Enhancing the Bug Tossing Graphs and Characteristics of Bug Similarities

[1]T.L.N.Varaprasad, [2]Rambabu M
[1]M.Tech, Asst.pofessor, [2]M.Tech, Asst.pofessor,
[1]MCET JNTUH, SIDDIPET District, Telangana, India, [2]KGRCET JNTUH, RR District, Telangana, India,

Abstract: A bug report is typically assigned to a single developer who is then responsible for fixing the bug. 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 trigger 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 triages, 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 triggers 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 trigger to examine all existing bug reports to assign developers for fixing these bugs. One of the common reasons for bug tossing is that bugs are sometimes assigned to developers by mistake.

When tossing a bug report, it is often unclear who is the correct person to fix the bug ("who should the bug go to next?"). Another common reason for bug tossing is to include developers with additional expertise in the discussion of the bug report. In any case, many tossing events generally slow down progress when fixing bugs because, as we found in our study, a bug tossing event takes an average of 50 days.

This paper introduces a tossing graph model which is based on the Markov property (Section III). The proposed model captures tossing probabilities between developers from the tossing history available in bug tracking systems. This graph model has two desirable qualities:

- construct the tossing graph by extracting the bug tossing sequences.

This graph model has two desirable qualities:

1. Discovers developer networks and team structures.

We showed the tossing graphs to the Mozilla and Eclipse developers and received positive feedback. They confirmed the graphs are useful for bug processing tasks.

2. Helps to better assign developers to bug reports.

In our experiments with the 450,000 bug reports from Mozilla and Eclipse, our model could reduce the amount of bug tossing substantially. In addition, our graphs increased automatic bug assignment accuracy by up to 23 percentage points. We believe the proposed tossing graph model provides useful and actionable information to improve bug report processes.

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]). 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 sources 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. Forexample, 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, [2] used text categorization approach, which applied Support Vector Machine (SVM) to recommend to the bug trigger a set of developers who may be appropriate bug resolvers

SAMPLE TOSSING PATHS

A → B → C →E
A → C →B
C → D →E
B → C → F →E

A → E(1)	A → B(1)
B → E(2)	C → B(1)
C → E(3)	D → E(1)
F → E(1)	

THE NUMBERS IN PARENTHESIS INDICATE THE OCCURRENCES OF EACH SINGLE STEP

Carbonic and Murphy [7] applied Naive Byes 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.

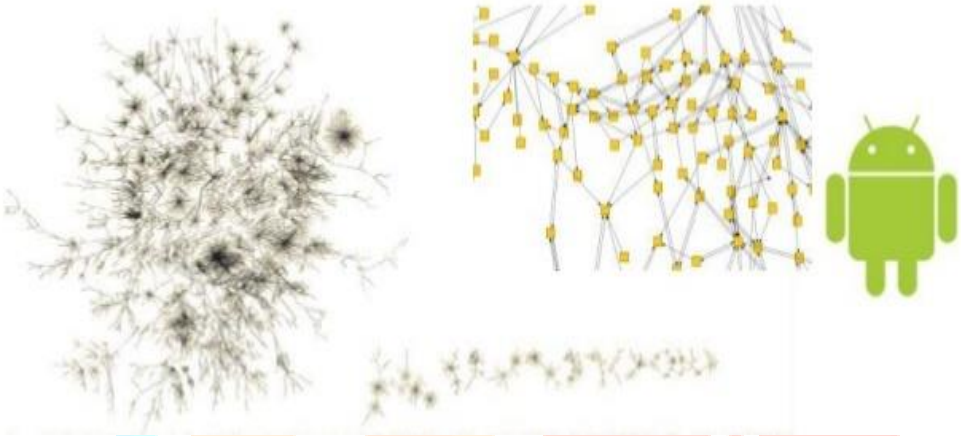


Figure 1: Average time spent per tossing step for all bugs.

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 .

III. BUG REPORT ANALYSIS

This section presents the bug assignment and tossing analysis results to provide understanding of bug report processes and properties.

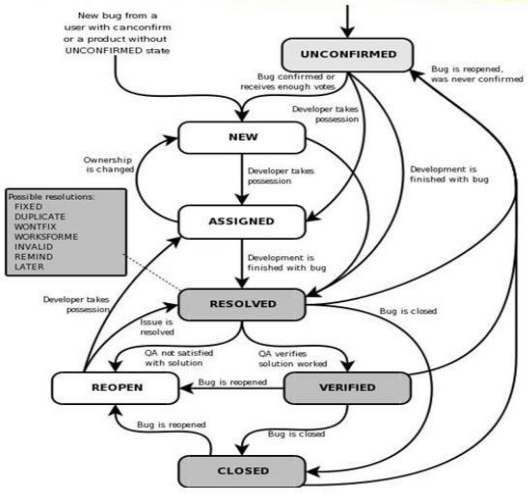
1 Subject Systems and Bug Reports:

We analyzed the first 145,000 bug reports from Eclipse (starting with bug id 5,001 and ending with id 150,000) and the first 300,000 bug reports from Mozilla. At the time of this writing, there are about 267,000 bugs in Eclipse and 482,000 bugs in Mozilla. Since our analysis is observing bug assignment and tossing processes, we only use bugs which are old enough to be assigned and processed.

2 Bug Assignments:

We noticed 426 Eclipse bugs are reported in one day. Similarly, Mozilla received 390 bug reports on Nov 22, 1999. Assuming that all bug processes are manual, how long would it take managers to perform an action on a new bug or to assign the bug to developers?

As shown in Figure 1(a) and Figure 2(a), the bug report process takes a long time. We observed the time spent between bug creation and the first action which is any action taken by a manager such as changing the status from UNCONFIRMED to NEW. Then we measured the time spent between bug creation and assignment. The first action on an Eclipse bug took 16.7 days on average. For Mozilla, it took 26.1 days on average. The bug assignment also takes a long time. After the first action, it takes 23.6 more days for Eclipse and 161.1 more days for Mozilla.



Next, we examine the time spent for the first action and assignment of only verified bugs as show in Figure 1. Some bug reports do not have enough information to be fixed, and they remain unassigned for a long time. Figure 1(b) and Figure 2(b) shows the time spent on verified bugs. It takes 5.2 days in Eclipse and 7.1 days in Mozilla for the first action on verified bugs. The bug assignment task then takes 19.3 days for Eclipse and 38.1 days for Mozilla.

3 Bug Tossing Paths:

In this section, we present statistics related to bug tossing. We first formally define the bug

Fig 2:Bug Assignning to the neareset neibours

tossing concepts. We describe bug tossing as follows. A tossing process starts with one developer, say d1, and moves from one developer to another until it reaches the fixer, df , the developer who fixed the bug. Each move is called a tossing step and a set of finite tossing steps T = {d1, d2, ..., df } of a bug is called a tossing path.

Among the 145,000 bug reports from Eclipse, 144,102 bugs have at least one action and 88,706 bugs are assigned. On average, a bug takes 16.7 days to have the first action and 23.6 days to be assigned after the first action.

A tossing interval denotes the time spent between two consecutive elements in a tossing path. A tossing path must have one fixer. The number of elements in a tossing path, |T| − 1 is called tossing length. If a tossing path, T = { d1 | d1 = df } has only one element, it has no or zero tossing. The bug was assigned to a developer, and she fixed the bug, i.e., no tossing event occurred.

We observed the tossing length of each bug. Figure 4 shows the Eclipse bug distribution based on their tossing length. For example, about 8,400 bugs (56%) have only one assigned developer, and no tossing event. However, 4,200 bugs (28%) have a single tossing event. Overall, about 44% of bugs have at least one tossing event.

IV. THE PROPOSED APPROACH

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. In this section, we describe our tossing graph model.

1. Bug Tossing Graphs:

Recall that we defined a tossing path as a set of developers, T = {d1, d2, ..., df } There are various ways to get tossing properties from tossing paths. Suppose we have a tossing path, A ! B ! C ! D. The bug is fixed by D, the fixer. A simple way to obtain tossing properties is to consider every single step in the path, A ! B, B ! C, and C ! D. This model is called the actual path model. Formally, for a given tossing path T, we define the actual path model as a set of steps, P = { di ! di+1 | di 2 T & di , df }. However, to quickly find or predict the bug fixer D from a given developer, we can use a model that decomposes the given paths to goal oriented steps, A ! D, B ! D, and C ! D. For a given tossing path T, we define the goal oriented model as a set of single steps, G = { di ! df | di 2 T & di , df }. This model encodes the relationship between intermediate developers and the fixer.

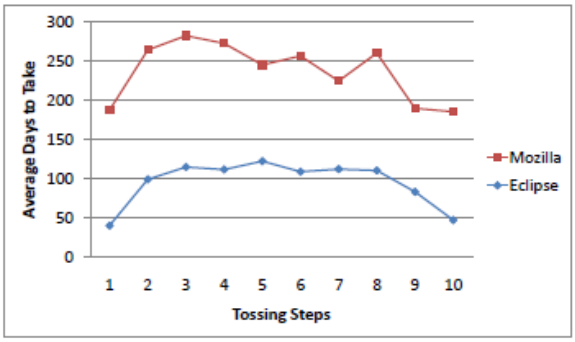


Figure 3: Average time spent per tossing step for all bugs.

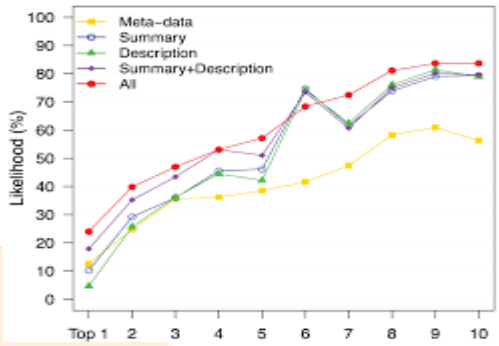


Figure 4: Average time spent per tossing step for verified bugs.

2. Bug Tossing BY Characteristics of Bug

Bug Tossing Path: A bug is assigned to the first developer 1 d , then it is reassigned to other developers until it reaches the fixer f d . Let the set of developers to whom the bug is assigned

T= d1→ d2 → ...→df

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 →Ian_Petersen, where Ian_Petersen is the bug fixer.

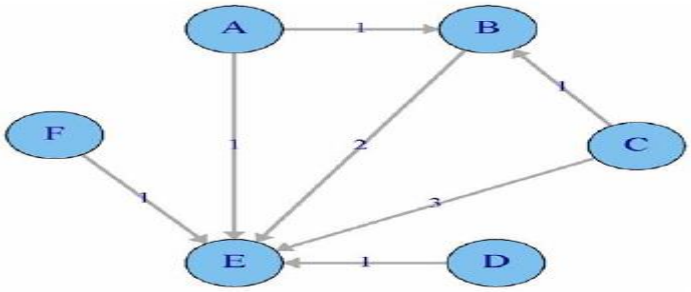


Figure 6. A tossing graph generated from the decomposed steps

Suppose there is a tossing path, A→B→C→D→ E, where E is the bug fixer.We can decomposes the path to goal oriented steps [3], A → E, B → E, C → E, and D→ 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→ E and one step F→ 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→ E in Tbl. III, and the edge weight corresponds to the occurrences of the step.

3. 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 (w1 ,w2, ...,wn), where n is the number of unique words (or terms) appearing in all the documents or queries. The ith element i w 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 1 q =(w11,w12, ...,w1n) and q2 = (w21,w22, ..., w2n), the cosine similarity of 1 q and 2 q is defined by Formula (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).

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 $A \rightarrow B \rightarrow C \rightarrow D$, 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 wefind 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.

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.

V. 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 largescale open source projects will be performed.

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, [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?" inProceedings 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.