Using Sequential Pattern Mining and Interactive Recommendation to Assist Pipe-like Mashup Development

Xinyi Liu, Hailong Sun, Hanxiong Wu, Richong Zhang, Xudong Liu
School of Computer Science and Engineering
Beihang University, Beijing, China 100191
Email: {liuxinyi,sunhl}@act.buaa.edu.cn, metalcrashw@gmail.com,{liuxd,zhangrc}@act.buaa.edu.cn

***Abstract*—Mashups represent a typical type of service oriented applications targeting end-user development. However, due to lack of development expertise, end-users usually find it hard to build a mashup. Therefore, it is of paramount importance to provide effective assistance to achieve efficient mashup development. In this work, we aim at leveraging the expertise that can be mined from voluminous mashups on Internet to recommend appropriate mashup modules and their composition patterns to facilitate pipe-like mashup development. First, we crawl all the mashups available in Yahoo!Pipes and extract the meta-data of each mashup from original JSON data. Second, we use GSP (Generalized Sequential Pattern) algorithm to mine the frequent composition pattern of mashup modules, and design an interactive recommendation algorithm to assist mashup development. Third, we implement a system prototype based on the proposed method and evaluate its effectiveness with 848 Yahoo! mashups through cross-validation.**

***Keywords*-end-user programming; mashup; sequential pattern mining; interactive recommendation;**

I. Introduction

The web mashup is an application developed with less programming ability in drag-and-drop way with integrated diverse Internet resources. Nowadays, rich numbers of web mashups have been created by Internet users. The two main mashup resources collectors are Yahoo!Pipes [1] which has more than 10000 web mashups and ProgrammableWeb [2] which has 7250 mashups, and the number of mashups still grows steadily. In recent years, Mashup recommendation plays an important role of mashup research.

The huge amount of resources cause difficulties in selecting suitable data sources for data integration. For instance, a mashup user, Tom, prepares to create a mashup application for integrating news from some news sites. He expects that the content of news information should involve current affairs, social news, sports news and entertainment news from various news websites which have their individual characteristics. This requirement makes it hard to select the suitable data sources in the numerous candidates. While constructing the news mashup, Tom may not know how to take his first step in the face of an empty edit area of the mashup application development tool and its toolbar has a variety of options to drag. This "blank-obstacle" usually appears when green-hand users use a development tool such as Eclipse, Photoshop. Although mashup development allows programming-less skill, the lack of programming ability of general Internet users, whom the most of mashup development tools serve to, leads to a rough and error-prone work. The tool offers recommendation of data sources and operators at each step to give the user clue to build the rest of their own mashups.

In order to achieve a high quality and effective recommendation, it can be concerned to make use of existing mashup resources to recommend. There is a lot of engaged mashup applications, and these resources are considered to be mature and reliable. This study takes advantage of mashup applications from Yahoo! pipes, mines and analyzes the data, then extracts the data sources and reusable sequences by the optimized classic sequence mining algorithm, GSP (Generalized Sequential Pattern) [3]. An implementation is realized for evaluating the effect and illustrating the feasibility of this approach.

The state-of-the-art of mashup recommendation can be considered in two distinct approaches. Both of them take data sources' similarity into consideration. The difference between them is the form of recommendation result. The first is trying to recommend a whole application with domain-dependency, and the second discusses the internal structure to recommend a frequent component or partial composition patterns. Inspired by the second approach, a different method which not only consider frequency of internal components but also concern the logic order of components is proposed in this study. The contributions of this work are:

- We crawl all the mashups available in Yahoo!Pipes and extract the meta-data of each mashup from original JSON data.
- We use GSP (Generalized Sequential Pattern) algorithm to mine the frequent composition patterns of mashup modules, and design an interactive recommendation algorithm to assist mashup development.
- We implement a prototype system based on the proposed method and evaluate its effectiveness with 848 Yahoo! mashups through cross-validation.

The rest of the paper is organized as follows. Section II discusses the related work. Section III explains the motivation and the problem. Section IV presents the recommendation approach. Section V introduces an implementation and evaluated the effectiveness by the result of an simulation experiment. Finally, conclusions and future works are clarified in Section VI.

II. Related Works

Mashup emerged in 2007, at first, the study of mashup paid much attention on implementing tools to construct mashup without recommendation such as Marmite [4], Damia [5]. Marmite [4] allows users to create mashups that repurpose and combine existing web resources. To address the problem that creating mashups requires a great deal of programming expertise, Marmite was designed to support a data flow architecture, where data is processed by a series of operators in a manner similar to Unix pipes. Damia [5] is a integration platform which access and combine data from a variety of sources in web style. It allows business users to rapidly and simply create mashups associate with desktop, web, and traditional IT sources into feeds that can be consumed by AJAX, and other types of web applications. With trend of recommendation technology, some researchers try to adapt recommendation in building mashups. MashMaker [6] can automatically suggest widgets that have been applied to similar data inside the users social network. Both [7], [8] are in a same basic idea to present a tag-base approach with a semantic analysis. Bouillet [7] uses a tag model where domain-specific tags, organized into facets, are used to describe end-user information processing goals, component functional processing and data semantics, and application requirements and structural constraints. The composition approach makes use of the tags to decide whether certain components can be composed together into a flow. Goarany [8] propose a tag-based approach for predicting mashup patterns which applies association rule mining techniques to discover relationships between APIs and mashups based on communities' annotated tags. Cao [9] presents a recommendation tool assisting users to find proper data source during building mashups. These research all focus on building semantic model. Another related research work directly solve the problem by looking for connections between the components. When the user provides a set of components or data sources, the tool will recommend them suitable connector (so-called glue pattern) to connect the components which satisfy user-specified demand. MatchUp [10] is a system based on a novel autocompletion mechanism which exploits similarities in mashup components and glue patterns to recommend completions. It tries to not only find the most suitable domain-dependent mashlet components, but more importantly, glue them together in an effective way. So MatchUp exploits similarities between the ways users glue together mashup components and recommends possible completions by given a users partial mashup specification. MashupAdvisor [11] is a system for assisting mashup builder by suggesting relevant output to partially constructed mashups. MashupAdvisor relies on a repository of mashups developed by a community of users to generate recommendations. The recommendations are computed based on a probabilistic model that ranks the mashup outputs based on their popularity (i.e., usage in the mashup repository). The method in [12] allows users to describe navigation flows of interest and presents an effective query evaluation algorithm on how to navigate. The solution is based on a simple, generic model for MashAPPs and navigation flows within them, and an intuitive query language that allows users to define navigation flows of interest. Chowdhury [13] focuses on mashups and modeling tools to provide a step-by-step recommendation for developers in their composition task. He [14] also presents an assisted development approach that integrates automatic composition and interactive pattern recommendation techniques. Chowdhury [15] demonstrates how such a hybrid assistance solution can be designed in practice. The system is developed on top of an existing open-source, widget-based mashup platform Apache Rave, which combines the simplicity of a dialog-based automatic composer with the step-by-step assistance by an interactive pattern recommender. Stolee [16] focuses on refactoring smelly pipe-like mashups to improve quality. The authors investigate how to bring the benefits of software engineering to end users programming mashups. And they do this by focusing on automated smell identification and refactoring.

We take advantage of sources from Yahoo!Pipes [1] since that some additional available mashup directories do not offer enough information. Programmableweb.com [2] provides less information of combinations between mashup interior components and WebAPI.org [16] contains too small collection of mashup applications.

III. Problem Description

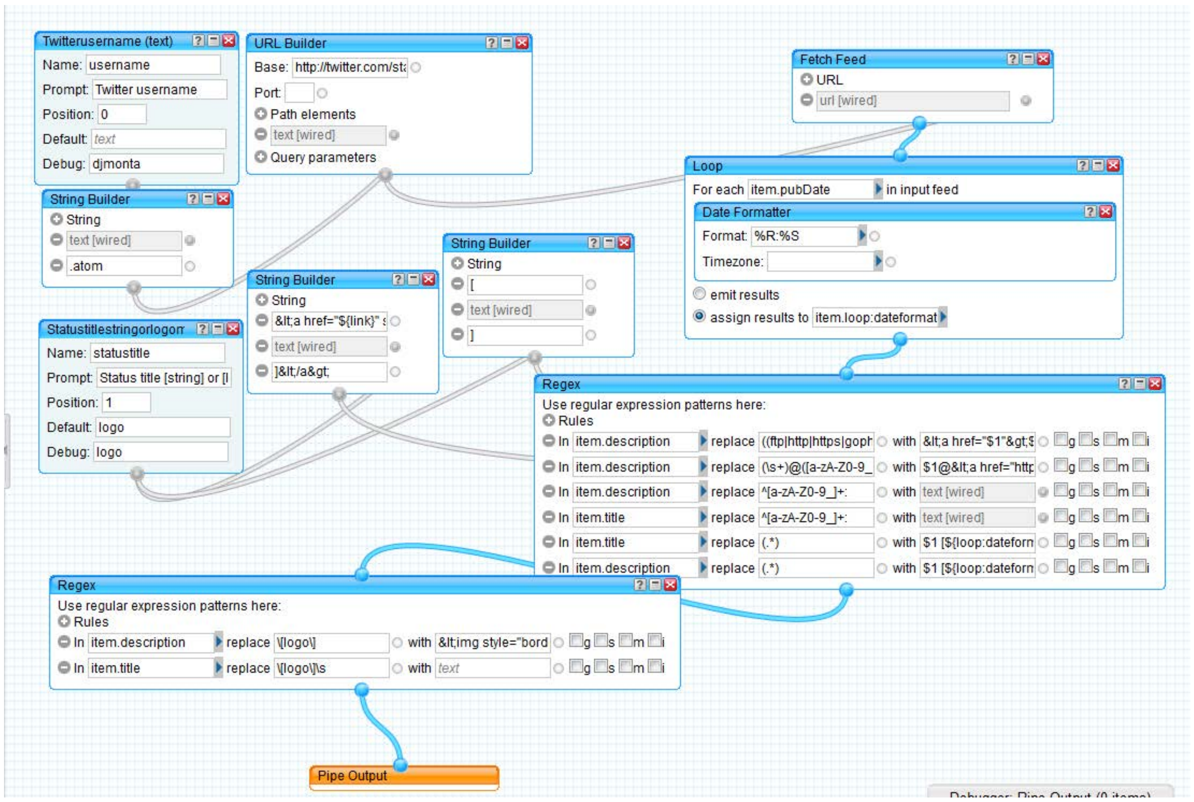
A. The Model

This section details a model for a mashup application.

A *pipe* refers to a mashup application, also defines a pipe graph like Figure 1 illustrates, which is a composition of multiple modules and wires with specific rules.

A *sequence* refers to a sequential string that contains modules on the *main wires*(explain in this section later). There may be one or more sequences in a pipe. Figure 1 shows a typical simple pipe including one sequence. Furthermore, Figure 2 illustrates two heterogeneous pipes with loop and branch which can be taken apart into several sequences, and these pipes are called *multiple pipes*. Both pipes in Figure 2 can be separated into two sequences.

A *module* is a functional component named by action of functions such as *fetch, loop, sort, union*, e.g., the fetch module is used for accessing and extracting data from data sources. Three categories of modules classified by varied functions are *start module* , *operation module*, and *end module*. There are three kinds of interfaces (the balls on


Twitterusername (text)
Name: username
Prompt: Twitter username
Position: 0
Default: text
Debug: djmonta

URL Builder
Base: http://twitter.com/st
Port:
Path elements
text [wired]
Query parameters

String Builder
String
text [wired]
.atom

Statustitlestringorlogom
Name: statustitle
Prompt: Status title [string] or [l
Position: 1
Default: logo
Debug: logo

String Builder
String
<a href="${link}" s
text [wired]
]</a>

String Builder
String
[
text [wired]
]

Fetch Feed
URL
url [wired]

Loop
For each item.pubDate in input feed
Date Formatter
Format: %R:%S
Timezone:
emit results
assign results to item.loop:dateformat

Regex
Use regular expression patterns here:
Rules

In	item.description	replace	((ftp|http|https|goph	with	<a href="$1">$	g s m i
In	item.description	replace	(\s+)@([a-zA-Z0-9_	with	$1@<a href="http	g s m i
In	item.description	replace	^[a-zA-Z0-9_]+:	with	text [wired]	g s m i
In	item.title	replace	^[a-zA-Z0-9_]+:	with	text [wired]	g s m i
In	item.title	replace	(.*)	with	$1 [${loop:dateforn	g s m i
In	item.description	replace	(.*)	with	$1 [${loop:dateforn	g s m i

Regex
Use regular expression patterns here:
Rules

In	item.description	replace	\[logo\]	with	<img style="bord	g s m i
In	item.title	replace	\[logo\]\s	with	text	g s m i

Pipe Output

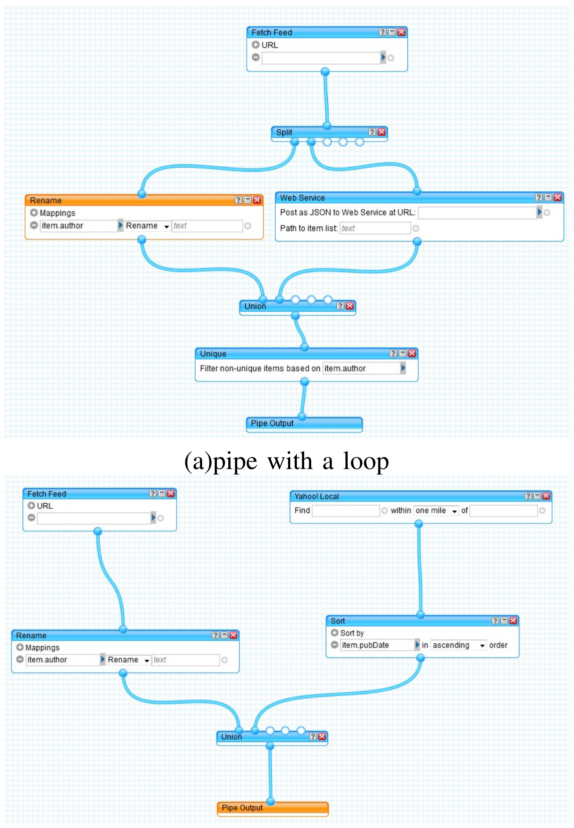
Figure 1: A typical pipe

the end of wires shown in Figure 1) on modules to transmit information: the *imported interface* allows data stream entering the module; the *exported interface* provides an exit for data stream; and the *configure interface* is used for parameters completion in the blank forms of modules. All start modules have an exported interface and their function is mainly to extract data from *data sources* in same way, and the format of data sources can be RSS feeds, XML data or JSON data; the operation module with imported and exported interfaces is like a modifying factory for fetched data stream and process them to satisfy user-specified demand; the end module with an imported interface receives modified integration data stream. In particular, the *output* module is the one and only end module.

A *wire* is a linkage that connects modules and carries data stream with a specific direction. Wires can be divided into two types: the *main wire* and the *supportive wire*. The blue wires in Figure 1 are main wires and the grey ones are supportive wires. The main wire links two different modules. It begins at the exported interface of the former module, and ends at the imported interface of the latter module. Thus the wire transmits data from the beginning to the end. The supportive wire combines an operation module on a sequence with an assistant module (e.g., the StringBuilder module in Figure 1), which completes the missing parameters of the operation module.

B. The Problem Scenario

The following is a specific scenario to explain the problem we would like to solve. A programming-less user, Jerry, he wants to achieve a pipe which can extract popular tweets on twitter.com and the word length of each tweets should be no more than 20. Without the recommendation, Jerry needs to drag three modules from varied modules in tool bar and to connect them with three wires in proper order which is a fairly complex and time-consuming task for him. While with the recommendation, first of all, it's only necessary for Jerry is to choose the twitter.com as a data source input, then the recommendation system returns six sequences as a temporary result:(1) *fetch-regex-output*; (2) *fetch-uniq-sort-truncate-output*; (3) *fetch-loop-regex-output*; (4) *fetch-filter-loop-output*; (5) *yql-loop-truncate-output*; (6) *fetch-loop-output*; because of the need to extract the data source, the *fetch* must be chosen; and for each tweet has to be tailored, the *loop* meets Jerry's requirement. It can be seen that (2)(5) have the subsequence, *fetch - loop*, so Jerry can select the subsequence *fetch-loop* as new input for next iteration, the system then returns five candidates:(1) *output*; (2) *sort-output*; (3) *regex-output*; (4) *uniq-output*; (5) *filter-output*; and (2) has *sort* module which can list tweets by popularity,



(a)pipe with a loop

(b)pipe with a branch

Figure 2: Two cases of multiple pipe

now with more *output* module leads to success. Finally Jerry will get a sequence *fetch-loop-sort-output* namely a pipe satisfied all his demands after once input and one iteration. The comparison notes that the recommendation improves the efficiency of mashup development and requires less contribution and programming skills of users.

IV. Overview of Our Approach

A. Data Collection

The Mashup application is a pipe with input and output, consists of modules and wires. The pipes provided by Yahoo!Pipes were created since 2007 by users of the Yahoo. The general Internet users can run a mashup application on a web page which additionally contains various attributes of this pipe, such as name, author and function description. We crawled 851 available pipes from Yahoo!Pipes as data collection to analyze.

B. Analysis and Mining

Pipe data can be stored in a variety of formats, such as XML, RSS, and JSON. We get the original JSON (Javascript Object Notation) format of the pipes. JSON data format is described in the form of key-value pairs. Table I contains the keys which are also attributes describe a pipe.

Table I: Attributes describing a pipe

name	The name of a pipe named by the author		
pipeID	The unique ID number of a pipe given by yahoo system		
domain	The websites of the data sources used in a pipe		
user	The keywords of the data sources like so-called tags		
module	The types of module used in a pipe		
description	User-defined detailed description for a pipe, including resources, parameters and functions		
working	The key to describe a pipe. The construction of a pipe is based on the information in the working.	layout	Describe the layout of the pipe.
		terminaldata	Describe the output of a module.
		wires	Describe the connection between the two modules
		modules	The detailed description of each module in the pipe
runs	Describe the times the pipe has been run. Each pipe is in the form of a web page and the web page is like a mashup application, when users run the pipe on the mashup page, the running times will be added automatically		
clones	Describe times the pipe has been cloned by other users. Yahoo! Pipes offers the clone function for users to clone and re-edit an old pipe to a new pipe		
favorite	The times of the pipe is marked as a favorite pipe by users.		

The following describes the important components of a pipe, the sub-attributes *wires* and *modules* of the attribute *working*, and details the wire and the module perviously mentioned in the model.

Wires. Each pipe has more than one wires. The format of a wire is like {“id”:“”,“tg”:{“id”:“ ”,“moduleid”:“ ”}, “src”:{“id”:“ ”,“moduleid”:“ ”} }. Its properties involve the id number of the wire, the target module and the source module. And the target key “tgt” and the source key “src” indicate the direction of the wire from “src” to “tgt”.c

Modules. Each pipe has more than one modules, and two modules are attached to a wire. The JSON format of a single module looks like {“id”:“ ”, “conf”:“ ”,“type”:“ ”}. The properties involve the id number that given automatically during editing, the configuration information and the type of the module. The configuration information is much complex and it contains more details of the module. This information can be used in extracting data sources of a pipe.

Modules and *wires* are the emphasis to describe the structure of a pipe, and other attributes are corresponding to the information on the pipe web page.

C. Interactive Recommendation

In this study, we use a classic sequential pattern mining algorithm, GSP, an algorithm used for sequence mining. The algorithm discovers sequences that are frequently reused in patterns. Those sequences can be considered as high-quality segments, so we present them to the end-users.

We firstly present the concepts and definitions used in the algorithms.

- *Seqs* is a set of raw sequences;
- *topK* is the size of recommendation list;
- *minSupp* is the threshold support for GSP;
- *SeqPatterns* carries all sequential patterns mined by GSP;
- *CloSeqPatterns* is an optimized pattern result with closed sequential patterns;
- *recommList* is the recommendation list with the number of sequences ranked descending by the support;
- S_{input} is a sequence as current input for recommendation;
- S_{tgt} is a set of sequences which satisfied user's demand. The multiple pipes have more than one sequences in the set, however the rest normal pipes only have one sequence;
- *source* is data source input.
- *startNodes* is a set of 1-length sequence with start modules such as the *fetch*, the *fetchpage*, the *fetchdata* .

Algorithm 1 recommendMain

```
Input:
    Seqs,topK,source,minSupp;
Output:
    A sequence;
1: CloSeqPatterns = prepareData(Seqs,minSupp);
2: recommList = retrieveSequence(source,topK);
3: recommList.add(startNodes);
4: if $S_{tgt}$ is multiple pipe then
5:
6:     if each sequence in $S_{input}$ all in recommList then
7:         return $S_{tgt}$;
8:     end if
9: else
10:
11:     if $S_{tgt} \in$ recommList then
12:         return $S_{tgt}$;
13:     else
14:
15:         repeat
16:             $S_{input}$ = prefix of $S_{tgt}$;
17:             recommList
                = recommendNextSequence($S_{input}$,topK)
18:             update $S_{input}$ ;
19:         until $S_{input} \subset S_{tgt}$
20:     end if
21: end if
22: return $S_{input}$;
```

Algorithm 1 is the proposed recommendation algorithm. According to the characteristics of pipes, after deciding a data source, there must be a start module as a container for data sources, so the algorithm provides a default recommendation set *startNodes* ranked by the support of these 1-length sequences. Particularly, if a user prepare to integrate several data sources or do multi-processing, which means the target pipe for the user is a multiple pipe, we will take the multiple pipe apart into several sequences first. By setting apart the sequences, we can reduce the complexity of mining sequences in multiple pipes, and which direction that data stream in the loop or the branch moves is determined.

Algorithm 2 prepareData

```
Input:
    Seqs,minSupp;
Output:
    CloSeqPatterns;
1: seqPatterns = GSP(minSupp,Seqs);
2: if $s_i$ is sub-pattern of $s_j$ and the support of $s_i$ is equal
   to $s_j$ //$s_i$,$s_j \in$ seqPatterns then
3:     remove $s_i$ from seqPatterns;
4: end if
5: CloSeqPatterns = seqPatterns;
6: return $S_{input}$;
```

Algorithm 2 is a step to prepare and optimize the sequential patterns set. Depending on the classic sequential pattern mining method, GSP [3], complete sequential patterns could be extracted, and this algorithm 2, which is based on the idea of the closed sequential pattern, continues to optimize the sequential patterns. The reason for optimization is to eliminate the redundancy in the set of sequential patterns. The redundant sequences may occupy the positions in the *recommList* with fixed size.

Algorithm 3 retrieveSequences

```
Input:
    topK,source;
Output:
    recommList;
1: tempList = related sequences of source;
2: sort(tempList);
3: recommList = sub-list of tempList with top k sequences;
4: return recommList;
```

Algorithm 3 depends on input data sources to recommend. When the value of input data sources are not empty, based on the given sources, the algorithm is looking for a set of high-frequency sequential patterns in *Seqs*.

Algorithm 4 leverages the input sequence to recommend, which is an interactive and iterative process, which requires the user to input at every step for the best recommendation. Let each sequence in S_{input} compares to the sequential patterns in the *CloSeqPatterns* , then the algorithm puts

Algorithm 4 recommendNextSequence

```
Input:
    topK,$S_{input}$;
Output:
    recommList;
1: tempList = NULL;
2: for each sequence in CloSeqPatterns do
3:
4:     if items of $S_{input}$ match prefix-sequence of this sequence then
5:         add suffix-sequence of this sequence in tempList;
6:     end if
7: end for
8: sort(tempList);
9: recommList = sub-list of tempList with top k sequences;
10: return recommList;
```

the suffix-sequences of matched sequences into *recommList*. The matched sequences have a prefix-sequence the same as S_{input}. Suppose that S_{input} is *fetch-sort*, so the matched sequences should be start with *fetch-sort*. The *regex-uniq-output*, a suffix of a matched sequence *fetch-sort-regex-uniq-output*, is a suffix-sequence required to add into *recommList*.

V. Experimental Evaluation

A. Experiment Setup

The experiment is conducted under the environment of Windows 7 with a Java 6 implementation. In order to satisfy the demand of dividing data into groups with same size, the experiment just takes 848 pipes as an experimental data set. The average number of modules in sequences is 5.39. The initial size of recommendation list is 10. At every step after triggering iteration mechanism, the size of recommend list expands 5. The minimum support for the GSP is 5.9%. Neo4j is adopted to store pipes, each pipe is stored as one graphic database.

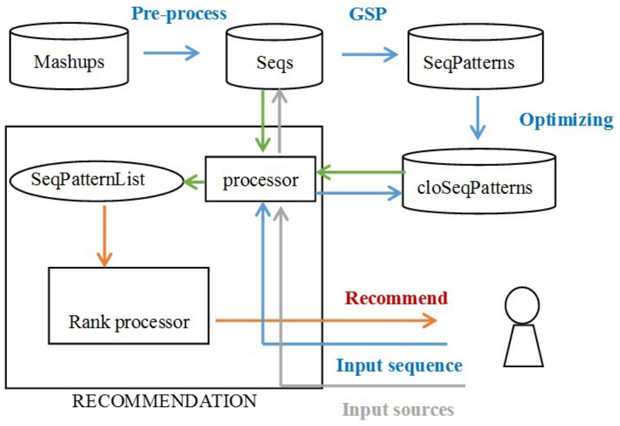


Figure 3: Architecture of system prototype

The architecture of system prototype of this study in Figure 3 is divided into three levels: the data set, the recommendation system and the end-users. The data set is composed of four subsets. The processing process including three steps. Firstly we obtain subset **Seqs** with raw sequences after pre-processing raw pipes in the subset **Mashups**. Then with the assist of the GSP we get a sub-set **SeqPatterns**. Finally optimizing the sequences in **SeqPatterns** leads to a subset **CloSeqPatterns**. The recommendation system has a **SeqPatternList** contains the recommended result and two processors that one is to identify the user's input and to decide what to recommend as the feedback, another one is processing the order of sequences in **SeqPatternList** and tackling with the number of recommended sequences.

The experiment presents a simulation with the implementation which can detect the recommendation effect. We use five metrics to measure the performance of the recommendation approach. The successful recommendation is defined as all separated sequences are recommended correctly, and the accuracy of multiple pipe recommendation can be only 0 or 1. Although the compromise will cause loss of recommendation accuracy, but it does reduce complexity.

Metrics:

1) *p*: the proportion of test pipes that can be recommended successfully, is also called hit ratio;
2) *i*: the average number of iteration times to recommend successfully;
3) *m*: the proportion of test multiple pipes which are recommended successfully;
4) $NDCG$: Normalized Discounted Cumulative Gain. The metric that evaluates recommendation effect of each iteration. The $NDCG$ of the recommendation list in every iteration given a query q is defined as:

$$NDCG = Z_q \sum_i \frac{G(d_i)}{\log_2(i+1)}; \quad (1)$$

where $G(d_i)$ is the gain of the matching degree d_i of the sequence at position *i*. The matching degree d is defined as the ratio of length of the matched sequence to length of rest of target sequence as a relevance score. For instance, the *fetch-loop-regex-output* is the target sequence, and the current input has one module *fetch*. The rest of target sequence is *loop-regex-output* of which length is 3. And the d_1 for a 1-length matched sequence *loop* at first place of a two-sized recommendation list is 1/3, the d_2 of a 2-length matched sequence *loop-regex* at second place is 2/3. Z_q is the normalized term. The higher the $NDCG$, the better result are considered to be.

B. Evaluation Results

We conduct the experiments by separating the dataset into four testing sets, for each set the size is 106, 212, 300, 400 respectively, and the rest of pipes in data set is used as a training set. We emphasize that all metric values in our study are average values.

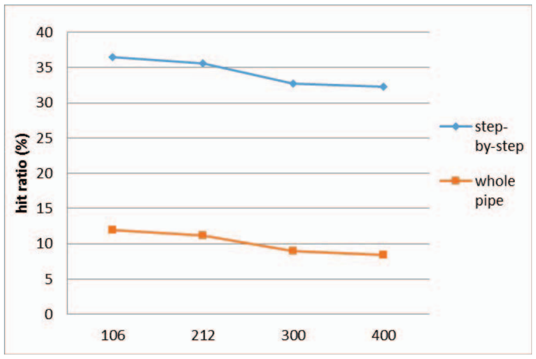
The experimental results are shown in Figure 4. Consider first Figure 4(a) that depicts p with a growing number of the size of test set values varying from the step-by-step recommendation to the whole pipe recommendation. We can observe a moderate linear decrease of p as the size of tests set increases. And p of the step-by-step recommendation is higher than the whole pipe recommendation. We can sum up with both p values in different recommendation ways to acquire successful recommendation is over 40% to 48%. Figure 4(b) shows the $NDCG$ is not always decreasing, when the size of test set is 400, the $NDCG$ increases suddenly. It can be seen as the size of training set reducing, the recommend effect begins to decline. The i in Figure 4(c) are increasing as the size of test set varying from 106 to 300, a sudden change happens again at 400. In the experimental result shown in Figure 4(d) for multiple pipe recommendation only achieve 16% hit, which illustrates the strategy to reduce the complexity of processing to improve the recommendation effect.

We next examines the distribution of $NDCG$ in Figure 5. Let us explain the value on the horizontal. Expect for 0 indicates an explicit 0 value of $NDCG$, others represent a region. For example, 1 represents the $NDCG$ is more than 0 and less than or equal to 0.1; 2 represents the $NDCG$ is more than 0.1 and less than or equal to 0.2 and so on. The diagram shows that almost 35% $NDCG$s are 0, which means nothing to recommend; and 20% $NDCG$s are in the region 5, which need a start module recommendation; and 15% $NDCG$s are greater than or equal to the 1, which mean to make the appropriate recommendations. The last is the test set with 106 pipes to verify the optimization for sequential patterns, the histogram in Figure 6 depicts after optimization the effect is better than before.

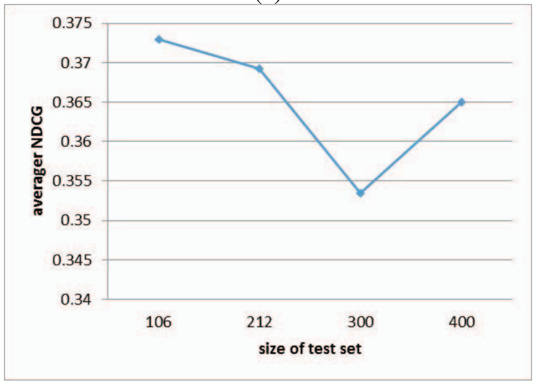
VI. Conclusion

In this paper, we crawl all the mashups available in Yahoo!Pipes and extract the reused resouces of each mashup from original JSON data. Then we propose an interactive recommendation algorithm to assist mashup development by mining the frequent composition patterns of mashup modules with GSP. At last, we implement a system prototype based on the recommendation approach and evaluate its effectiveness with 848 Yahoo! mashups through cross-validation.

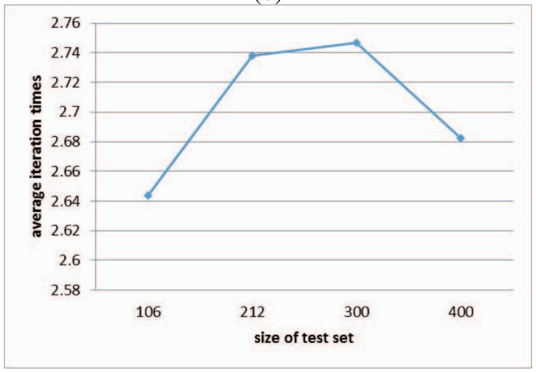
At present, we only use a system prototype to simulate users' actions and implement a demo for a few real users. We are trying to promote this to general users to collect more data to confirm the recommendation approach of the mashup development tool. While dealing with the multiple pipes, we separate them apart to several sequences to ease the complexity. However, in order to recommend more precisely, we consider to join the frequent sub-graph mining to our



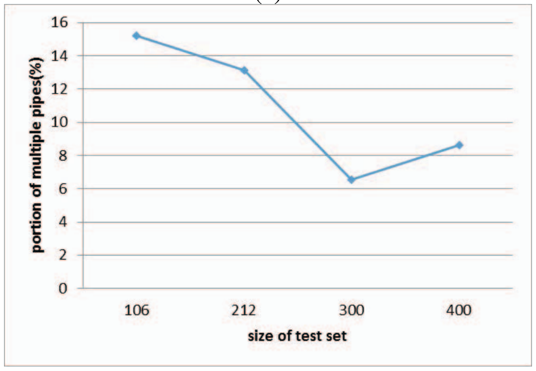
(a)



(b)



(c)



(d)

Figure 4: Performance with different sizes of the test set

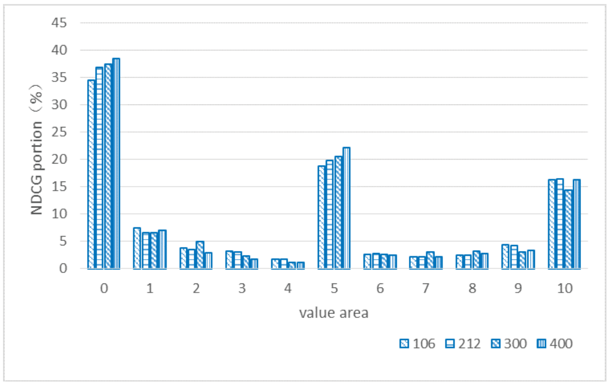
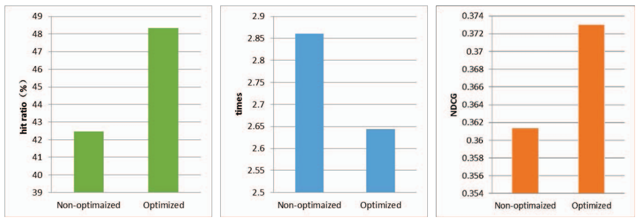


Figure 5: The distribution of $NDCG$



(a) hit ratio (b) iteration times (c) $NDCG$

Figure 6: Comparison of non-optimized and optimized algorithms

method in the future work. And the group of parameters may not be optimal in the simulation, since the evaluation can be improved as well. We also mine a number of useful data not been used in this study, while the data has the potential to support more efficient recommendation.

Acknowledgment

This work was supported by National Natural Science Foundation of China (No. 61103031, No. 61370057, No. 61300070), China 863 program (No. 2012AA011203), China 973 program (No. 2014CB340300, No. 2014CB340304), A Foundation for the Author of National Excellent Doctoral Dissertation of PR China(No.201159), and Beijing Nova Program (No.2011022).

References

[1] "Yahoo!pipes," http://pipes.yahoo.com/.

[2] "Programmableweb.com," http://www.programmableweb.com/.

[3] J. Han, M. Kamber, and J. Pei, *Data mining: concepts and techniques*. Morgan kaufmann, 2006.

[4] J. Wong, "Marmite: towards end-user programming for the web," in *Visual Languages and Human-Centric Computing, 2007. VL/HCC 2007. IEEE Symposium on*. IEEE, 2007, pp. 270–271.

[5] D. E. Simmen, M. Altinel, V. Markl, S. Padmanabhan, and A. Singh, "Damia: data mashups for intranet applications," in *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. ACM, 2008, pp. 1171–1182.

[6] R. J. Ennals and M. N. Garofalakis, "Mashmaker: mashups for the masses," in *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*. ACM, 2007, pp. 1116–1118.

[7] E. Bouillet, M. Feblowitz, Z. Liu, A. Ranganathan, and A. Riabov, "A tag-based approach for the design and composition of information processing applications," in *ACM Sigplan Notices*, vol. 43, no. 10. ACM, 2008, pp. 585–602.

[8] K. Goarany, G. Kulczycki, and M. B. Blake, "Mining social tags to predict mashup patterns," in *Proceedings of the 2nd international workshop on Search and mining user-generated contents*. ACM, 2010, pp. 71–78.

[9] J. Cao and C. Xing, "Data source recommendation for building mashup applications," in *Web Information Systems and Applications Conference (WISA), 2010 7th*. IEEE, 2010, pp. 220–224.

[10] O. Greenshpan, T. Milo, and N. Polyzotis, "Autocompletion for mashups," *Proceedings of the VLDB Endowment*, vol. 2, no. 1, pp. 538–549, 2009.

[11] H. Elmeleegy, A. Ivan, R. Akkiraju, and R. Goodwin, "Mashup advisor: A recommendation tool for mashup development," in *Web Services, 2008. ICWS'08. IEEE International Conference on*. IEEE, 2008, pp. 337–344.

[12] D. Deutch, O. Greenshpan, and T. Milo, "Navigating in complex mashed-up applications," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 320–329, 2010.

[13] S. R. Chowdhury, F. Daniel, and F. Casati, "Efficient, interactive recommendation of mashup composition knowledge," in *Service-Oriented Computing*. Springer, 2011, pp. 374–388.

[14] S. R. Chowdhury, "Assisting end-user development in browser-based mashup tools," in *Software Engineering (ICSE), 2012 34th International Conference on*. IEEE, 2012, pp. 1625–1627.

[15] S. Roy Chowdhury, O. Chudnovskyy, M. Niederhausen, S. Pietschmann, P. Sharples, F. Daniel, and M. Gaedke, "Complementary assistance mechanisms for end user mashup composition," in *Proceedings of the 22nd international conference on World Wide Web companion*. International World Wide Web Conferences Steering Committee, 2013, pp. 269–272.

[16] K. T. Stolee and S. Elbaum, "Refactoring pipe-like mashups for end-user programmers," in *Proceedings of the 33rd International Conference on Software Engineering*. ACM, 2011, pp. 81–90.