2013 20th Asia-Pacific Software Engineering Conference

Patch Reviewer Recommendation in OSS Projects

John Boaz Lee
Information Systems and Computer Science Dept.
Ateneo de Manila University
Quezon City, Philippines
jlee@ateneo.edu

Akinori Ihara, Akito Monden, and Ken-ichi Matsumoto
Graduate School of Information Science
Nara Institute of Science and Technology
Ikoma City, Japan
{*akinori-i, akito-m, matumoto*}*@is.naist.jp*

***Abstract*—In an Open Source Software (OSS) project, many developers contribute by submitting source code patches. To maintain the quality of the code, certain experienced developers review each patch before it can be applied or committed. Ideally, within a short amount of time after its submission, a patch is assigned to a reviewer and reviewed. In the real world, however, many large and active OSS projects evolve at a rapid pace and the core developers can get swamped with a large number of patches to review. Furthermore, since these core members may not always be available or may choose to leave the project, it can be challenging, at times, to find a good reviewer for a patch. In this paper, we propose a graph-based method to automatically recommend the most suitable reviewers for a patch. To evaluate our method, we conducted experiments to predict the developers who will apply new changes to the source code in the Eclipse project. Our method achieved an average recall of 0.84 for top-5 predictions and a recall of 0.94 for top-10 predictions.**

***Keywords*-patch reviewer recommendation; CVS; random walk; mining software repositories**

I. Introduction

In OSS projects, a core group of developers are granted certain privileges including the right to modify the source code of the software. These individuals, also called "committers", are usually responsible for the review of software updates (patches) that other developers submit for inclusion in the software [21] since they are seen as trusted members of the community.

In large projects, committers usually have to review many submitted patches [8]. Moreover, each review often takes a non-trivial amount of time as the committer has to carefully check the submitted code for bugs and compliance to coding standards. For instance, in the PostgreSQL project , the median time for a committer to review a patch is 508 minutes which is roughly equivalent to a whole day's work [9]. Because of this, "reviewers are usually overwhelmed with the number of patches they have to review" [17] and at times many patches in OSS development are even left unreviewed [5], [21].

When a patch is submitted for review, it is not always immediately clear to whom to assign it to for review. This may be because the best candidate reviewer (the developer with commit rights) is busy or may no longer be part of the project [6].

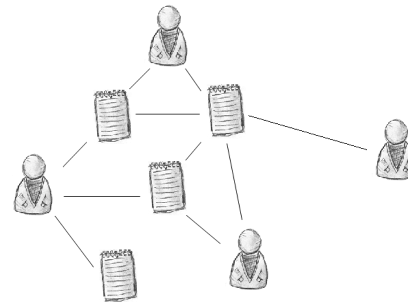

Figure 1. A graph can be used to represent a software project, committers are linked to source codes they have worked on and similar source codes can also be connected to one another.

To help take some load off the developers' backs, many recent work have suggested methods to automate certain parts of the software development process. Several papers discuss methods that can automatically triage incoming bug reports to developers with the skills to fix the bug [3], [11], [15]. A text-based approach to identify the expertise of developers for bug triaging is proposed in [15] while a graph model is described in [11]. Similar work has also been done to automatically detect the code that contains the bug described in reports [25]. Many works have thoroughly examined the patch review process. [17] notes that although the reviewers are primarily responsible for patch reviews, the process is only successful if there is adequate help from the community. The mechanisms that defined an effective and efficient peer review in certain OSS projects were studied in [20] while [1], [4] studied the different types of commits.

Kagdi and Poshyvanyk introduced a method to recommend a ranked list of developers to assist in performing software changes [12]. It is not hard to imagine the use of their method to recommend reviewers for a patch. Our method, however, is different from [12] since we use a graph-based approach in modeling commit history. We believe a graph, containing developer and source code nodes as well as their relations, can capture many important interactions in a software project. Our contributions are as follows.

- A graph-based method for ranking/recommending po-

1530-1362/13 $31.00 © 2013 IEEE
DOI 10.1109/APSEC.2013.103

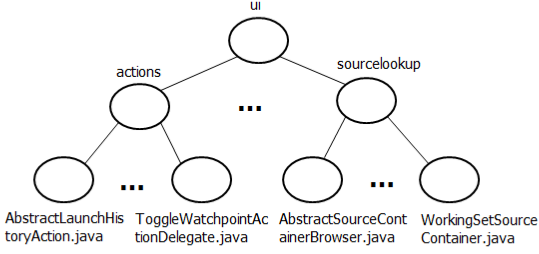

Figure 2. A namespace-based tree showing some packages and files in the Eclipse project. The subtree is rooted at the package *org.eclipse.debug.ui*.

tential patch reviewers is proposed and tested.

- Experiments on data taken from a real world OSS project demonstrate the efficiency of the proposed method.

The rest of the paper is structured as follows. In the next section we talk about the proposed method. In section 3, we introduce the dataset, describe the experiments and elaborate on the experimental results. We then conclude the paper with some suggestions for future work.

II. Proposed Method

A. Network Based on Commit History

A Content Versions System (CVS) is a system that can keep a record of all the changes made to a set of files and is used by developers to collaboratively maintain the source code for the project. Once a patch has been reviewed and approved, the reviewer can "commit" the changes to the corresponding source code. Our idea is to "profile" the different committers by analyzing their commit history on the different source codes in the project. We can then create a system to recommend candidate reviewers for a software patch by checking committer profiles.

In general, a software project can be modeled as a graph $G = \langle V, E\rangle$ comprised of a vertex set V with n types of nodes and an edge set E that represents a maximum of $m = n^2$ relations. In other words, $V = \cup_{i=1}^{n} V_i$, and $E = \cup_{i=1}^{m} E_i$.

In this particular paper, we create an undirected network with $V = V_c \cup V_s$ where V_c is the set of committers and V_s is the set of source codes. Furthermore, $E = E_c \cup E_r$, where $E_c = \{\langle i, j\rangle \mid i \in V_c$ and $j \in V_s$ or vice versa$\}$ is the set of edges between committers and code denoting an i "commits to" j relationship. $E_r = \{\langle i, j\rangle \mid i, j \in V_s\}$ is the set of edges connecting two source code entities capturing the relationship i "is related to" j. An illustration is shown in Fig. 1.

The first kind of relationship, E_c, is actually quite straightforward and can be observed by analyzing the CVS commit history of a software project. E_r, however, is a little ambiguous as many different measures can be proposed to capture similarity between source code files.

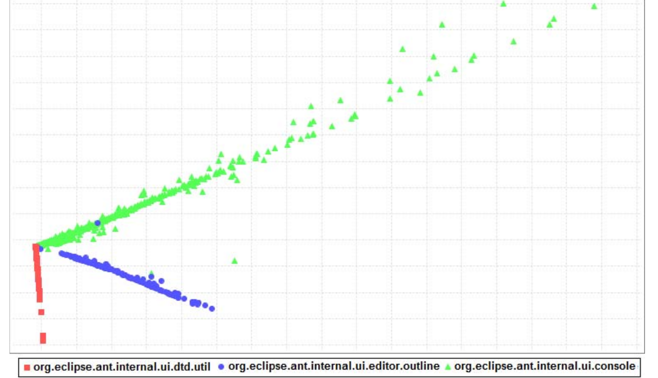

Figure 3. The similarities of source codes in three related modules based on commits by different developers.

Fig. 2 shows an example of a tree that can be created based on the namespaces of files in a project. In our work, we use a simple heuristic to measure relatedness, the similarity of two source code files is simply the length of the shortest path between them in the tree divided by two. For instance, *AbstractLaunchHistoryAction.java* and *ToggleWatchpointActionDelegate.java* have similarity of 1 while the the latter's similarity with *WorkingSetSourceContainer.java* is 2. An edge can then be drawn between two source code entities if their similarity is below some threshold. It is not hard to imagine the use of measures based on LSI [24], LDA [10], or common committers too and an interesting area for future work would be to identify the efficiency of various source code similarity measures on different datasets.

B. Committer Recommendation in Commit History Network

Given, as input, the commit history graph $G = \langle V, E\rangle$, a source code $s \in V_s$, and a positive integer $k \in \mathbb{N}$, our task is to return a sequence of committers $(c_1, ..., c_k)$, where $c_i \in V_c$ for $1 \leq i \leq k$, that are most qualified to review and commit a patch on the source code s.

To validate our method, we create G based on the CVS commit history of an OSS project for some time period t and predict the set of committers $(c_1, ..., c_k)$ for source code s where $\langle c_i, s\rangle \notin E$ but is expected to appear in some future time period t'. In other words, we validate the efficiency of our recommendation method by predicting future committers to a source code based on the current commit history graph.

We chose to predict future committers, because as mentioned above, committers are also often tasked with the job of reviewing new patches. Also, the act of committing changes to a file can be viewed as a sign of a committer's suitability to review patches for the said file.

C. Edge Weighting Method

We now describe two simple ways to add weights to the two kinds of edges described earlier.

1. Source-Developer Edge. Developers may have the right to commit to a large set of source codes but may spend majority of their time on a certain subset. This should be reflected by adding weights to the edges between source codes and committers. $w(c, s)$ is the function that calculates the weight of an edge between a committer node c and a source code node s. In this work, $w(c, s)$ is simply the aggregate number of lines changed by the committer c when committing patches to the source code c during the time window t. If $w(c, s) = 0$, then no edge exists between c and s; otherwise, an edge with weight $w(c, s)$ can be found between nodes c and s.

2. Source-Source Edge. Not all sources are equally related, some may be more related than others. To capture this, we define $w'(s, s')$ that measures the similarity of two source codes s and s'. In this work $w'(s, s') = \frac{l(s,s')}{2}$ where $l(s, s')$ is the length of the path from s to s' on the namespace-based tree in Fig. 2. Similarly, no edge exists between s and s' if $w'(s, s') = 0$; otherwise, edge $\langle s, s' \rangle$ has weight $w'(s, s')$.

At this point, we would like to talk briefly about the reason why we chose this simple weight function instead of using a text-based method like LSI [24]. In the experiments we conducted, we actually used LSI to weight edges between source codes but contrary to our intuition it actually decreased the accuracy of our method. This led us to believe that committers in our test dataset (Eclipse) are actually responsible for logical groups of code partitioned by namespace (or packages in Java).

To evaluate this hypothesis, we represented each source code as a vector v with the component $v_{s,i}$ containing the aggregate lines added by developer i to source s. We then used MDS [13] to reduce the dimensions to 2-d. Fig. 3 shows the result of MDS on the source codes in three related modules: **ant.internal.ui.dtd.util*, **.ant.internal.ui.editor.outline*, and **.ant.internal.ui.console*. It is interesting to note that even though these are all UI related modules, different sets of developers seem to be working on the source files in the three modules. We believe that this seems to indicate that the area of responsibility of an OSS project committer may be more module-based although further studies should be made to verify this.

D. Random Walk Based Algorithm for Reviewer Recommendation

To be able to run a random walk on a graph, the graph must first be turned into a Markov Chain [22]. To do this we create a stochastic matrix by normalizing the weights of all edges. Since our graph G is a connected, non-bipartite, and undirected graph, it can be shown that the markov chain over it is irreducible and aperiodic [2]. Thus, by the Perron-Frobenius theorem [7], it is clear that the random walk on G will converge to a stationary state which corresponds to the left eigenvector of the stochastic matrix.

The normalized edge weight $W(c, s)$ of the edge $\langle c, s \rangle$ from a committer c to a source code s is defined as follows.

$$W(c, s) = \frac{w(c, s)}{\sum_{s' \in N_{code}(c)} w(c, s')}$$

where $N_{code}(c)$ is the set of all source codes that are directly connected to c.

The normalized edge weight $W(s, \cdot)$ of an edge $\langle s, \cdot \rangle$ from a source code s to another node can be defined as follows.

$$W(s, s') = \begin{cases} \frac{\lambda w'(s,s')}{\sum_{\hat{s} \in N_{code}(s)} w'(s,\hat{s})} & \text{: if } |N_{code}(s)| > 0 \text{ and } |N_{dev}(s)| > 0; \\ \frac{w'(s,s')}{\sum_{\hat{s} \in N_{code}(s)} w'(s,\hat{s})} & \text{: if } |N_{code}(s)| > 0 \text{ and } |N_{dev}(s)| = 0; \\ 0 & \text{: otherwise.} \end{cases}$$

$$W(s, c) = \begin{cases} \frac{(1-\lambda) w(c,s)}{\sum_{c' \in N_{dev}(s)} w(c',s)} & \text{: if } |N_{dev}(s)| > 0 \text{ and } |N_{code}(s)| > 0; \\ \frac{w(c,s)}{\sum_{c' \in N_{dev}(s)} w(c',s)} & \text{: if } |N_{dev}(s)| > 0 \text{ and } |N_{code}(s)| = 0; \\ 0 & \text{: otherwise.} \end{cases}$$

where $N_{dev}(s)$ is the set of committers that are neighbors of source code s and $\lambda = [0, 1]$ is a parameter that is used to indicate how much priority is given to a certain kind of link.

To perform reviewer recommendation for a patch to be applied to a source file s, we use a random walk process similar to the process described in [14]. The process is started from a single query source code s and once the random walk has converged, the stationary random walk probabilities from s to the committer nodes in the network is considered the likelihood of a link occurring between s and the respective nodes in the future. The higher the random walk probability from s to a committer c is, the more suitable c is as a potential reviewer of a patch for s as c's activities relate it closely to s.

To calculate the random walk probabilities from a query node s^* to the rest of the nodes in the network, we iteratively apply the following process.

$$r_s^{\langle t \rangle} = (1-p) \sum_{s' \in N_{code}(s)} W(s', s) r_{s'}^{\langle t-1 \rangle} + (1-p) \sum_{c' \in N_{dev}(s)} W(c', s) r_{c'}^{\langle t-1 \rangle} + p r_s^{\langle 0 \rangle}$$

$$r_c^{\langle t \rangle} = (1-p) \sum_{s' \in N_{code}(c)} W(s', c) r_{s'}^{\langle t-1 \rangle}$$

where the vector components $r_s^{\langle t \rangle}$ and $r_c^{\langle t \rangle}$ hold the random walk probabilities, after the tth iteration, from s^* to source code node s and committer node c, respectively. Before the process is started, the vector $r^{\langle 0 \rangle}$ has all its components

initialized to zero except for $r_{s^*}^{\langle 0 \rangle}$ which is set to 1. In the succeeding time steps, the random walker then begins to explore the graph. The parameter p here is the restart probability and the higher its value the more likely the random walker will restart at the query node which, in turn, confines the random walk to a more local neighborhood (relative to s^*).

Below is a short summary of the proposed method.

Algorithm 1 Random Walk Algorithm

1. Input: The network $G = \langle V, E \rangle$ based on CVS commit history, a source code $s^* \in V_s$, and parameters λ and p.
2. Assign weights to the edges in G and normalize values.
3. Create the random walk vector r and set $r_{s^*}^{\langle 0 \rangle} = 1$ and $r_{s'}^{\langle 0 \rangle} = 0$. Set $r_c^{\langle 0 \rangle} = 0$ for all committer nodes $c \in V_c$. Iterate to update r until values have converged.
5. Select all committers c where $c \notin N_{dev}(s^*)$ and order the selected nodes by decreasing $r_c^{\langle * \rangle}$, where $r_c^{\langle * \rangle}$ is the stationary random walk probability from s^* to c.
6. Output the sequence as the recommended set of reviewers for a patch in source code s^*.

E. Some Advantages of Graph Approach

To the best of our knowledge, this is the first work that explicitly tackles the problem of recommending reviewers for a software patch. While one can certainly apply the algorithms for assigning bug fixers to the problem we study here, we argue that a graph approach also has its advantages. Studies have shown that graph-based metrics can tell us important things about a piece of software [18].

If a patch consists of changes to a single source file, it is intuitive to identify the reviewer from among the past reviewers assigned to the source file. However, as is the case often, a patch consists of changes to multiple source files and often times no single reviewer has had the chance to review all the source files before.

In such a scenario, a graph approach is quite suitable. All we would have to do is replace the query node s^* with a set of query nodes S, and distribute the starting random walk probabilities among all $s_i \in S$, for $1 \leq i \leq |S|$, such that $\sum_{i=1}^{|S|} r_{s_i}^{\langle 0 \rangle} = 1$. Our approach would then still be able to identify the developers that are most suitable to review the patch given all the affected source files. Note that S does not necessarily have to contain a uniform distribution and can depend on the importance of the affected source files.

Meneely et al. [16] discovered that a developer network can be used as an estimate for developer collaboration. While developer collaboration is important in reviewer recommendation, an advantage of our graph approach is that the system can be quickly extended to include other types of nodes to allow for even richer analysis.

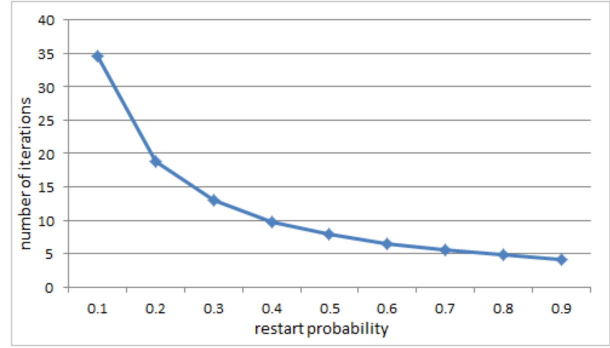

Figure 4. Average number of iterations for the random walk process to reach the stationary state.

III. Experiments and Results

A. Dataset

We gathered all commit logs for patches committed to the Eclipse project between June 30, 2005 and June 29, 2006. We chose this time-frame because it starts with a major release of Eclipse and covers the time period until the next major release. The total number of commits made to the project was around 26,800. We also downloaded the entire June 29, 2006 source code snapshot of the project. Over the one year period, 56 different committers made changes to 6,178 files found in the snapshot that we downloaded. The graph that we built had around 10,500 edges and 6,234 nodes. An edge was drawn between two source files if they were connected by a path of at most length 2 (meaning they belonged to the same package).

Our snapshot of the Eclipse project contained source files grouped into 608 packages, the largest package contained 203 source files. Each committer committed changes to files in 29 different packages on average. A total of 26,794 commits were made during the time period and an average of around 5 different committers committed patches to each source file which shows that there is a substantial amount of overlap in the activities of the developers.

B. Experimental Results

We use the proposed method to recommend the top-k candidate reviewers for all source codes in the data set and used precision and recall to measure the relevance of the recommendations. Prec@k $= \frac{1}{|S|} \sum_{s \in S} \frac{P_k(s)}{k}$ where S is the set of source codes and $P_k(s)$ is the set of committer nodes that the algorithm recommended correctly, $i.e.$ these committers did indeed commit changes to the source code s in some future time interval. Precision captures the number of correct recommendations out of all k recommendations. Recall, on the other hand, can be defined as rec@k $= \frac{1}{|S|} \sum_{s \in S} \frac{P_k(s)}{R(s)}$ where $R(s)$ is the entire set of committers that eventually committed to the source code s; this captures the number of correct recommendations over all possible

	Prec@5	Rec@5	Prec@10	Rec@10
Sep 2005 (1)	0.227/0.241	0.943	0.116/0.120	0.954
Oct 2005 (2)	0.211/0.228	0.932	0.110/0.114	0.963
Nov 2005 (1)	0.172/0.213	0.801	0.089/0.107	0.828
Dec 2005 (1)	0.194/0.228	0.768	0.106/0.114	0.837
Jan 2006 (5)	0.167/0.211	0.797	0.089/0.106	0.841
Feb 2006 (4)	0.167/0/219	0.756	0.104/0.109	0.951
Mar 2006 (4)	0.164/0.212	0.773	0.100/0.106	0.945
Apr 2006 (1)	0.197/0.212	0.911	0.103/0.106	0.960
May 2006 (1)	0.191/0.206	0.930	0.102/0.103	0.992
Jun 2006 (2)	0.195/0.216	0.900	0.107/0.108	0.991
Weighted Ave	**0.183/0.215**	**0.843**	**0.101/0.107**	**0.937**

Table I
PRECISION AND RECALL FOR RECOMMENDED COMMITTERS IN DIFFERENT MONTHS. NUMBER OF PREVIOUS MONTHS USED TO BUILD THE GRAPH IS IN PARENTHESES. MAXIMUM POSSIBLE PRECISION, WHICH IS THE PRECISION SCORE IN THE OPTIMAL CASE, IS ALSO SHOWN FOR COMPARISON.

correct recommendations. We use precision and recall in lieu of accuracy because, in this case, true negatives outnumber true positives substantially. An example of an inefficient method with high accuracy is one that predicts zero new committers for all source files since there is only a small number of new committers for each source file.

We attempted to predict all new committers to the source codes for each month from September 2005 to June 2006. We did not do the experiment for the months of July and August because there was insufficient prior data to build the graph. For each month, we built a graph based on commit history from the previous months and used the proposed method to predict committers to each of the source code in the current month. Predicted committers for a source code are those who have never committed to it in the commit history of the previous months which we use to build the graph.

Table 1 displays the precision and recall for the top-k recommendations for each month. Note that in most cases, we only use commit history from the last 1 or 2 months to build the graph for recommendation. This may seem counter intuitive at first as less information about past activities could affect our prediction of future activities. However, in the Eclipse dataset, we have found that using the most recent history actually improves the recommendation. This may be because it captures the current profile of the committers. This seems to show that the area of responbility of developers in the project tend to evolve over time.

It is interesting to note that the method already predicts around 84% of all committers correctly in the top-5 recommendations, this is further increased to 94% for top-10 recommendations.

As expected, precision is rather low since, on average, only one new committer changes each source file. For top-5 predictions, the average number of true positives is 1. Even though around 80% of all recommended developers are considered false positives since they did not edit the code during the test period, their activities are actually quite similar to real committers. In fact, 33% of all the false positives in September committed to the source code much later. On the other hand, true negatives account for 77% of all non-recommended reviewers which shows the strong negative predictive value of the model.

In lieu of precision, one good measure of efficiency in this case is search length which can be defined as the average rank of each successfully recommended committer. A good algorithm would have search length close to 1 (in this case) which means the true positives were ranked highly in the recommendation. Our method had a search length of 2.11 and 2.66 for $k = 5$ and $k = 10$, respectively. This shows that most true positives were ranked high in the system's recommendations.

In our experiments, we found the ideal value for the restart probability c to be between 0.7 and 0.95 which suggests that the local neighborhood is more important in predicting future committers. We also found the ideal value for λ to be from 0.7 to 0.9 which shows that in our case the links connecting source files are more important than the edges between developers and source files. We arrived at the optimal values by testing the proposed method on all possible combinations of c and λ from the range $(0.0, 1.0)$ using 0.5 intervals.

C. Random Walk Convergence

Fig. 4 shows the average number of iterations needed for the random walk to converge. It can be observed that an average of around 35 iterations is needed for the random walk with restart probability $c = 0.1$ to converge while this number drops to around just 5 iterations for higher values of c.

On our test machine which had a 2.67GHz Intel Core i5 processor and 4GB of RAM, an implementation of the proposed algorithm predicted the future committers for all 6,178 source files in less than 3 seconds on average. From this, we can intuit that the method should be able to scale to graphs with a few million nodes. On larger graphs, one can always use the method described in [23] to increase the speed of the random walk process.

IV. CONCLUSION AND FUTURE WORK

In this paper, we perform a preliminary study for the problem of recommending patches to reviewers in an OSS project. We use data from the Eclipse project and built a simple model to predict future committers to source codes in the project. We have found that the model can already correctly predict 84% of all committers given its top-5 recommendations.

In the future, we would like to improve the model further by studying the actual text content of each patch committed by developers to identify the "topics" that each committer

is interested in. This will allow us to identify the expertise of each committer and can be used not only for reviewer recommendation but also for bug triaging since we can link the topics found in a bug report to developers with matching experience. By studying the textual information, we can also assess the importance of each patch. For instance, a general patch applied to multiple source files should be deemed less important than another that contains an implementation of a complex algorithm committed to a single source file.

In the current graph, only committers are included, future studies should also include patch submitters as well as it would be useful too to recommend some of them as reviewers.

Furthermore, we would like to refine the method in order to predict the actual developer who committed each patch. In our current approach, we simply predict whether a developer will commit to a source file in the future or not. Also, at the moment, there is no way to determine whether a developer is committing code written personally or code submitted as a patch by somebody else; this is something we'd like to address in future work.

Another thing we plan to do is to design more complex measures that can better capture the similarity between nodes in a commit history graph. In particular, we would like to investigate process metrics as these stay robust even as the codebase evolves [19]. We also plan to test the model on more software systems, including proprietary projects. Finally, a possible next step is to indicate the probability of a reviewer committing a mistake when assigned to review a certain patch.

It should be noted that in this preliminary study the graph modeling an OSS project is quite simplistic. However, one can easily imagine a graph containing other entities like topics, language, and even additional relationships like developer-developer links. Even then, we believe the general approach that we have discussed here should still be suitable for graphs that describe more complex relationships in OSS projects.

Acknowledgment

J.B. Lee would like to thank NAIST for sponsoring his visit to the Software Engineering Lab. We thank Satoshi Uchigaki for help with data gathering. This research is conducted as part of Grant-in-Aid for Young Scientists (B) 25730045, and for Scientific Research (B) 23300009 by the Japan Society for the Promotion of Science (JSPS). It is also supported in part by the Ateneo de Manila University.

References

[1] A. Alali, H. Kagdi, and J.I. Maletic. What's a Typical Commit? A Characterization of Open Source Software Repositories. In *Proc. of ICPC'08*, pp. 182-191, 2008.

[2] N. Alon, I. Benjamini, E. Lubetzky, and S. Sodin. Non-backtracking random walks mix faster. *Commun. Contemp. Math.*, 9:585-603, 2007.

[3] J. Anvik, L. Hiew, and G.C. Murphy. Who should fix this bug? In *Proc. of ICSE'06*, pp. 361 - 370, 2006.

[4] O. Arafat and D. Riehle. The Commit Size Distribution of Open Source Software. In *Proc. of HICSS'09*, pp. 1-8, 2009.

[5] C. Bird, A. Gourley, and P.T. Devanbu. Detecting Patch Submission and Acceptance in OSS Projects. In *Proc. of MSR'07*, pp. 26-29, 2007.

[6] C. Bird, A. Gourley, P.T. Devanbu, A. Swaminathan, and G. Hsu. Open Borders? Immigration in Open Source Projects. In *Proc. of MSR'07*, pp. 6, 2007.

[7] P. Blanchard and D. Volchenkov. Random Walks and Diffusions on Graphs and Databases: An Introduction. Springer (2011).

[8] G. Canfora and L. Cerulo. Supporting change request assignment in open source development. In *Proc. of SAC'06*, pp. 301-310, 2010.

[9] S. Fujita, M. Ohira, A. Ihara, and K. Matsumoto. An Analysis of Committers Toward Improving the Patch Review Process in OSS Development. In *Proc. of ISSRE'10*, pp. 369-374, 2010.

[10] T. Hofmann. Probabilistic Latent Semantic Analysis. In *Proc. of SIGIR'99*, pp. 50 - 57, 1999.

[11] G. Jeong, S. Kim, and T. Zimmermann. Improving bug triage with bug tossing graphs. In *Proc. of ESEC/FSE'09*, pp. 111-120, 2009.

[12] H. Kagdi and D. Poshyvanyk. Who Can Help Me with this Change Request? In *Proc. of ICPC'09*, pp. 273-277, 2009.

[13] J.B. Kruskal. Multidimensional scaling by optimizing goodness of fit to a nonmetric hypothesis. *Psychometrika*, 29(1): 1-27, 1964.

[14] J.B. Lee and H. Adorna. Link Prediction in a Modified Heterogeneous Bibliographic Network. In *Proc. of ASONAM'12*, 2012.

[15] D. Matter, A. Kuhn, and O. Nierstrasz. Assigning Bug Reports using a Vocabulary-Based Expertise Model of Developers. In *Proc. of MSR'09*, pp. 131-140, 2009.

[16] A. Meneely, M. Corcoran, and L. Williams. Improving Developer Activity Metrics with Issue Tracking Annotations. In *Proc. of ICSE'10*, pp. 75-80, 2010.

[17] M. Nurolahzade, S.M. Nasehi, S.H. Khandkar, and S. Rawal. The Role of Patch Review in Software Evolution: An Analysis of the Mozilla Firefox. In *Proc. of IWPSE-Evol'09*, pp. 9-18, 2009.

[18] M. Pinzger, N. Nagappan, and B. Murphy. Can Developer-Module Networks Predict Failures? In *Proc. of FSE'08*, pp. 2-12, 2008.

[19] F. Rahman and P.T. Devanbu. How, and why, process metrics are better. In *Proc. of ICSE'13*, pp. 432-441, 2013.

[20] P.C. Rigby, D.M. German, and M.-A. Storey. Open source software peer review practices: a case study of the apache server. In *Proc. of ICSE'08*, pp. 541-550, 2008.

[21] P.C. Rigby and M.-A. Storey. Understanding Broadcast Based Peer Review on Open Source Software Projects. In *Proc. of ICSE'11*, pp. 541-550, 2011.

[22] F. Spitzer. *Principles of Random Walk*. Springer, 2001.

[23] H. Tong, C. Faloutsos, and J.-Y. Pan. Fast random walk with restart and its applications. In *Proc. of ICDM'06*, pp. 613-622, 2006.

[24] Q. Wang, J. Xu, H. Li, and N. Craswell. Regularized Latent Semantic Indexing. In *Proc. of SIGIR'11*, pp. 685-694, 2011.

[25] J. Zhou, H. Zhang, and D. Lo. Where should the bugs be fixed? More accurate information retrieval-based bug localization based on bug reports. In *Proc. of ICSE'12*, pp. 14-24, 2012.