Wang, J. Liu, B. Cao and M. Tang, “ Mashup service classification and recommendation based on similarity computing”, Proceedings of the Conference on the 2nd Cloud and Green Computing, Xiangtan, Hunan, China, **(2012),** pp.621-628.
[9] G. D. Lorenzo, H. Hacid and H.-Y. Paik, “Data integration in mashups”, ACM Sigmod Record, vol. 38, no. 1, **(2009),** pp. 59-66.

[10] J. Liu, J. Wei, D. Ye and T. Huang, "Performance Optimization of Mashup through Data Flow Transformation", Journal of Chinese Computer Systems, vol. 32, no. 9, **(2011),** pp. 1716-1721.
[11] F. Daniel, J. Yu, B. Benatallah, F. Casati, M. Matera and R. Saint-Paul, "Understanding UI integration: A survey of problems, technologies, and opportunities", Internet Computing, vol. 11, no. 3, **(2007),** pp. 59-66.
[12] J.Yu, B. Benatallah, R. Saint-Paul, F. Casati, F. Daniel and M. Matera, "A framework for rapid integration of presentation components", Proceedings of the 16th International World Wide Web Conference, Banff, Alberta, Canada, **(2007),** pp. 923-932.
[13] D. Bianchini, V. D. Antonellis and M. Melchiori, "A recommendation system for semantic mashup design", Proceedings of the 21th International Conference on Database and Expert Systems Applications, Bilbao, Spain, **(2010),** pp.159-163.
[14] S. Abiteboul, O. Greenshpan and T. Milo, "Modeling the mashup space", Proceedings of ACM 17th Conference on Web Information and Data Management, Napa Valley, California, USA, **(2008),** pp. 87-94.
[15] H. Elmeleegy, A. Ivan, R. Akkiraju and R. Goodwin, "Mashup advisor: a recommendation tool for mashup development", Proceedings of the 6th International Conference on Web Services, Beijing, China, 2008:337-344.
[16] A. Marconi, M. Pistore and P. Traverso, "Automated Composition of Web Services: the ASTRO Approach", IEEE Data Engineering Bulletin, vol. 31, no. 3, **(2008),** pp. 23-26.
[17] S. R. Chowdhury, "Assisting end-user development in browser-based mashup tools", Proceedings of the 34th International Conference on Software Engineering, Zurich, Switzerland, **(2012),** pp. 1625-1627.
[18] W. Pan, B. Li, B. Shao and P. He, "Service Classification and Recommendation Based on Software Networks", Chinese Journal of computers, vol. 34, no. 12, **(2011),** pp. 2355-2369.
[19] T. Akutsu and H. Nagamochi, "Comparison and Enumeration of Chemical Graphs". Computational and Structural Biotechnology Journal, no. 5, **(2013)**.
[20] T. Milenković and N. Pržulj, "Topological Characteristics of Molecular Networks", Functional Coherence of Molecular Networks in Bioinformatics, Springer New York, **(2012).**
[21] M. Khan, V. A. Kumar, M. V. Marathe and Z. Zhao, "Algorithms for Finding Motifs in Large Labeled Networks", Dynamics On and Of Complex Networks, vol.2, Springer New York, **(2013)**, pp. 243-263.
[22] G. Wang, J. Yin and W. Zhan, "A novel graph classification approach based on embedding sets", Journal of Computer Research and Development, vol. 49, no. 11, **(2012),** pp. 2311-2319
[23] M. R. Keyvanpour and F. Azizani, "Classification and analysis of frequent subgraphs mining algorithms", Journal of Software, vol. 7, no. 1, **(2012),** pp. 220-227.
[24] A. Inokuchi, T. Washio and H. Motoda, "An apriori-based algorithm for mining frequent substructures from graph data", Principles of Data Mining and Knowledge Discovery. Springer Berlin Heidelberg, Lyon, France, **(2000).**
[25] M. Kuramochi and G. Karypis, "Frequent subgraph discovery", Proceedings of the 1st International Conference on Data Mining, San Jose, California, USA, **(2001),** pp. 313-320.
[26] X. Yan and J. Han, "gSpan: Graph-based substructure pattern mining", Proceedings of the 2nd International Conference on Data Mining, Maebashi City, Japan, **(2002),** pp.721-724

Authors

Dunlu Peng, he is a professor of University of Shanghai for Science and Technology, Shanghai, China. He received his Ph.D degree from Fudan University, Shanghai, China in June 2006. He served as a PC member of SCC, EDOC, APSCC, WHICEB, CLOUD, etc. His research interests include Cloud Computing, Web applications, service-oriented computing, XML data management and Web mining.

Lei Xie, she is a master degree candidate of University of Shanghai for Science and Technology, Shanghai, China. She received her bachelor's degree from University of Shanghai for Science and Technology, Shanghai, China in June 2011. Her research interests include Web Mashup application, Cloud computing and Web mining.

Kai Duan, he is a master degree candidate of University of Shanghai for Science and Technology, Shanghai, China. He received his bachelor's degree from University of Shanghai for Science and Technology, Shanghai, China in June 2011. His research interests include Cloud Computing, Service-oriented computing and Web mining.

Feitian Li, he is now a software architect of Shanghai BroadText Iswind Software Co. Ltd, Shanghai, China. He received his bachelor degree from Heifei University of Technology, Hefei, China in June 2001. His research interests include Web applications, SOA and software architecture.