An Approach to Improving Bug Assignment with Bug Tossing Graphs and Bug Similarities

Liguo Chen
School of Computer Science and Engineering
Beihang University
Beijing, China
Email: netclg@gmail.com

Xiaobo Wang
School of Computer Science and Engineering
Beihang University
Beijing, China
Email: bobywolf@gmail.com

Chao Liu
School of Computer Science and Engineering
Beihang University
Beijing, China
Email: liuchao@buaa.edu.cn

***Abstract*—In open-source software development a new bug firstly is found by developers or users. Then the bug is described as a bug report, which is submitted to a bug repository. Finally the bug triager checks the bug report and typically assigns a developer to fix the bug. The assignment process is time-consuming and error-prone. Furthermore, a large number of bug reports are tossed (reassigned) to other developers, which increases bug-fix time.**

In order to quickly identify the fixer to bug reports we present an approach based on the bug tossing history and textual similarities between bug reports. This proposed approach is evaluated on Eclipse and Mozilla. The results show that our approach can significantly improve the efficiency of bug assignment: the bug fixer is often identified with fewer tossing events.

***Index Terms*—bug assignment, bug reports, bug tossing, information retrieval, software engineering**

I. Introduction

The developers and users commonly submit bug reports to a bug repository (e.g. Bugzilla [1]) in open source software development. The bug triager, a person who decides what to do with an incoming bug report, assigns a developer to a new bug report. If the bug can't be resolved by the developer, for example because the bug has been assigned by mistake, it is tossed to another developer. The tossing process continues until the bug report reaches the bug resolver. The bug tossing not only increases bug fix time, but also it wastes time of bug triagers and developers. For instance in Eclipse and Mozilla, about 37%-44 % of bug reports are tossed to other developers, and one tossing event takes an average of 50 days [2]. Therefore, quick identification of resolvers to bug reports can improve the efficiency of bug fixing. Due to the large number of existing bug reports, it is challenging for the triager to examine all existing bug reports to assign developers for fixing these bugs.

The bug tossing process is reflected by the activities of bugs. Tbl. I shows a sample bug activity, in which the bug firstly be assigned to Randy_Giffen, then is tossed to Ian_Petersen (the bug resolver). The efficiency of identifying the bug resolver depends on deciding if the current bug holder can't solve the bug, which developer the bug should be transferred to next.

Jeong et al. [3] introduce a graph model based on Markov chains, which captures bug tossing history. Although their model reduces tossing events, by up to 72%, there is still a need to improve it due to the high search failure rate (up to 60%). In this paper, we introduce an approach that uses both the tossing graph and vector space model for calculating textual similarities between bug reports. This proposed approach is evaluated on Eclipse and Mozilla. Experimental results indicate that our approach achieves a significant improvement in reducing tossing events and search failure rate over previous approaches. This paper makes the following contributions:

- construct the tossing graph by extracting the bug tossing sequences.
- calculate the textual similarities between bug reports using vector space model.

doi:10.4304/jsw.6.3.421-427

TABLE I.
A SAMPLE BUG ACTIVITY

Who	When	What	Removed	Added
nick_edgar	2002-1-27	AssignedTo	Kevin_Haaland	Randy_Giffen
Randy_Giffen	2002-1-30	AssignedTo	Randy_Giffen	Ian_Petersen
Ian_Petersen	2002-1-31	Status	NEW	RESOLVED
		Resolution		Fixed

- present an approach to improving bug assignment with bug tossing graph and bug similarities.
- conduct experiments on the popular open-source projects Eclipse and Mozilla to verify our approach and demonstrate that this approach can effectively reduce bug tossing events. Thus it can be adopted as a recommendation tool for bug assignment.

The remainder of the paper is organized as follows. Section II surveys related work. Section III introduces background knowledge used in our approach. Section IV presents main steps of our approach. Section V sets up experiments and presents the results of applying the approach to the Eclipse and Mozilla projects. Section VI discusses the limitations of our study. Finally, section VII concludes the paper and outlines future work.

II. RELATED WORK

Bug tossing is similar to the ticket routing ("transferring problem ticket among various expert groups in search of the right resolver to the ticket" [12]). Shao et al. [12] use Markov model to model the ticket routing and present a search algorithm to search the problem resolver. Instead of their search algorithm we simply apply Weighted Breadth First Search algorithm (WBFS) [9] to the tossing graph, because in my case their search algorithm is ineffective (often consume a longer time to find the target node). Like our work, some researchers use probabilistic models to mine workflow from activity logs in the machine learning literature [13–18].

Bug triage mainly includes two activities in open-source software development. One is to detect the duplicate bugs. Another is to assign developers to bug reports. There have been some approaches to automate detecting duplicate bugs reported in the literature. For example, some researchers [4,5,6] identified duplicate bugs by calculating textual similarities between the new bug report and existing bug reports with vector space model. Our approach also uses bug similarities, but which are applied to the bug tossing graph.

To automate bug assignment, Anvik et al. [2] used text categorization approach, which applied Support Vector Machine (SVM) to recommend to the bug triager a set of developers who may be appropriate bug resolvers

TABLE II.
SAMPLE TOSSING PATHS

$A \rightarrow B \rightarrow C \rightarrow E$
$A \rightarrow C \rightarrow B$
$C \rightarrow D \rightarrow E$
$B \rightarrow C \rightarrow F \rightarrow E$

TABLE III.
DECOMPOSED SINGLE STEPS FROM THE SAMPLE TOSSING PATHS IN TBL II. THE NUMBERS IN PARENTHESIS INDICATE THE OCCURRENCES OF EACH SINGLE STEP

$A \rightarrow E$(1)	$A \rightarrow B$(1)
$B \rightarrow E$(2)	$C \rightarrow B$(1)
$C \rightarrow E$(3)	$D \rightarrow E$(1)
$F \rightarrow E$(1)	

for each new bug. Cubranic and Murphy [7] applied Naive Bayes classification techniques to assist in bug assignment by using text categorization to predict the developer that should work on the bug based on the bug's description. The two methods only exploited the text information of bug reports. Jeong et al. [3] presented a different method, in which a tossing graph was constructed by capturing bug tossing history and was optimized by two model options, then the Weighted Breath First Search (WBFS) algorithm was employed to detect the bug resolver from the tossing graph. However, their approach resulted in about 50% to 60% search failure rate, because by applying two options to the original tossing graph a number of edges were deleted. Instead of deleting edges for each new bug report we use textual similarities between bug reports to delete unrelated nodes (developer) in original tossing graph, which results in lower search failure rate and improves the efficiency of bug assignment.

III. BACKGROUND

In our approach, we deal with bug tossing history and bug textual information. The section presents how to construct bug tossing graph from bug tossing history and how to calculate bug similarity from bug textual information.

A. Bug Tossing Graph

Definition 3.1 (Bug Tossing Path): A bug is assigned to the first developer d_1, then it is reassigned to other developers until it reaches the fixer d_f. Let the set of developers to whom the bug is assigned T=$d_1 \rightarrow d_2 \rightarrow ... \rightarrow d_f$ denotes a bug tossing path [3].

From the activities of bugs we extract the bug tossing path, which are used to generate the bug tossing graph. Tbl. I shows a sample bug activity, where the tossing path is Randy_Giffen $\rightarrow$Ian_Petersen, where Ian_Petersen is the bug fixer.

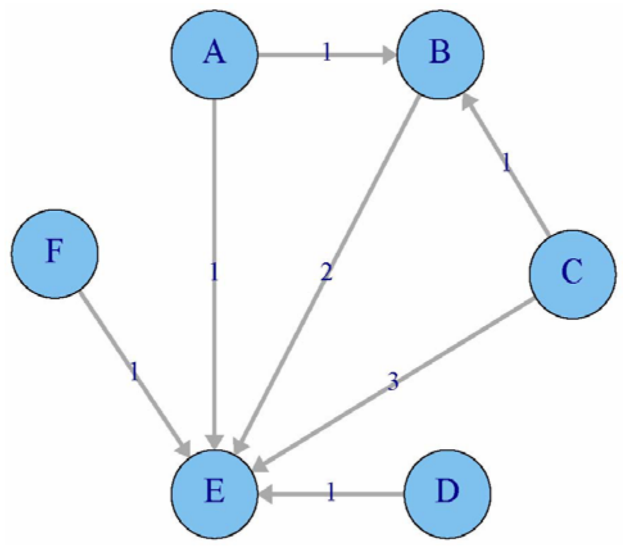
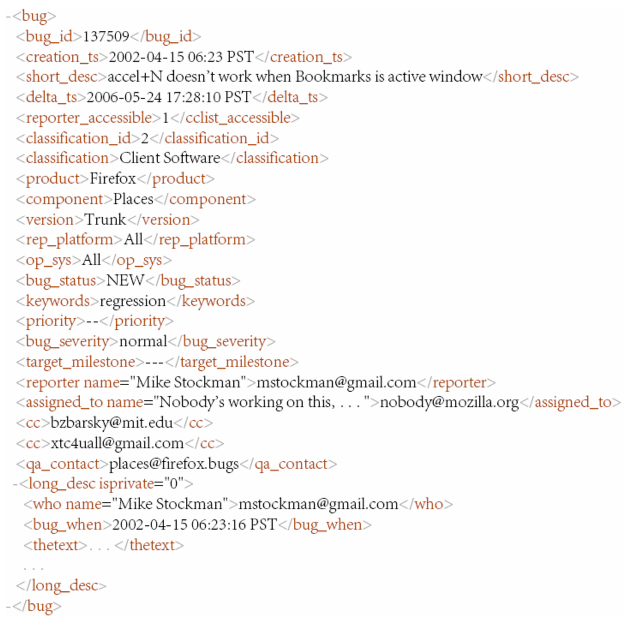


Figure 1. A tossing graph generated from the decomposed steps in Tbl. III.



```
-<bug>
  <bug_id>137509</bug_id>
  <creation_ts>2002-04-15 06:23 PST</creation_ts>
  <short_desc>accel+N doesn’t work when Bookmarks is active window</short_desc>
  <delta_ts>2006-05-24 17:28:10 PST</delta_ts>
  <reporter_accessible>1</cclist_accessible>
  <classification_id>2</classification_id>
  <classification>Client Software</classification>
  <product>Firefox</product>
  <component>Places</component>
  <version>Trunk</version>
  <rep_platform>All</rep_platform>
  <op_sys>All</op_sys>
  <bug_status>NEW</bug_status>
  <keywords>regression</keywords>
  <priority>--</priority>
  <bug_severity>normal</bug_severity>
  <target_milestone>---</target_milestone>
  <reporter name="Mike Stockman">mstockman@gmail.com</reporter>
  <assigned_to name="Nobody’s working on this, . . . ">nobody@mozilla.org</assigned_to>
  <cc>bzbarsky@mit.edu</cc>
  <cc>xtc4uall@gmail.com</cc>
  <qa_contact>places@firefox.bugs</qa_contact>
 -<long_desc isprivate="0">
   <who name="Mike Stockman">mstockman@gmail.com</who>
   <bug_when>2002-04-15 06:23:16 PST</bug_when>
   <thetext>. . . </thetext>
   . . .
  </long_desc>
-</bug>
```

Figure 2. A bug report stored in Bugzilla.

Suppose there is a tossing path, $A \rightarrow B \rightarrow C \rightarrow D \rightarrow E$, where E is the bug fixer. We can decomposes the path to goal oriented steps [3], $A \rightarrow E$, $B \rightarrow E$, $C \rightarrow E$, and $D \rightarrow E$. Tbl. II shows sample tossing path, and Tbl. III displays the decomposed steps from the sample paths. As shown in Tbl. III the numbers in parenthesis represent the occurrences of each single step. For example, there are two steps $B \rightarrow E$ and one step $F \rightarrow E$.

We can generate the tossing graph from the decomposed single steps (see Tbl. III). Fig. 1 shows the tossing graph generated from steps in Tbl. III. The graph expresses tossing relationship. For example, the directed edge <B, E> in Fig. 1 corresponds to the tossing step $B \rightarrow E$ in Tbl. III, and the edge weight corresponds to the occurrences of the step.

TABLE IV.
INFORMATION FIELDS IN A BUG REPORT SAMPLE

Field	**Explanation**
short_desc	the short description of the bug
reporter	the person who reported the bug
classification	which classification the bug concerns
component	which component the bug concerns
product	which product the bug concerns

The tossing graph expresses the tossing relationship. For example, Fig. 1 indicates E is a good candidate to toss a bug from C, since C and E have stronger tossing relationship than those of others.

B. Bug Similarities

The bug reports are written in natural language. By information retrieval model we can calculate bug textual similarities. The vector space model [8] is widely used in information retrieval domain. We apply vector space model to calculate the textual similarities between the new bug and existing bugs. Firstly, we introduce the foundational knowledge of vector space model. Then we show how to calculate the bug similarities with vector space model.

1) Vector Space Model (VSM): In VSM each document or each query is represented as a n-dimension vector ($w_1, w_2, ..., w_n$), where n is the number of unique words (or terms) appearing in all the documents or queries. The ith element w_i is a measure of the weight of the ith word in the vector.

After transforming documents and queries into vectors, we can calculate the similarity of a pair of documents or queries through cosine formula. Given two vectors q_1 = (w_{11}, w_{12} , ..., w_{1n}) and q_2 = (w_{21}, w_{22} , ..., w_{2n}), the cosine similarity of q_1 and q_2 is defined by Formula (1)

$$\text{Similarity}(q_1, q_2) = \frac{\sum_{i=1}^{n} w_{1i} w_{2i}}{\sqrt{\sum_{i=1}^{n} w_{1i}^2 \times \sum_{i=1}^{n} w_{2i}^2}} \qquad (1)$$

2) Calculating Bug Similarities with VSM: Software bug reports usually are stored in the bug repository systems (e.g., Bugzilla, JIRA and CollabNet) [5]. A bug report stored in one of these repositories consists of a number of fields. For example, a bug report stored in Bugzilla contains bug_id, short_desc, reporter, creation_ts, , long_desc fields and etc (see Fig. 2). Tbl. IV lists some information fields which are used to calculate similarities between bugs.

Calculating the similarity between two bugs with vector space model is very direct, which follows below several steps:

Algorithm 1 Subgraph(G, Bug, Bugset, T)

```
Input:
    $G$: the tossing graph;
    $Bug$: the new bug;
    $Bugset$: the set of existing bugs;
    $T$: the similarity threshold;
Output:
    the subgraph of the tossing graph
1: $L \leftarrow ()$    {Empty bug set}
2: $NodeSet \leftarrow ()$    {Empty node set}
3: for each bug $b \in Bugset$ do
4:     Calculate the similarity between $bug$ and $b$, obtaining the similarity $S$;
5:     if $S \geq T$ then
6:         Add $b$ to set $L$;
7:     end if
8: end for
9: for each bug $b \in L$ do
10:    find the corresponding the developer (the node) who resolved bug $b$ in bug repository, add the node to set $NodeSet$
11: end for
12: return The subgraph of the tossing graph that only contains the nodes that exist in set $NodeSet$;
```

- extract the unique words from bug fields information (see Tbl. IV). Among these words many words, like the, by and when (called stop words [8]), are not likely to be useful to measure the similarity of bugs. Therefore we exclude these stop words.
- calculate the weight of every word occurring in every bug report using the method of term frequency/ inverse document frequency [7].
- calculate the similarity of bug reports with formula (1).

IV. The Proposed Approach

A. *Leveraging Bug Tossing Graph*

In Eclipse and Mozilla more than 35% of bugs have at least one tossing event [3]. Our approach aims to reduce the tossing events as far as possible. Suppose the original tossing path is A→B→C→D, where the first assigner is A and the fixer is E. Given the first assigner, we can search for the fixer by applying Weighted Breadth First Search algorithm (WBFS) [9] to the tossing graph. If we find a path A→C→D, the tossing events of the path are reduced to 3 from 4. Unfortunately, when the tossing graph is huge, the path that we get by search the target node is usually long. In order to reduce the length of path a method is to narrow the search space. Jeong et al. [3] used two options to delete some edges of the tossing graph. However, their method resulted in high search fail rate.

Algorithm 2 bugAssignment(G, Bugset1, Bugset2, T)

```
Input:
    $G$: the tossing graph;
    $Bugset1$: the set of new bugs ;
    $Bugset2$: the set of existing bugs;
    $T$: the similarity threshold;
Output:
    $L$: total tossing path length;
    $R$: search failure times;
1: $L \leftarrow 0$;
2: $R \leftarrow 0$;
3: Optimize the tossing graph $G$;  {see section IV}
4: for each bug $b \in Bugset1$ do
5:     $targetNode \leftarrow$ extract the fixer of the bug b;
6:     $subgraph \leftarrow subgraph(G, b, Bugset2, T)$;
7:     search $targetNode$ applying the $WBFS$ algorithm to $subgraph$;
8:     if successfully find the target then
9:         $len \leftarrow$ count the length of path which is passed;
10:        $L \leftarrow L + len$;
11:    else
12:        $R \leftarrow R + 1$;{consider the current bug assignment as a failure}
13:    end if
14: end for
15: return
    total tossing path length $L$;
    search failure times $R$;
```

B. *Optimizing Bug Tossing Graph*

From the history of bug assignment we can construct bug tossing graph, but the graph need to be optimized. It is possible that some of the developers are retired and do not fix bugs in future. In open source software development most of developers can freely leave the developer groups and the project repository don’t record the retired developers. We think developers who don’t fix any bugs in recent long time are very likely to be retired developers. After identifying the retired developers we can delete the corresponding nodes of the original tossing graph.

C. *Pruning Bug Tossing Graph against Every New Bug*

We observe the bug reports in bug repository and find that similar bugs tend to be fixed by the same developer. For example, a developer who is responsible for developing a software component might only care the bugs of the component, So he might refuse fixing the bugs of other components. Base on the discovery we present a method to deleting a great number of unrelated nodes of the tossing graph. The method is described in algorithm 1 in detail. For instance, given a new bug, firstly the similarities between the new bug and existing bugs can be calculated with VSM. Secondly, with Algorithm 1 we obtain the corresponding subgraph of the tossing graph.

Above, we present the three components of our approach. Algorithm 2 describes how to assign developers to new bug with our approach in detail.

V. Experiments

A. Measuring the Performance of Our Approach

Definition 5.1 (Mean Length of Tossing Paths (MLTP)): Given m tossing paths sequence $\{T_i\}_{i=1,2,\ldots,m}$. The *Mean Length of Tossing Paths* is defined as follows:

$$MLTP = \frac{\sum_{i=1}^{m} |T_i|}{m}$$

The *MLTP* depends on the similarity threshold. The high threshold results in low *MLTP*. The *MLTP* measures the efficiency of bug assignment with our approach. The more length MLTP is, the more low the efficiency of bug assignment is.

Definition 5.2 (Failure Rate (FR)): Given n bug reports, in which m bug reports are correctly assigned to bug fixers using our approach, then FR is calculated by the following formula:

$$FR = \frac{n-m}{n}$$

Given a new report, it is possible that our approach can't find the target node in the bug tossing graph in following cases:

- The fixer of the given new report is new, i.e., the corresponding node of the fixer is not included in the tossing graph.
- With ***Algorithm 1***, we get the subgraph of the original tossing graph. If the corresponding node of the bug fixer is not include or not reachable in the subgraph.
- The initial assigner is not included in the tossing graph

In above cases our approach produces the failure rate (*FR*). Similar to *MLTP*, FR also depends on the similarity threshold. The high threshold produces high *FR*. The high failure rate may affect the utility of our approach as a recommendation tool for bug assignment. In practice, by select suitable threshold our approach can obtain excellent results in *MLTP* and *FR*.

B. Experimental Setup

Eclipse and Mozilla are both well known software projects, in which all bug reports are submitted to the bug repository (i.e. Bugzilla). They have already been investigated (for instance in [3]10][11]). Also both projects have large bug repositories so as to provide enough data for evaluating our approach. We selected a subset of each repository to set up an experimental bug set in our study.

However, some of bug reports are invalid for the evaluation. For example, some bugs are resolved as duplicated ones or are never resolved. Thus, before applying our approach to these reports we must eliminate these useless reports.

C. Evaluation on Eclipse

We analyzed the first 115,058 bug reports from Eclipse (bug id from 5,001 to 120,059). We excluded Eclipse bug

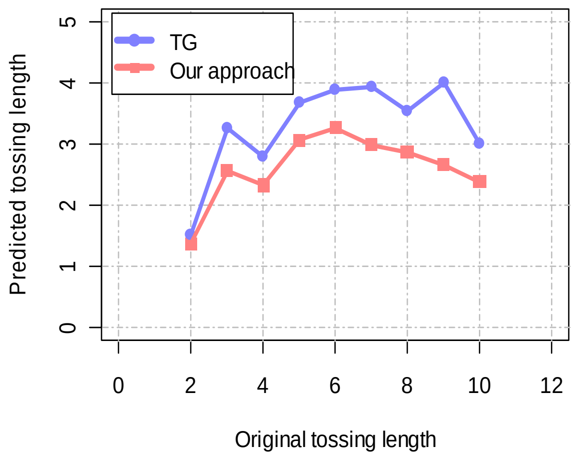


Figure 3. The tossing length: Original vs. Predicted in Eclipse, where TG denotes the approach that is only based on the tossing graph

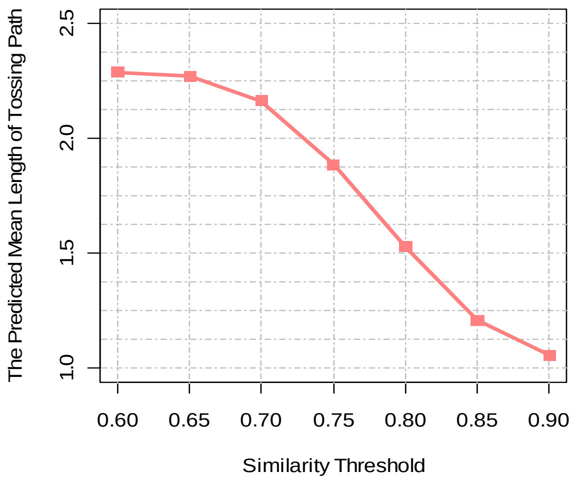


Figure 4. The predicted *Mean Length of Tossing Path* using different similarity thresholds in Eclipse

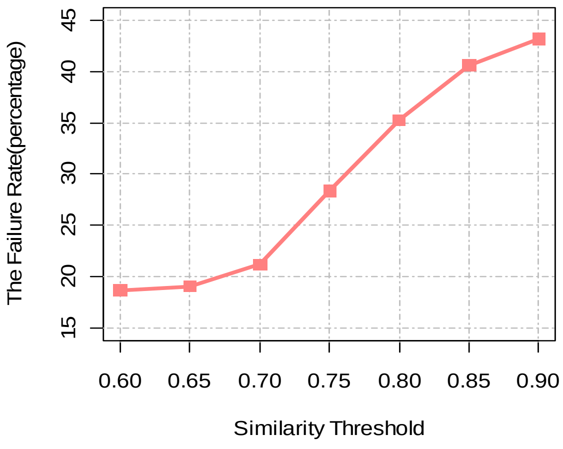


Figure 5. The failure rate using different similarity thresholds in Eclipse

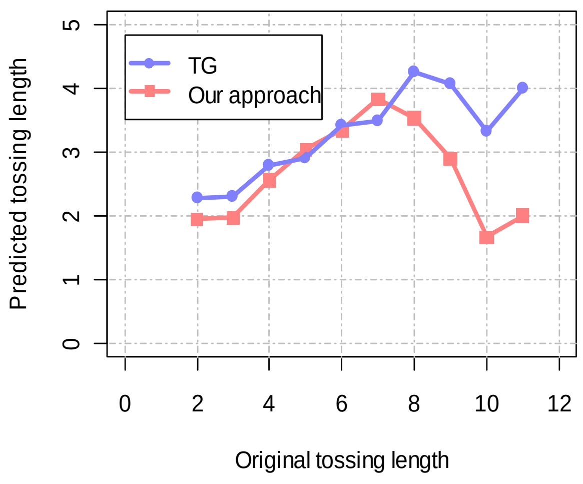


Figure 6. The tossing length: Original vs. Predicted in Mozilla, where TG denotes the approach that is only based on the tossing graph

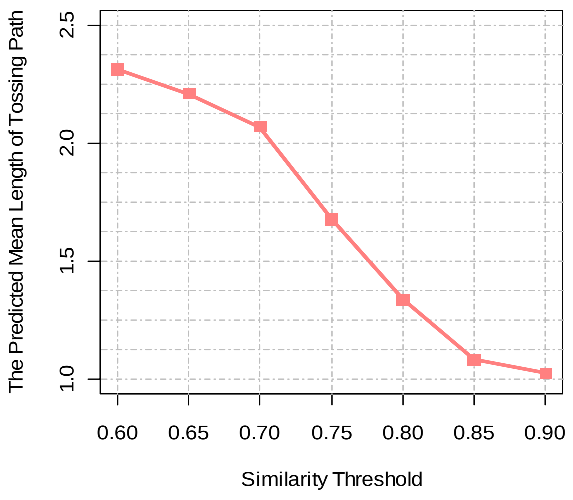


Figure 7. The predicted *Mean Length of Tossing Path* using different similarity thresholds in Mozilla

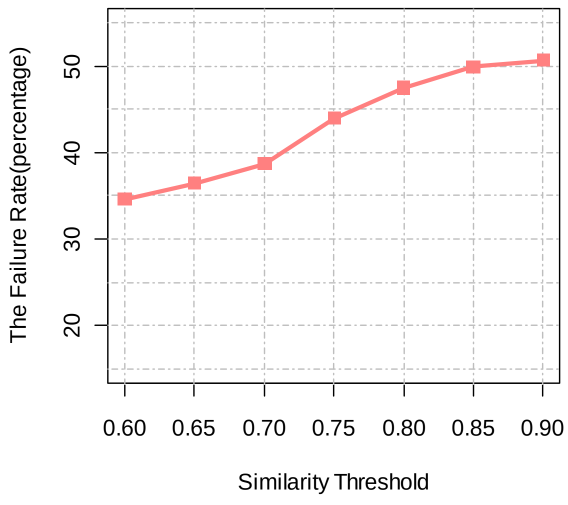


Figure 8. The failure rate using different similarity thresholds in Mozilla

reports from bug id 1 to 5,000 because these bug reports have been considered as suspicious bug reports [3]. Furthermore, as our approach is used to assign bug reports to bug fixers, duplicated and invalid bug reports must be eliminated because these bug reports never need to be fixed. The data set is divided into training set and testing set. The first 98433 Eclipse bugs are used as a training set, and the remaining bugs are used as a testing set.

In our approach only one parameter (similarity threshold T) need to be set (See Algorithm 1). By experimental method, we find T=0.65 produce the best result. Fig. 3 shows the comparison between the original tossing length and the predicted tossing length in Eclipse. We can see that our approach can significantly reduce the bug tossing length by up to 76.25%. For example, the original tossing length 10 is reduced to less than 3 on average. Moreover, Fig. 3 also displays that our approach based on the tossing graph and bug similarities is superior to the method that only uses the tossing graph.

By varying the similarity threshold T, different *MLTP* (See Definition 5.1) can be obtained. As shown in Fig. 4, the *MLTP* is decreasing with the increasing of the similarity threshold. However, Fig. 5 shows that the increasing of the similarity threshold results in the high search failure rate. The search failure rate mainly is because the target node (the bug fixer) is filtered out when the similarity threshold is too high. Nevertheless, the predicted *MLTP* is very short in case of keeping the lowest failure rate.

D. *Evaluation on Mozilla*

In order to further evaluate our approach, We carry out another experiment with the bug repository of Mozilla project. We analyze the first 119,852 bug reports from Mozilla (bug id from 1 to 119,852). After filtering out invalid and duplicated bug reports the first 96542 bug reports are used as a training set, and the other bug reports are used as a testing set.

To compare two experimental results, the parameter is set the same value in the two experiments. Fig. 6 shows the comparison between the original tossing length and the predicted tossing length in Mozilla project. The results can confirm almost all the findings in our evaluation on Eclipse. From Fig. 6 We can observe that our approach can significantly reduce the bug tossing length by up to 84%. For example, the original tossing length 10 is reduced to less than 1.6 on average. Moreover, the figures also display that in most case our approach outperform the method that only uses the tossing graph. Comparing Fig. 3 with Fig. 6 we can see that our approach reduce bug tossing length more in Eclipse than in Mozilla.

Fig. 7 presents the predicted *MLTP* using different similarity thresholds in Mozilla, which is similar to that of Eclipse. However, Comparing Fig 5 with Fig. 8 we observe that the *FR* is more high in Mozilla than in Eclipse under the same similarity threshold.

VI. Threats to Validity

In some cases, several developers work together for fixing a bug. For example, developers discuss how to fix the bug together. However, in our approach we assume that fixing a bug is completed only by a developer.

Our approach is only evaluated on two open source projects. Maybe the two projects are not representative of closed-source projects.

VII. Conclusion and Future Work

In this paper, we present an approach to improving bug assignment with bug tossing graph and bug similarities. In experiments, our approach is applied to two open source projects. Experimental results show that our approach can significantly reduce the bug tossing length. Moreover, Compare with the approach that only uses the bug tossing graph our approach can reduce tossing length more and decrease the search failure rate.

Calculating bug similarities are the most important part of our approach. In future work, we will take advantage of more field information of bug reports to calculate bug similarities more exactly, such as the detailed description of bug reports. In addition, more experiments on large-scale open source projects will be performed.

Acknowledgment

This work is supported by the national high technology research and development program of China (No. 2007AA010302) and national natural science foundation of China (No. 90718018).

References

[1] Bugzilla. http://www.bugzilla.org/.

[2] J. Anvik, L. Hiew, and G. Murphy, “Who should fix this bug?” in Proceedings of the 28th international conference on Software engineering. ACM, 2006, p. 370.

[3] G. Jeong, S. Kim, and T. Zimmermann, “Improving bug triage with bug tossing graphs,” in Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering on European software engineering conference and foundations of software engineering symposium. ACM, 2009, pp. 111–120.

[4] P. Runeson, M. Alexandersson, and O. Nyholm, “Detection of duplicate defect reports using natural language processing,” in Proceedings of the 29th international conference on Software Engineering. IEEE Computer Society, 2007, pp. 499–510.

[5] L. Hiew, “Assisted detection of duplicate bug reports,” Master’s thesis,The University Of British Columbia, 2006.

[6] X. Wang, L. Zhang, T. Xie, J. Anvik, and J. Sun, “An approach to detecting duplicate bug reports using natural language and execution information,” in Proceedings of the 30th international conference on Software engineering. ACM New York, NY, USA, 2008, pp. 461–470.

[7] D. Cubranic, “Automatic bug triage using text categorization,” in SEKE 2004: Proceedings of the Sixteenth International Conference on Software Engineering & Knowledge Engineering. Citeseer, 2004, pp. 92–97.

[8] C. D. Manning, P. Raghavan, and H. Sch¨utze, An Introduction to Information Retrieval. Cambridge

[9] Y. Wang, L. Li, and D. Xu, “Pervasive QoS routing in next generation networks,” Computer Communications, vol. 31, no. 14, pp. 3485–3491, 2008. University Press, 2008.

[10] N. Bettenburg, S. Just, A. Schroter, C. Weiss, R. Premraj, and T. Zimmermann, “What makes a good bug report?” in Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering. ACM, 2008, pp. 308–318.

[11] N. Bettenburg, R. Premraj, T. Zimmermann, and S. Kim, “Duplicate bug reports considered harmful... really?” in Proceedings of the 24th IEEE International Conference on Software Maintenance, September 2008.

[12] Q. Shao, Y. Chen, S. Tao, X. Yan, and N. Anerousis, “Efficient ticket routing by resolution sequence mining,” in Proceeding of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining. ACM, 2008, pp. 605–613.

[13] R. Silva, J. Zhang, and J. G. Shanahan. Probabilistic workflow mining. In KDD ’05: Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining, pages 275–284, New York, NY, USA, 2005. ACM.

[14] W. van der Aalst, T. Weijters, and L. Maruster. Workflow mining: Discovering process models from event logs. IEEE Trans. on Knowl. and Data Eng., 16(9):1128–1142, 2004.

[15] R. Agrawal, D. Gunopulos, and F. Leymann, “Mining process models from workflow logs,” Advances in Database Technology in EDBT’98, pp. 467–483, 1998.

[16] A. Rozinat and W. van der Aalst, “Decision mining in ProM,” Business Process Management, pp. 420–425, 2006.

[17] H. Mannila and D. Rusakov, “Decomposition of event sequences into independent components,” in Proc. of the First SIAM Conference on Data Mining. Citeseer, 2001.

[18] J. Pei, J. Han, B. Mortazavi-Asl, H. Pinto, Q. Chen, U. Dayal, and M. Hsu, “PrefixSpan: Mining sequential patterns efficiently by prefix-projected pattern growth,” in icccn. Published by the IEEE Computer Society, 2001, p. 0215.

Liguo Chen was born in Hubei Province, China in 1980, received his B.S. and M.S. degrees from Wuhan University of Science and Technology and China University of Geosciences in 2004 and in 2008 respectively. Currently, he is a Ph.D. candidate at School of Computer Science and Engineering, Beihang University. His current research interests include software engineering, data mining and information retrieval.

Xiaobo Wang was born in Henan Province, China in 1982, received his B.S. and Ph.D. degrees from Zhengzhou University and Beihang University in 2004 and in 2010 respectively. Currently, he is an engineer in Baidu China Co., Ltd. His current research interests include software engineering, data mining and business search.

Chao Liu was born in Beijing, China in 1958, received his Ph.D. degree from Beihang University. He is a professor and doctoral supervisor. Currently, he is assistant dean of School of Computer Science and Engineering and director of Software Engineering Institute in Beihang University. His current research interests include software testing, software quality, object-oriented technique and test case generation.