



A Tool for Mining Patterns based on Data Structure

Prachi Badera¹, Chetna Gupta²

Department of Computer Science

Jaypee Institute of Information Technology

A-10, Sector-62, Noida

INDIA

Abstract: The primary goal of a software developer is to produce and deliver high quality software systems which are produced at low cost and high productivity. In order to produce such software systems, programmers generally reuse existing libraries rather than developing the source code from scratch. However, while reusing a library, developers face many problems such as lack of knowledge about program interfaces and lack of time because of which effective test inputs cannot be generated during white box testing. The first two problems reduce software productivity whereas the latter affects software testing. To resolve these problems, we propose a tool named DSP (Data Structure Patterns). The tool will help users to search the relevant code, analyze it to find the most appropriate data structures and then mine the relevant patterns from the data structures and finally recommend the most suited pattern for use.

Keywords: Software library, Data Mining, Association Rules, Frequent Item sets, Lexical Analysis, Jacobian Coefficient

I. Introduction

The major concerns for those who are responsible for research and development of large software systems are: what affects software productivity and how do we improve it? The primary goal of a software developer is to produce and deliver high quality software systems, which are produced at low cost and productivity. In order to produce high quality software systems, programmers reuse existing libraries, rather than developing similar code from scratch [1].

Nowadays programmers develop systems from already built components by combining various modules from existing software systems to produce new one required by the user. One such program methodology is called opportunistic software systems development (OSSD), which works on the principal of reusing existing libraries [1] and helps in reducing the effort during software maintenance and also increases software productivity.

However, reusing existing libraries is not an easy task. Programmers who want to reuse existing libraries face many problems, for example, the large number of existing libraries contains many application programming interfaces (APIs), because of which it becomes difficult for the user to understand which module to choose. Secondly, many libraries are outdated or are not properly documented. In most of the cases, the user will straightforwardly follow the implicit rules, which will gradually lead to another problem i.e. of introduction of defects in the code [7]. Hence, the user (who is new to the library) will end up in confusion.

Therefore, in order to reuse an API, a user must have complete knowledge about the library before he starts to use it. Though, it is not possible for every user to sit back and understand each and every library in order to reuse just a small piece of API from the library. Therefore, to address this issue, we propose a tool named data structure patterns (DSP) and it would help users to view all the data structure patterns used in all the relevant codes of a software library and hence enable them to make decisions about the codes they should use for their own implementation purposes. The tool will provide a user with detailed analysis of relations among various data structures used within a program. This will help the user to find all possible patterns in all codes and then also recommend the pattern that is most used by other users. The processing of the tool takes place as follows: first it will search the relevant code, next it will analyze the searched codes to classify the information according to various data structures used in the searched codes. It is followed by mining of relevant patterns using data mining approaches within the data structures categorized in the previous step and finally it recommends the most suited pattern for use.

A lot of research has been done in this field. However, most of these tools have provided the users with the patterns among functions used in the codes [2, 12, 13, 14, 15, 18, 19]. Many of the tools have highlighted the concept of hierarchy, polymorphism and recursions among functions in their analysis [12, 14, 18]. DSP stands different to all of them as it provides analysis of data structures implemented. It first finds the relevant code from the library; analyzes the data structures used and finally mines and presents the patterns found in the various data structures detected in the codes.

The rest of the paper is organized as follows: section II gives a detailed literature survey. Section III sketches the proposed tool for DSP. Section IV presents a discussion of data collection and result observation. Section V describes the actual implementation done with the help of data and screenshots and section VI concludes the paper.

II. Related Work

The study conducted on papers listed in literature is divided into two parts: (a) code/document analysis (document here refers to a file containing only code) and (b) API recommendations.

A. *Brief overview of code/document analysis techniques*

Haiping Wu, Eunjung Park, Mihailo Kaplarevic, Yingping Zhang propose a machine learning method [8] for automated detection of performance sensitive program segments based on program segment similarity. Their technique involves a compiler which divides the input program into segments and correlates them with the proxy program segment. This proxy program segment is based on the similarity between the processed program segments and the proxy program segments. Another methodology proposed by Ottenstein [9] compares the basic structure of two codes based on the Halsted metrics. Donaldson, Lancaster and Sposato's system [10] on the other hand, uses eight attribute-counting metrics and their tool also generate a string representation of the program text. Anna Huang [11] describes measures for calculating document similarity which is calculated on the basis of distance with the help of coefficients such as cosine coefficients, Jaccard's coefficient, Pearson correlation coefficient. Steven Burrows, Seyed M. M. Tahaghoghi & Justin Zobe use the concept of indexing for finding the similarity between the two codes. Murphy and Notkin [3] use the concept of lexical analysis to generate source code models for matching two codes.

B. *Brief overview of techniques based on API recommendations*

API recommendation techniques mainly use two approaches: Approaches which recommend APIs by using mining techniques belong to the first category and approaches which use mining to search certain patterns or code snippets from sample code repositories belongs to the second. Prospector developed by Mandelin et al. [12] synthesizes on Jungloid graph to answer a query by providing the input and output types. Prospector traverses the possible paths from the input type to output type and recommends certain code snippets according to API signatures and a corpus of the client code. XSnippet developed by Sahavechaphan et al. [13] extends Prospector by adding more queries and ranking heuristics to mine code snippets from a sample repository. Context-sensitive is introduced to enhance the queries in XSnippet and results produced by this approach are more relevant.

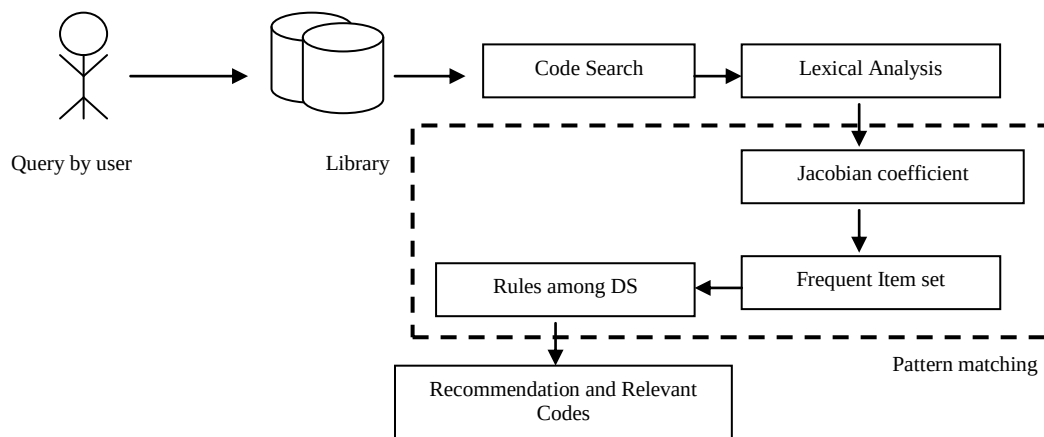
Strathcona developed by Holmes et al. [14] is dedicated to recommending code examples matching the structural context. Six heuristics are applied to obtain the structural context description in the stored repository. MAPO developed by Zhong et al. [15] takes the advantage of mining frequent usage patterns of an API with the help of code search engines. PARSEWeb developed by Thummalapenta et al. [16] mines open source repositories by using code search engines. Different from MAPO, PARSEWeb accepts the queries of the form "Source type" and "Destination type" which suggest the relevant methods that yield the object with the destination type.

The approaches which use mining to search certain patterns or code snippets aim at recommending APIs with respect to structural dependency. Zhang et al. propose a random-walk approach [17] based on the PageRank algorithm to rank "popular" and "significant" program elements in Java programs. Inoue et al. proposed another approach [18] inspired by PageRank algorithm, which can be employed to rank valuable components in software systems based on use relations. Suade [19] developed by Robillard is focused on providing suggestions for aiding program investigation. It accomplishes suggestion by ranking desired program elements concerned with topological properties of structural dependency in software systems. Fran [20] developed by Saul et al. extends topological properties by considering neighboring relationships in the call graph. Altair developed by Long et al. [21] recommends the relevant APIs according to overlap of commonly accessed variable information.

III. Proposed Tool Sketch for DSP

The working of the tool is divided into following steps: searching, code analysis, pattern matching and result depiction. The tool sketch for the concept of finding relevant and interesting data structure patterns used by various users is explained in Figure 1 above followed by discussion on each of its phases of narrowing down in predicting efficient results of searching, code analysis, pattern matching and finally result depiction.

The user is required to specify the query to the tool. The query format is as specified in the following sections. According to the query, a code is searched in the static library as created on the system. The retrieved codes are analysed to find the set of data structures used in the implemented code. Based on the frequency of a particular data structure being used, frequent item sets are formed. Using these sets, various rules are inferred which clears the implementation of a higher order data structure with the help of basic data structures. Then the corresponding codes are displayed to the user in accordance with the rules generated. The detailed description is as follows:

Figure 1 Steps followed in DSP

A. Code Searching

Query defined by the user - A user can type any problem which is solved with the help of data structures. These problems can be any for example the popular problems like 'Polish Notations', 'Round Robin Scheduling', 'Tower of Hanoi', etc.

Library: A library containing C codes is maintained at the hard drive of the system. This library consists of folders in which the codes are segregated based on the problem they are implemented for. According to the query by the user, code is searched in the library based on the keywords as input by the users. In a separate folder standard codes for data structures such as a stack, queue, binary tree, basic linked list, etc. are maintained.

B. Code Analysis

A large number of codes are maintained in the library as stated above. The codes are retrieved matching the query given by the users. In this phase, the code is analyzed in order to determine which data structure is used in order to solve the problem. The analysis is done for all the standard codes i.e. stack, queue, binary tree, etc, and also for the retrieved codes. Code analysis can be done with the help of many techniques as described above. The techniques we have used involves various steps which are as follows:

Step 1-- Lexical Analysis: The code is parsed and tokenized [4]. This step is called lexical analysis of the code. The basic tokens used are defined in the Table I below:

Table I Defined Keywords

int, float, for, while, break, etc.	Keywords
({ }) ; ,	Delimiter
++, --, +, -, *, /, <=, >=, ==	Operators
//	Comment
#include	directive
Anything other than above	Identifier

These tokens are further shortened for easy computation and are stored as 'k' for the keyword, 'd' for the delimiter, 'o' for operators, 'c' for the comment, 'x' for directive and 'id' for the identifier.

Step 2 -- Grouping Tokens: In the second step, the tokens formed are represented by an alphanumeric character to form a simple sequence there by preserving the structure of the program. Fingerprints are created using n-gram technique. N-gram technique fights efficiently against adding redundant statements in the code [5]. N-grams are calculated like shown in this example: Suppose the sequence generated is KDOCDDKCCOOKX. The 3-grams generated (by sliding window containing 3 tokens at a time) will be KDO DOC OCD CDD DDK DKC KCC CCO COO OOK OKX. So the additional statement will only affect the neighbouring n-grams.

Step 3 -- Code Similarity: A lot number of tools exist for finding the code similarity which can be found in the related work section. In this step, the sequences generated are compared and using the formula of Jaccard Coefficient [5], similarity is predicted.

$$\text{Jaccard's similarity coefficient } (p1, p2) = \frac{|p1 \cap p2|}{|p1 \cup p2|}$$

It is necessary that the Jaccard's coefficient value is greater than or equal to the threshold value in order to accept the prediction of similarity.

C. Pattern Matching & Rule Determination

Any item is said to be a frequent item if its frequency is greater or equal to the threshold support count as specified by the user. A pattern among these frequent item sets is found with the help of data mining technique called Association Rules [22].

Once the data structures implemented are found (as explained in 2.2), frequent item sets are found from this data based on the comparison of the frequency of occurrence of a particular data structure with the threshold support count. Association rules are a technique used to find various relations such as inter dependability among item sets and hence predict the compatibles patterns [6].

The stability of a rule can be supported by two parameters namely confidence and support. For a rule $X \Rightarrow Y$, confidence is defined as $c\%$ if $c\%$ of transaction set that contain X also contain Y . And the parameter support is defined to be $s\%$ in the transaction set if $s\%$ of transactions contains $X \cup Y$.

D. Result Depiction

Based on the rules found, one with better support and confidence is recommended to the user and the relevant codes are suggested by the proposed tool. Any rule with low support and low confidence or high support and low confidence is rejected by the tool. Only the rules with high support and high confidence are mainly accepted. Sometimes, rules with low support and high confidence are also accepted (in case, there are no rules with high support and high confidence).

IV. Data Collection and Result Observation

A basic transaction set consists of the set of data structures that a file contains. For example, consider the code given below in Figure 2. From the code given below, use of structure and static array can explicitly be seen from lines 5 and 6 respectively. The use of stack can be found after analyzing the push function (line 8-14) and pop function (line 15-20) with the help of lexical analysis and Jaccard Coefficient. Hence, the data structures used for this code will be static array, structure, stack. Similarly, all the relevant codes are analyzed and a transaction set containing the names of the data structures is stored. Table II shows the analysis and the set of transactions.

Table II Transactions

Id	Transaction
101	Static Array, Structure
102	Dynamic Array, Structure
103	Static Array, Structure, Stack
104	Dynamic Array, Structure, Stack
105	Static Array, Structure, Queue
106	Dynamic Array, Structure, Queue

A. Frequent Item Sets

After the code has been analyzed and the data structures used have been deduced, a count of all the data structures used is maintained and is shown in Table III below.

Table III Frequent Item sets

Item	Frequency
Static Array	3
Dynamic array	3
Structure	6
Stack	2
Queue	2

Assuming that the minimum support is specified as 20%, support count = 1. Therefore, in the above table (Table 3), only the items with support count greater than 1 are stored. After analyzing Table 3 2-k item is set is computed and the same is shown in Table 4 in which combination of two elements taken at time is called the item set. Then again support or support count (frequency) is calculated.

\

Figure 2 Sample code

```

1. #include<stdio.h>
2. int top=-1;
3. int maxsize=19;
4.
5. struct stack
6. { char ch[maxsize];
7. }s;
8. void push(char c)
9. { ++top;
10. if(top>=maxsize)
11.     printf("\nOverflow");
12. else
13.     s.ch[top]=c;
14. }
15. char pop()
16. { if(top<0)
17.     { printf("Underflow");
18.     return 0;
19.     }
20. else
21.     return s.ch[top--];
22. }
23. int prec(char c1, char c2)
24. { char cha;
25. if(((int)c1)>((int)c2))
26.     return 0;
27. else
28.     return 1;
29. }
30.
31. void main()
32. { char c[19];
33. char ans[19],chr;
34. int kop;
35. int k=0,p,m=0;
36. printf("\nEnter the infix exp:");
37. scanf("%s",c);
38. while(c[k]!='\0')
39. { p=(int)c[k];
40. if((c[k]=='[')||(c[k]=='('))
41.     push(c[k]);
42. else if((p>96)&&(p<=122))
43.     { ans[m]=(char)p;
44.     m++;
45.     }
46. else if((c[k]=='+')||(c[k]=='-')||(c[k]=='*'))
47.     { kop=prec(c[k],s.ch[top]);
48.     if (kop==0)
49.     { push(c[k]);
50.     }
51.     else if(kop==1)
52.     { chr = pop();
53.     ans[m]=chr;
54.     m++;
55.     }
56.     }
57. else if((c[k]==')') || (c[k]==']'))
58.     { chr=pop();
59.     if(chr!='(')
60.     { ans[m]=chr;
61.     m++;
62.     }
63.     }
64.     k++;
65. }
66. m=0;
67. while(ans[m]!='\0')
68. { printf("%c",ans[m]);
69. m++;
70. }
71. getch();
72. }

```

Table IV 2-k item Sets

ID	Item	Support
100	Static Array, Dynamic Array	0
101	Static Array, Structure	3/6=50%
102	Static Array, stack	1/6
103	Static Array, Queue	1/6
104	Dynamic Array, Structure	3/6=50%
105	Dynamic Array, stack	1/6
106	Dynamic Array, Queue	1/6
107	Structure, Stack	2/6=33%
108	Structure, Queue	2/6=33%
109	Stack, Queue	0

Some of the item sets get pruned as their support are not higher than the minimum support. Therefore, the item set with ID 100, 102, 103, 105, 106 and 109 are pruned. The resultant table is shown below (Table V).

Table V After pruning 2-Item sets

ID	Item	Support
200	Static Array, Structure	3/6=50%
201	Dynamic Array, Structure	3/6=50%
202	Structure, Stack	2/6=33%
203	Structure, Queue	2/6=33%

In the next step 3-item sets will be formed by re-pairing of item sets and their support and confidence is calculated. The results of 3-item sets are shown in Table VI below.

Table VI 3-Item sets

ID	Item	Support
300	Static Array, Structure, Dynamic Array	0
301	Static Array, Structure, Stack	2/6=33%
302	Static Array, Structure, Queue	2/6=33%
303	Dynamic Array, Structure, Stack	2/6=33%
304	Dynamic Array, Structure, Queue	2/6=33%

As we can see in the above table, item set with ID 100 is ruled out and also no further combination of item sets can be made since the further combinations of 4 item sets have support < 20%. Therefore, following rules are generated which are shown in Table VII.

Table VII Possible Rules

Rule 1	Static Array – Structure, Stack
Rule 2	Structure – Static Array, Stack
Rule 3	Stack -- Static Array, Structure
Rule 4	Static Array – Structure, Queue
Rule 5	Structure – Static Array, Queue
Rule 6	Queue -- Static Array, Structure
Rule 7	Dynamic Array – Structure, Stack
Rule 8	Structure – Dynamic Array, Stack
Rule 9	Stack -- Dynamic Array, Structure
Rule 10	Dynamic Array – Structure, Queue
Rule 11	Structure – Dynamic Array, Queue
Rule 12	Queue -- Dynamic Array, Structure

B. Calculating Support and Confidence for the above rules

In order to find which rule is a valid rule, support and confidence for every rule needs to be calculated. Table VIII presents the support and confidence is rules below.

Table VIII Support, Confidence of Rules

Rules	Support	Confidence
Static array – structure, stack	2/6=33%	2/6=66.67%
Structure – static array, stack	2/6=33%	2/6=33%
Stack – static array, structure	2/6=33%	3/3=100%

Support of rule ($X \rightarrow Y$) is calculated as $\text{count}(X \cup Y) / \text{total number of transactions}$ and confidence of the rules ($X \rightarrow Y$) is calculated as $\text{count}(X \cup Y) / \text{count}(X)$. Therefore, out of rules 1,2 and 3, rule 1 and 2 are discarded and rule 3 is accepted.

Similarly, confidence and support can be calculated for other rules and out of all the rules, the best rules can be selected based on its support and confidence percentage.

V. Implementation

The above tool has been implemented in Java (Net beans) and the basic requirement of this tool is the existence of library in the system on which it is to be run. The working of the tool is depicted using snapshots discussed in section below.

In Figure 3, the user inputs the problem as Tower of Hanoi. In this phase, the code from the existing library is searched and is processed to find the patterns in it. Later rules are developed for same. Below in the textbox are the rules displayed along with their support and confidence. Also, the most suitable rule is highlighted as being recommended by the application on its own.

When the user checks the view recommended box, codes containing the solution of Tower of Hanoi are displayed as depicted in Figure 4 i.e. the codes with solution of Tower of Hanoi using the rule which is recommended by the application are shown as links to the user. Hence, the user can directly access these codes by

clicking on the links. However, if the user wants to access the other codes, he can click on the view codes option. This will redirect him to the location where all the codes are stored.

Figure 3 Screenshot after user input

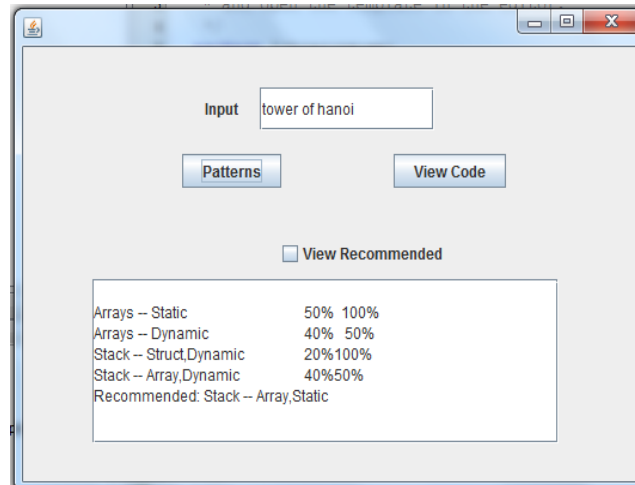
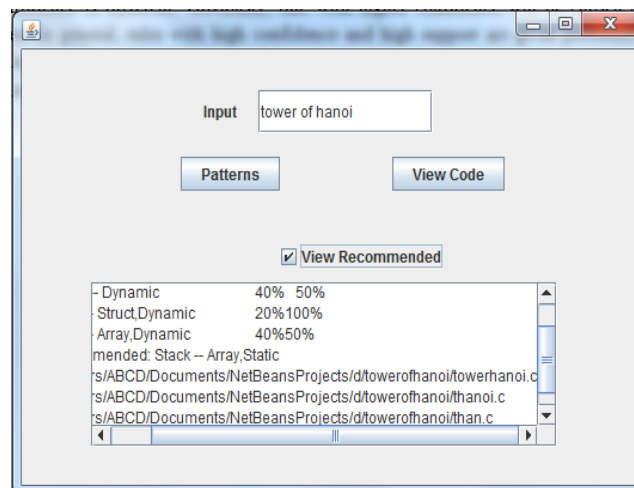


Figure 4 Screenshot with recommended rules



VI. Conclusion

We have successfully implemented our tool DSP (data structure patterns) which describes the relations among the set of data structures as used in the code and also recommends which rule will be followed in order to implement a particular problem. However, the analysis of the source code in order to know which data structure has been used gives a predicted answer. That is, there is not complete confidence in finding out the data structure. However, the answers generated are very much closer to the actual answers.

References

- [1] T.M. Krishna and V. Devara, "A Study of Mining Software Engineering Data and Software Testing" Journal of Emerging Trends in Computing and Information Sciences, vol. 11(2), Oct. 2011, pp. 598-603.
- [2] S. Agrawal, "Life Cycle Model for Software Reusability" Proc. Int. Conference on Technology and Business Management, Mar. 2012.
- [3] M. Halkidi, D. Spinellis, G. Tsatsaronis and M. Vazirgiannis, "Data Mining in Software Engineering" Journal of Intelligent Data Analysis, vol. 3(15), Aug. 2011, pp. 413-441, doi: 10.3233/IDA-2010-0475
- [4] E. Park, M. Kaplarevic, Y. Zhang, M. Bolat, X. Li and G.R. Gao, "Automatic Program Segment Similarity Detection in Targeted Program Performance Improvement", Proc. 21st International Parallel and Distributed Processing Symposium, Mar. 2007, pp.1-8, doi: 10.1109/IPDPS.2007.370642.
- [5] K.J. Ottenstein, "An Algorithmic Approach to the Detection and Prevention of Plagiarism" Proc. 6th SIGCSE Symposium on Computer Science Education, vol. 8(4), Dec. 1976, pp. 30-41, doi: 10.1145/382222.382462.
- [6] J.L. Donaldson, A.M.Lancaster and P.H. Sposato, "A Plagiarism Detection System" Proc. Twelfth SIGCSE Technical Symposium. Vol. 1(1), Feb. 1981, pp. 21-25.

- [7] H. Anna, "Similarity Measures for Text Document Clustering" Proc. New Zealand Computer Science Research Student Conference, April 2008, pp. 49-56.
- [8] G. Murphy, and D. Notkin, "Lightweight source model extraction" Symposium on the Foundations of Software Engineering, Oct 1995, pp. 116-127, doi: 10.1145/222132.222147.
- [9] D. Mandelin, L. Xu, R. Bod'ik, D. Kimelman, "Jungloid mining: helping to navigate the API jungle" Proc. 2005 ACM SIGPLAN conference on Programming language design and implementation, vol. 40(6), June 2005, pp. 48-61, doi: <http://acm.org/10.1145/1065010.1065018>
- [10] N. Sahavechaphan, K.T. Claypool, "XSnippet: mining for sample code" Proc. 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications, vol. 41(10), Oct. 2006, pp. 413-430, doi: <http://acm.org/10.1145/1167473.1167508>
- [11] R. Holmes, and G.C. Murphy, "Using structural context to recommend source code example" Proc. 27th international conference on Software engineering, May 2005, pp. 117-125, doi: 10.1109/ICSE.2005.1553554.
- [12] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei, "MAPO: Mining and Recommending API Usage Patterns" Proc. 23rd European Conference on ECOOP 2009-- Object-Oriented Programming, 2009, pp. 318-343, doi: 10.1007/978-3-642-03013-0_15.
- [13] S. Thummalapenta, and T. Xie, "PARSEWeb: A Programmer Assistant for Reusing Open Source Code on the Web" Proc. twenty-second IEEE/ACM international conference on Automated software engineering, Nov. 2007, pp. 204-213, doi: <http://acm.org/10.1145/1321631.1321663>
- [14] C. Zhang, and H.A. Jacobsen, "Efficiently mining crosscutting concerns through random walks" Proc. 6th International conference on Aspect-oriented software development, April 2007, pp. 226-238, doi: 10.1145/1218563.1218588.
- [15] K. Inoue, R. Yokomori, H. Fujiwara, T. Yamamoto, M. Matsushita, and S. Kusumoto, "Component rank: Relative significance rank for software component search" Proc. 25th International Conference on Software Engineering, May 2003, pp.14-24.
- [16] M.P. Robillard, "Automatic generation of suggestions for program investigation" Proc. 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on foundation of software engineering, Sept. 2005, pp. 11-20, doi: <http://acm.org/10.1145/1081706.1081711>
- [17] Z.M. Saul, V. Filkov, P.T. Devanbu, and C. Bird, "Recommending random walks" Proc. 6th Joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering, Sept. 2007, pp. 15-24, doi: <http://acm.org/10.1145/1287624.1287629>
- [18] F. Long, X. Wang, and Y. Cai, "API hperlinking via structural overlap" Proc. 7th Joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering, Aug. 2007, pp. 203-212, doi: <http://dx.doi.org/doi:10.1145/1595696.1595727>
- [19] B. Karthikeyan, "Similarity Detection in Source Code Using Data Mining Techniques" European Journal of Scientific Research, vol. 62(4), Oct. 2011, pp. 500-505.
- [20] N. Haritha, M. Bhavani and K.T. Reddy, "C Code Plagiarism Detection System" International Journal of Science and Advanced Technology, vol. 1(5), July. 2011, pp. 198-203.
- [21] Y. Xu, Y. Li and G. Shaw, "Reliable representations for association rules" Journal of Data & Knowledge Engineering, vol. 70(6), June 2011, pp. 555-575, doi: 10.1016/j.datak.2011.02.003
- [22] T. Xie, S. Thummalapenta, D. Lo, C. Lui, "Data Mining for Software Engineering" Journal of Computer, vol. 42(8), Aug. 2009, pp. 55-62, doi: <http://doi.ieeecomputersociety.org/10.1109/MC.2009.256>.