社団法人 電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
TECHNICAL REPORT OF IEICE.

Reviewer Recommendation for Peer Review based on File Path Similarity: A Case Study of the Android Open Source Project

Patanamon THONGTANUNAM†, Ana ERIKA CAMARGO CRUZ†, Norihiro YOSHIDA†, and Hajimu IIDA†

† Graduate School of Information Science, Nara Institute of Science and Technology
8916-5 Takayama, Ikoma, Nara 630-0192 Japan
E-mail: †{patanamon-t,camago,yoshida}@is.naist.jp, †iida@itc.naist.jp

Abstract Peer Review is regarded as an important quality assurance mechanism in Open Source Software (OSS) development. Every change and its impact need to be assessed by developers to assure the quality of software. For effective code review, experience and skilled reviewers are required. However, in OSS peer review process, there is no explicit policy for selecting appropriate reviewers among available developers. The manual selection of those reviewers can be a costly and time-consuming task. To ease developers in OSS peer review process, we propose a reviewer recommendation algorithm determining similarity of file paths. As a case study, we recommended reviewers for the Android Open Source Project (AOSP). For this case study, our proposed algorithm obtained an accuracy of 75% while a previous approach obtained an accuracy of 30%.

Key words Peer code review; Reviewer recommendation; Reviewer assignment; Open Source Software;

1. Introduction

Peer Review is a process of source code inspection for assuring the quality of software. In Open Source Software (OSS) development, core developers need to assess whether incoming contributions negatively impact upon the overall quality or not [1]. Thus, a strict review process is important to ensure the quality of the software. To inspect source code efficiently, reviewers requires experiences and a deep understanding of the system code and its components. However, in the decentralized and non-personal landscape of OSS, there is no explicit policy for assigning reviewers [2]. Manually finding an appropriate reviewer among available OSS developers can be a labored and time-consuming task. Thus, automatization of this process is desirable to reduce human effort as well as the cost of software development and maintenance.

Several approaches [3]〜[6] have been proposed for recommending developers for bug fixing process. However, their recommendations are based on textual description of bug reports. These approaches cannot be applied to recommend reviewers since the textual information of peer review is usually short. Recently, Balachandran [7] has proposed an assisted tool called Review Bot for peer review process in industrial development (VMware projects). This tool can recommend reviewers effectively based on the modification history. However, OSS projects may lack such kind of history.

In this paper, we propose a reviewer recommendation algorithm based on the assumption that *reviewers having similar knowledge of system would have reviewed similar files and these files usually are in the same or near directories.* To find appropriate reviewers for an incoming contribution (a software change), our algorithm determines the similarity of its file path with previously reviewed files. Additionally, we applies the time prioritization to prioritize reviewers who recently reviewed those similar file paths. For our evaluation, we recommended reviewers for 5,126 reviews of Android Open Source Projects (AOSP). We measured the accuracy of our algorithm against the Review Bot's Algorithm. Furthermore, we addressed the following research questions.

- **RQ1:** How accurate are the reviewer recommendation algorithms for AOSP?
- **RQ2:** Does the time prioritization increase the accuracy?

From our results, our algorithm can recommend reviewers for AOSP with an accuracy of 75% which was more accurate than the Review Bot's algorithm (30%). Moreover, we found that our algorithm can work effectively without time prioritization.

2. Background and Related Work

2.1 Peer Review Process of AOSP

AOSP uses a web-based code review tool known as Ger-

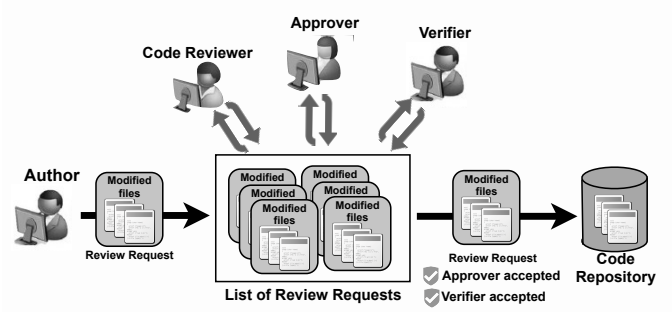

Figure 1 Simplified Version of the Code Review Process of Gerrit

rit(1). Figure 1 shows the code review process of Gerrit. A review begins with an author creating changes and submitting a set of those modified files as a review request to the system. Then, this request will be added to a list of review requests awaiting for reviewers. An author can request other developers to review her review request. However, those developers also can decide, whether to accept the request, based on their experience and interests [2].

In Gerrit, there are three kind of reviewers: Code reviewer, Approver and Verifier. Code reviewers give some comments and opinions about the changes. Approvers determine the quality and impact of the changes. Verifiers are responsible for integration testing of the change whether it conflicts with the project or not. According to AOSP's project roles(2), code reviewers can be any contributors in AOSP. Verifiers and approvers are experienced developers chosen by project leaders (employees of Google). Moreover, Gerrit provides an automatic tool named Deckard Autoverifier for integration testing. Verifiers usually test the changes when this tool cannot. The changes can be merged to project when their review request was accepted by at least one approver and one verifier. Thus, the most important reviewers in this peer review process are approvers and verifiers

According to this review process, requesting appropriate reviewers would make the review process faster and more effective. However, there is no explicit policy for assigning reviewers. Thus, an automatic reviewer recommendation tool would ease an author to find and request appropriate reviewers for her review request.

2.2 Review Bot

Review Bot is an extension of code review tool named Review Board(3). This system was proposed by Balachandran [7] to facilitate developers of VMware in peer review process. The reviewer recommendation algorithm of Review Bot is based on *line change history*. The line change history

(1) : https://android-review.googlesource.com/
(2) : http://source.android.com/source/roles.html
(3) : http://www.reviewboard.org/

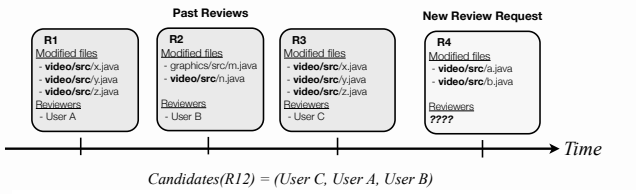

Figure 2 An example of ranking reviewers based on file path similarity

is a list of past reviews that affect to the changed lines of the new review. Furthermore, this algorithm prioritize the recent reviews by introducing a time prioritization factor (δ) which successfully increased the accuracy of recommendation.

This algorithm seems to be suitable for projects with a long history of changed lines, in other words, the source codes were changed frequently. However, AOSP may lack such kind of history since peer review process has been recently adopted.

3. Reviewer Recommendation Method

Usually, the directory structure of large system loosely mirrors the system architecture [8] and files with similar functions are stored in the same or near directories [9]. Thus, developers having knowledge of similar functions would understand those files with similar directory paths. Based on this assumption, we aimed to recommend reviewers for a given review request by selecting candidates from developers who had reviewed files with similar directory paths in the past. Furthermore, we would also consider time prioritization in our approach same as the Review's Bot algorithm.

For example, a review R4 in Figure 2 is a new review request containing two modified files. To find reviewer candidates, the path of modified files in the past reviews (R1, R2, R3) will be considered. As shown in the figure, all modified files of R1 and R3 are in the same directory as R4. Thus, reviewers of R1 and R3 (User A and User C) would probably be more experienced in the files of R4 than reviewer of R2 (User B). When recent reviews are prioritized, user C should have the highest rank. Therefore, the rank of reviewer candidates of R4 would be User C, User A, and User B.

3.1 Algorithm

Our reviewer recommendation algorithm is described in Algorithm 1. This algorithm takes two inputs: a new review request (R_n) and the number of top k candidates. In line 2, a list of past reviews of R_n is retrieved. The past reviews of this list are reviews created and closed (i.e. accepted or rejected) before the creation date of R_n. This list is sorted based on creation date of the past reviews in descending order. In line 3, the m value is defined to indicate the sequence of past review. The past review is the most re-

Algorithm 1 $RecommendReviewers(R_n, k)$

```
1: candidates ← list()
2: pastReviewList ← getPastReviews(R_n)
3: m ← 0
4: for Review R_p : pastReviewList do
5:     score ← FPS(R_n, R_p, m)
6:     for Reviewer r : getReviewers(R_p) do
7:         candidates[r] ← candidates[r] + score
8:     end for
9:     m ← m + 1
10: end for
11: candidates.sort()
12: return candidates[0 : k]
```

cent review when $m = 0$. This value will be used in file path similarity (FPS) function. In lines 4-10, the FPS function is iterated for every past review. In lines 6-8, for each past review, its reviewers are retrieved and assigned a FPS score. Then, these reviewers will be candidates for R_n. In line 11, the list of candidates is sorted based on candidates' scores in descending order. Line 12 returns the top k candidates with the highest score.

3.2 File Path Similarity

The FPS function calculates the score of a new review (R_n) and a past review (R_p) according to Equation 1.

$$FPS(R_n, R_p, m) = \frac{\sum_{\substack{f_n \in Files(R_n),\\ f_p \in Files(R_p)}} Similarity(f_n, f_p)}{|Files(R_n)| \times |Files(R_p)|} \times \delta^m \quad (1)$$

The $Files$ function returns a set of file paths of the input review. The $Similarity$ function measures the similarity between two file paths. The δ is a time prioritization factor ranging $(0, 1]$. When $\delta = 1$, the prioritization will not be considered.

The $Similarity$ function calculates the similarity between paths of file f_n and file f_p using Equation 2.

$$Similarity(f_n, f_p) = \frac{\#common\ path\ between\ f_n\ and\ f_p}{max(PathLength(f_n), PathLength(f_p))} \quad (2)$$

For *the number of common path between* f_n *and* f_p, we count the number of directories and file name that are common in both file paths. For example, the common path between ‘‘`src/camera/video/a.java`’’ and ‘‘`src/camera/photo/a.java`’’ is ‘‘`src/camera`’’. Thus, the number of common path is 2. For the $PathLength$ function, we count the total number of directories and file name of that file. Continuing the previous example, the path lengths of ‘‘`src/camera/video/a.java`’’ and ‘‘`src/camera/photo/a.java`’’ are 4. Therefore, the similarity between these file paths is $\frac{2}{max(4,4)} = 0.5$.

4. Experiments

To evaluate our approach, we iterated the proposed algorithm for every review in chronological and measured the accuracy of recommendation. We performed two experiments corresponding to our research questions as follows:

- **RQ1:** How accurate are the reviewer recommendation algorithms for AOSP?

We measured the accuracy of recommendation of our algorithm against the Review Bot's algorithm. For both algorithms, we recommended the top 5 candidates with time prioritization factor, $\delta = 0.8$.

- **RQ2:** Does the time prioritization increase the accuracy?

We addressed this research question because previous research successfully increased the accuracy of recommendation using time prioritization. Thus, we also determined the impact of time prioritization in our algorithm. For this purpose, we compared the accuracy of our algorithm with three different time prioritization factors. In this experiment, we recommended the top 5 candidates with time prioritization factors, $\delta = 0.6$, 0.8, and 1(No Prioritzation).

In this study, we focused only on approvers of AOSP. This is because tasks of verifiers can be done by automatic system; and code reviewers are not required to accept a review request. Authors also were not considered since they may not have permission of acceptance. Thus, we recommended candidates for a review request by selecting only approvers who had reviewed (i.e. accepted or rejected) a review in the past.

4.1 Dataset

In this study, we used the review history of AOSP provided by Hamasaki et. al [10]. We selected the reviews according to the following criteria:

- Already closed (Its status is Merged or Abandoned)
- Has at least one approver
- Has at least one file excluding the commit message file

The summary of dataset is shown as follows:

Table 1 Summary of Dataset

Study Period	# All reviews	# Selected Reviews
Jan 2009 - Jul 2012	11,631	5,126

4.2 Accuracy Measurement

To measure the accuracy of our algorithm, we used the method of the previous research. This is because the acceptance criterion of Gerrit is same as the tool used in previous research. The accuracy of top k candidates for N reviews is calculated using Equation 3. A recommendation is accurate

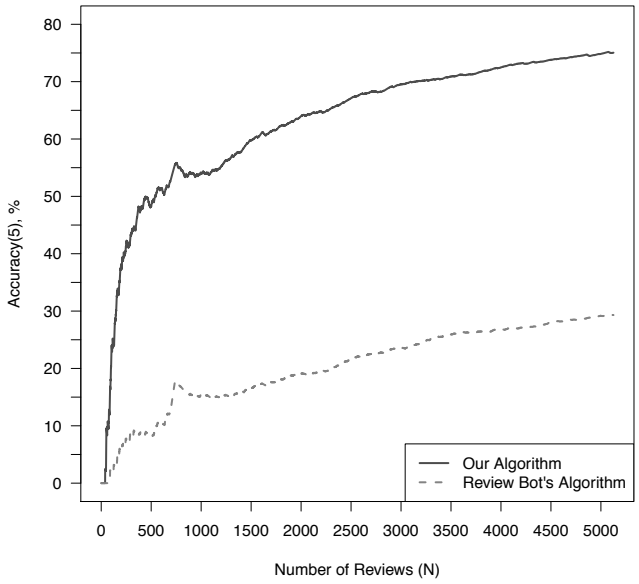

Figure 3 Accuracy of top 5 candidates recommendations of our algorithm and Review Bot' algorithm with $\delta = 0.8$

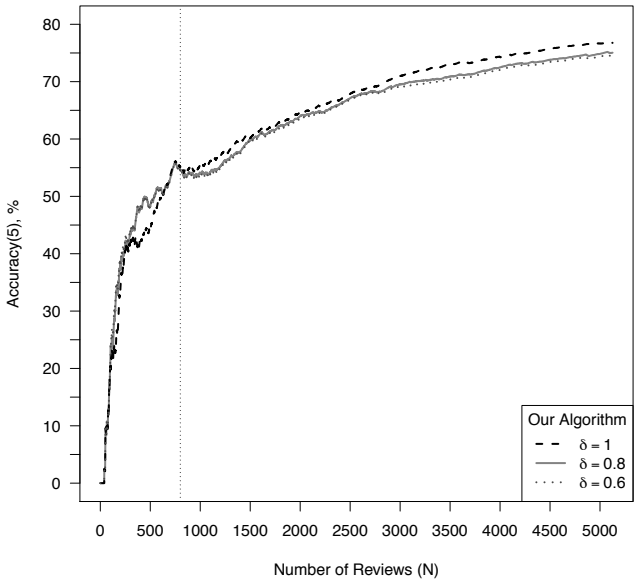

Figure 4 Accuracy of top 5 candidates recommendation of our algorithm with $\delta = 1, 0.6, 0.8$

when one of top k approver candidates actually reviewed a review request.

$$Accuracy(k) = \frac{\frac{\#Accurate\ recommendations}{of\ top\ k\ candidates}}{N} \times 100\% \quad (3)$$

5. Results

Figures 3 and 4 show the recommendation accuracies of the top 5 candidates corresponding to RQ1 and RQ2 respectively. The x-axis represents the number of reviews (N) which were recommended. The y-axis represents the accuracy gained.

5.1 RQ1: How accurate are the reviewer recommendation algorithms for AOSP?

As the results are shown in Fig. 3, the recommendations of our algorithm and the Review Bot's algorithm were accurate 75.04 % and 29.3% respectively when using 5,000 reviews. Both algorithms were more accurate when the number of reviews was increasing. This is because both algorithms used the review history of past reviews to find candidates.

Moreover, the recommendation of our algorithm was more accurate than the Review Bot's Algorithm ranging 2% - 46.20% for every N reviews. The Review Bot's algorithm cannot recommend reviewers accurately because for each file in AOSP, the past reviews that affect to changed lines were very few. From our exploration of the AOSP dataset, 90% of all files were modified only once. On the other hand, our algorithm accurately recommended reviewers for AOSP. This is because, in the review history, most of approvers reviewed files being in the same or near directories.

We successfully recommended reviewers for AOSP. However, the accuracy of our algorithm was not constantly increasing over time. As shown in the figure, for $N = 0$ to 2,500 reviews, the accuracy increased from 0% to 65%. After that, for $N = 2{,}500$ to 5,000 reviews, this accuracy increased from 65% to 75% (only 10%). These suggest that other factors relate to the reviewer assignment process of AOSP.

From our results and discussions, we can answer RQ1 that *for AOSP, the reviewer recommendation of our algorithm was more accurate than the Review Bot's Algorithm.*

5.2 RQ2: Does the time prioritization increase the accuracy?

As shown in Fig. 4, the accuracies of both recommendations with time prioritization ($\delta = 0.6$ and 0.8) were nearly equal for any N reviews. For the recommendation without time prioritization ($\delta = 1$), at the beginning of the history (from $N = 0$ to 800 reviews), it was less accurate than the recommendation with time prioritization. This means actual approvers were those who recently reviewed. However, when the number of reviews in the history was increasing, the recommendation without prioritization was becoming the most accurate recommendation. This means actual approvers can be any approvers who had reviewed in the past. One possible reason for inaccuracy of time prioritization is the influx of reviews. The contributions with different files may be submitted simultaneously as has been found in bug fixing process [11]. Thus, the files of new review requests can be irrelevant with previous reviews. Thus, developers who recently reviewed possibly did not have knowledge to review these new review requests.

From the results, we can answer RQ2 that *based on file*

path similarity, the time prioritization cannot increase the accuracy of the recommendation for AOSP as the number of reviews in history increases.

6. Limitation and Future work

As we mentioned earlier, our algorithm should determine other factors to improve the accuracy of recommendation. Furthermore, the following limitations will be also considered as our future works.

- **Workload of Reviewers** We successfully recommended five appropriate reviewers for AOSP. However, at this stage, our algorithm did not consider workload of those reviewers. It is possible that they were potential reviewers and were assigned for a huge number of review requests. Thus, considering workload balancing would reduce tasks of these potential reviewers as well as the number of awaiting reviews. For this purpose, our algorithm will take the number of tasks of candidates into account. Then, candidates with fewer tasks would be a higher rank of recommendation.
- **Quality of Reviews** Our algorithm recommend developer candidates who are possibly able to review those review requests. However, we cannot say that these candidates can review efficiently. To find those expert reviewers, we need to determine the quality of their reviews. However, Gerrit does not provide such kind of information. Therefore, we will explore a method for measuring quality of reviews. Then, we can use this measurement to improve our algorithm.

7. Conclusion

Manually finding appropriate reviewers can be a labored and time-consuming task. Recommending reviewers by automization would speed up peer review process. In this paper, we proposed a new algorithm for reviewer recommendation based on similarity of file paths.

In this study, we applied our algorithm and the Review Bot's algorithm for AOSP. We selected and used 5,126 reviews of its review history. We recommended approvers for every review request by selecting developers who had reviewed in the past. We evaluated the accuracy of our algorithm against the Review Bot's algorithm. Our results have shown that our algorithm was accurate 75% while the Review Bot's algorithm was accurate only 30%. Moreover, we determined the impact of using time prioritization in our algorithm. The results have shown that our algorithm can recommend reviewers accurately without time prioritization. Furthermore, we found that most of files in AOSP were modified only once, and approvers of AOSP usually review files being in the same or near directories. Both reasons for which our algorithm perform better for AOSP than the Review Bot's algorithm.

Our future works will be focused on balancing of reviewers' workload and assessing the quality of reviews. Our recommendation algorithm and these future works will improve peer review process to perform more efficiently.

ACKNOWLEDGMENTS

We gratefully acknowledge Dr. Raula Gaikovina Kula from Osaka University and Xin Yang from Nara Institute of Science and Technology (NAIST) for their valuable suggestions and discussions.

References

[1] O. Baysal, O. Kononenko, R. Holmes, and M.W. Godfrey, "The Secret Life of Patches: A Firefox Case Study," Proceedings of the 2012 19th Working Conference on Reverse Engineering - WCRE '12, pp.447–455, Oct. 2012.

[2] P.C. Rigby and M.-A. Storey, "Understanding broadcast based peer review on open source software projects," Proceeding of the 33rd international conference on Software engineering - ICSE '11, p.541, 2011.

[3] J. Anvik, L. Hiew, and G.C. Murphy, "Who should fix this bug?," Proceeding of the 28th international conference on Software engineering - ICSE '06, pp.361–370, 2006.

[4] G. Jeong, S. Kim, and T. Zimmermann, "Improving bug triage with bug tossing graphs," 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 - E, pp.111–120, 2009.

[5] R. Shokripour, J. Anvik, Z.M. Kasirun, and S. Zamani, "Why So Complicated ? Simple Term Filtering and Weighting for Location-Based Bug Report Assignment Recommendation," Proceedings of the 10th Working Conference on Mining Software Repositories - MSR'13, pp.2–11, 2013.

[6] H. Kagdi and D. Poshyvanyk, "Who Can Help Me with this Change Request?," The 17th IEEE International Conference on Program Comprehension - ICPC 2009, pp.273–277, 2009.

[7] V. Balachandran, "Reducing Human Effort and Improving Quality in Peer Code Reviews using Automatic Static Analysis and Reviewer Recommendation," Proceedings of the 35th International Conference on Software Engineering - ICSE'13, pp.931–940, 2013.

[8] E.T. Barr, C. Bird, P.C. Rigby, A. Hindle, D.M. German, and P. Devanbu, "Cohesive and Isolated Development with Branches," Proceedings of the 15th international conference on Fundamental Approaches to Software Engineering - FASE'12, pp.316–331, 2012.

[9] I.T. Bowman, R.C. Holt, and N.V. Brewster, "Linux as a case study: Its extracted software architecture," Proceedings of the 1999 International Conference on Software Engineering - ICSE'99, pp.555–563, 1999.

[10] K. Hamasaki, R.G. Kula, N. Yoshida, C.C.A. Erika, K. Fujiwara, and H. Iida, "Who does what during a Code Review ? An extraction of an OSS Peer Review Repository," Proceedings of the 10th Working Conference on Mining Software Repositories - MSR'13, pp.49–52, 2013.

[11] K. Herzig and A. Zeller, "The Impact of Tangled Code Changes," Proceedings of the 10th Working Conference on Mining Software Repositories - MSR'13, pp.121–130, 2013.