0-1 Programming Model-Based Method for Planning Code Review using Bug Fix History

Hirohisa Aman
Center for Information Technology
Ehime University
Matsuyama, Japan 790–8577
Email: aman@ehime-u.ac.jp

***Abstract*—Code review is a powerful activity for software quality improvement, and is ideal to review all source files being developed. However, such an exhaustive review would be difficult because the available time and effort are in reality limited. Thus, practitioners prioritize the source files in terms of bug-proneness by using related data such as bug fix history, and review them in decreasing order of priority—such strategy in this paper is called the "conventional method." While the conventional method is straightforward, it focuses only on the bug-proneness and cannot consider the review cost properly, so the method does not produce a cost-effective review plan. To take into account both the bug-proneness and the review cost, this paper proposes a 0-1 programming model-based method for planning code review. The proposed method formulates a review plan as a 0-1 programming problem, and the solution is the recommendation list of source files to be reviewed. Moreover, the proposed method considers the type of file—if the file is newly-developed or not. Such difference in file type may affect on how to evaluate the bug-proneness and the review strategy: newly-developed files are notable but not appeared in the bug fix history. This paper conducts a case study using popular open source software, shows that the proposed method is up to 42% more effective than the conventional method in recommending buggy files as the review targets.**

***Keywords*-code review; cost-effectiveness; 0-1 programming model; bug fix history;**

I. Introduction

A successful software development requires early detection and fixing of software bugs. Code review is one of effective activities to find bugs [1], so it is better to perform code review as much as possible in the development. However, a code review is also a costly activity since it cannot be performed automatically; it is done by engineers. Thus, an exhaustive performance of code review for all source files would be almost impossible because of realistic limitations such as lack of time, insufficient manpower and so on.

To perform code review effectively, a prioritization of source files in terms of bug-proneness is a key technique. If the bug-proneness of files are given, engineers can review bug-prone files preferentially. In order to evaluate bug-proneness, various metrics, methods and models have been widely studied in the software engineering world. In many studies, software quality attributes are quantified by product metrics [2], [3], and they are used as inputs for mathematical models [4], [5]. Those models provide information about bug-proneness. Recently, approaches other than product metrics have been actively studied: an application of Spam mail filtering [6], applications of data mining methods to repositories (mining software repositories: MSR) [7] and so on. Especially, process metrics, used in MSR world (e.g., code churn [8], the number of bug fixings [9]), are promising factors for buggy file prediction in recent years [10], [11]. Thus, in this paper, we focus on the prioritization of source files by using process metrics related to bug fixings.

Once the prioritization of source files is done, each file has a worth to review. Engineers can start their code review in decreasing order of the worth. Such method is straightforward and easy-to-perform. However, the method may not be cost-effective because it considers only the bug-proneness and misses the cost required for the review. When our available time and effort for code review are limited, we will have to consider not only the review worth but also the review cost. To take into account both the worth and the cost, we propose another method that formulates the review plan as a 0-1 programming problem in this paper.

The key contribution of this paper is to propose a 0-1 programming model-based method for planning cost-effective code review (see Section III). The proposed method takes into account not only the review worth but also the review cost. Moreover, the method considers the type of file—whether the file is a newly-developed one or not. Such difference in file type affects on how to evaluate the bug-proneness and the review strategy: newly-developed files should be reviewed preferentially, but they may not appear in the bug fix history. The proposed method is designed to consider the difference in file type as well.

The remainder of this paper is organized as follows. Section II describes the code review based on the bug fix history. Section III proposes a 0-1 programming model-based method. Then, Section IV presents a case study using popular open source software and discusses the usefulness of proposed method. Section V describes our related work. Finally, Section VI provides our conclusion and future work.

II. Code Review based on Bug Fix History

The aim of code review in this paper is to detect as many bugs as possible by inspecting bug-prone source files. This section provides the basis of code review discussed in this paper: how to evaluate the bug-proneness of source files using the bug fix history, and the conventional strategy of code review based on the quantified bug-proneness.

A. Evaluation of Bug-proneness

There have been many studies of buggy file prediction as mentioned in Section I. For instance, the cyclomatic complexity [2] is a well-known metric to find a bug-prone program: a larger value of cyclomatic complexity means more bug-prone. However, such metrics computed through static code analysis may not be fit in selecting review targets because the static code analysis cannot reflect the code change history. For example, suppose we have a complex source file having a high value of cyclomatic complexity. If that file is newly-developed, we will have to review it preferentially. But if it had never been changed since the previous version, we would have little or no need to review the file even if it were still a complex program. Thus, it would be better to utilize the bug fix history rather than the static code analysis in selecting source files to be reviewed. In this paper, we will evaluate the bug-proneness by using process metrics based on the bug fix history. Since such process metrics seem to be promising in recent studies of buggy file prediction with MSR [8], [9], [10], [11], the use of those process metrics would be reasonable.

In this paper, let $b(s)$ be the bug-proneness of source file s, and evaluate $b(s)$ according to either the following Criterion 1 or 2, which are basic criteria based on process metrics; while there have been various metrics and models in MSR world [7], [8], [9], [10], [11], further studies using other metrics and models are our significant future work.

Criterion 1: Number of bug fixings.
It is empirically known that the number of bug fixings performed for a source file is related to the bug-proneness of that file. The evaluation of bug-proneness by the following equation is simple but effective:

$$b(s) = (\text{the number of bug fixings in } s). \quad (1)$$

□

Criterion 2: Bugspot score.
Google's tool "bugspots" uses the following score [11]:

$$b(s) = \sum_{i=1}^{n} \frac{1}{1 + e^{-12t_i+12}}. \quad (2)$$

where n is the number of bug fixings in s, and t_i is the time at which the i-th bug fixing was done (for $i = 1, \ldots, n$); t is normalized such that $0 \le t \le 1$. $t = 0$ is the time at which s was born, and $t = 1$ represents the current time. Since the score of i-th bug fixing converges to $1/(1 + e^{12}) \simeq 0$ as $t_i \to 0$, and that score converges to $1/(1 + 1) = 0.5$ as $t_t \to 1$, more recent bug fixing has a larger score. Equation (2) represents the total of those bug fixing scores for s.

□

By using Criterion 1 or 2, we can quantify the bug-proneness of all source files. Then, the typical strategy is to review the files in decreasing order of the quantified bug-proneness.

B. Treatment of Newly-developed Source File

While the above strategy seems to be appropriate, there is a lack of consideration for newly-developed source files. Newly-developed files may not appear in the bug fix history because they have just been born. Thus, newly-developed files may be evaluated as the least bug-prone ones by the above criteria. However, many practitioners empirically know that newly-developed files are notable. They may review newly-developed files prior to the others; that would be a reasonable order of priority because newly-developed files may cause unknown new failures and be riskier.

When we do not have enough time or effort to review all newly-developed files, we will also have to prioritize the files in terms of bug-proneness. In such a case, process metrics do not work well, so we use product metrics instead; in this paper, we evaluate the bug-proneness of newly-developed file by using McCabe's cyclomatic complexity [2].

C. Conventional Method for Planning Code Review

We formally describe the above typical (straightforward) strategy of code review as follows. For the sake of convenience, we call it "conventional method" in this paper.

Definition 1 (Conventional Method):
Now, there are N source files $\{ s_i \}_{i=1}^{N}$ in our software being developed. Then, M out of N files $\{ s_j \}_{j=1}^{M}$ are newly-developed files; The remaining files $\{ s_k \}_{k=M+1}^{N}$ have been maintained since the previous version. Perform the code review through the following steps.

1) Evaluate $b(s_i)$. If s_i is a newly-developed file, compute $b(s_i)$ as the cyclomatic complexity; otherwise, compute it using Eq.(1) or (2).
2) Quantify the cost to review $c(s_i)$. The cost may be expressed in the time or effort required for the review of s_i. This paper simply quantifies $c(s_i)$ as the physical source lines of code (physical SLOC) of s_i in the case study (see Section IV).
3) Set the upper limit of cost L such that the total review cost cannot exceed L. L is corresponding to available time or effort for the code review.
4) Review newly-developed files $\{ s_j \}_{j=1}^{M}$ in decreasing order of $b(s_j)$. If the total cost of reviewed files reaches L, skip the code review.
5) Review the remaining files $\{ s_k \}_{k=M+1}^{N}$ in decreasing order of $b(s_k)$. If the total cost of reviewed files reaches L, skip the code review.

□

Table I
SYMBOLS USED IN DEFS.1 AND 2.

symbol	description
N	total number of source files
M	number of newly-developed source files
s_i	$i = 1, \ldots, M$: newly-developed source files $i = M+1, \ldots, N$: source files having been maintained since the previous version
$b(s)$	bug-proneness of source file s; if s is newly-developed file, $b(s)$ is computed as the cyclomatic complexity; otherwise, computed by Eq.(1) or Eq.(2).
$c(s)$	cost required for the code review of s
L	cost limit (corresponding to available time or effort)

Table II
EXAMPLE SET OF REVIEW CANDIDATES.

i	$b(s_i)$	$c(s_i)$	type
1	14	100	newly-developed
2	8	80	newly-developed
3	28	300	newly-developed
4	22	220	newly-developed
5	3	100	other
6	0	90	other
7	5	230	other
8	4	180	other
9	1	50	other
10	2	300	other

(other: source file having been maintained since the previous version)

Let us consider an example to see how to work the conventional method. Table II shows the review candidates: we have $10(= N)$ source files and $4(= M)$ files are newly-developed ones. Their bug-proneness $b(s_i)$ and cost to review $c(s_i)$ are given in the table. At the step 3) of Def.1, we consider two example cases $L = 400$ and $L = 1000$.

Case 1: $L = 400$.
Start to review newly-developed files $\{ s_1, s_2, s_3, s_4 \}$. The decreasing order of $b(s)$ is as $s_3 \to s_4 \to s_1 \to s_2$.
First, review s_3 with cost 300. After that, try to review s_4 with cost 220, but it will exceed the cost limit because $c(s_3) + c(s_4) = 300 + 220 = 520 > L$. Thus, skip the review for s_4, and try to review the next one: s_1 with cost 100. Since the total review cost does not exceed L ($c(s_3) + c(s_1) = 300 + 100 = 400 = L$), review s_1. Consequently, s_1 and s_4 are selected as our review targets when $L = 400$. The total worth is $b(s_1) + b(s_4) = 14 + 28 = 42$.

Case 2: $L = 1000$.
In this case, all newly-developed source files can be reviewed within L because the total cost of newly-developed ones is 700. Thus, consider our review plan for the remaining files with $L = 1000 - 700 = 300$. Start our review for the remaining ones $\{ s_5, s_6, s_7, s_8, s_9, s_{10} \}$. The decreasing order of bug-proneness is as $s_7 \to s_8 \to s_5 \to s_{10} \to s_9 \to s_6$. Through comparing the cost with L in a similar way explained in Case 1, s_7 and s_9 are eventually selected. The total worth is $(\sum_{i=1}^{4} b(s_i)) + b(s_7) + b(s_9) = 72 + 5 + 1 = 78$. □

III. 0-1 PROGRAMMING MODEL-BASED METHOD

The conventional method is a straightforward and easy-to-perform method for planning code review. However, the conventional method cannot produce the optimal strategy when our available time and effort are limited, because the file selection by that method focuses only on the bug-proneness of each file. In order to take into account both the bug-proneness and the review cost properly, we propose a 0-1 programming model-based method in this section.

A. Formulation as a 0-1 Programming Problem

Each file has a different bug-proneness and a different cost to review. When the available effort is limited, i.e., the total cost cannot exceed a certain limit, to make the optimal selection of files can be written as a 0-1 programming problem [12]. Now, we consider 0-1 variables $\{ x_i \}_{i=1}^{N}$ to represent whether we will review s_i or not: $x_i = 1$ means that we will review s_i, and $x_i = 0$ means that we will not. Then, we try to maximize the total worth $\sum_{i=1}^{N} b(s_i) \cdot x_i$, such that the total cost required for reviewing the selected files $\sum_{i=1}^{N} c(s_i) \cdot x_i$ does not exceed our cost limit. The aim of the 0-1 programming problem is to find the best $\{ x_i \}$ satisfying the above conditions. Such set of x_i can be the recommendation list of source files to be reviewed.

B. 0-1 Programming Model-Based Method for Planning Code Review (Proposed Method)

Now we formally define our 0-1 programming model-based method for planning code review as follows. Hereinafter, the method is referred to as the "proposed method." The proposed method also takes into account the presence of newly-developed source files mentioned in Section II.

Definition 2 (Proposed Method):
We use the symbols N, M, s_i, $b(s_i)$, $c(s_i)$ and L defined in Def.1 (see Table I), and consider the same condition.

1) Compute $b(s_i)$ and $c(s_i)$ for each file s_i in the same way as in 1) and 2) of Def.1 (for $i = 1, \ldots, N$).
2) Set the upper limit of cost L as in 3) of Def.1.
3) If all newly-developed files can be reviewed within L, then review all newly-developed files: set $x_i = 1$ for $1 \leq i \leq M$. After that, formulate the review plan for the remaining source files:

$$\text{maximize} \quad \sum_{i=M+1}^{N} b(s_i) \cdot x_i \ , \tag{3}$$

$$\text{subject to} \quad \sum_{i=M+1}^{N} c(s_i) \cdot x_i \ \leq \ L - \sum_{i=1}^{M} c(s_i) \ . \tag{4}$$

4) If all newly-developed files cannot be reviewed within L, focus only on the review of newly-developed files: set $x_i = 0$ for $M + 1 \leq i \leq N$. Then, formulate the review plan only for the newly-developed files:

$$\text{maximize} \quad \sum_{i=1}^{M} b(s_i) \cdot x_i \ , \tag{5}$$

$$\text{subject to} \quad \sum_{i=1}^{M} c(s_i) \cdot x_i \;\leq\; L \;. \qquad (6)$$

5) Solve the above problem and obtain the solution $\{ x_i \}$.

□

Nowadays, formalized 0-1 programming problems can be easily solved by solver software: there are freely available software such as lpsolve[1].

When you also want to consider a dependent relationship between source files at your code review, you can easily reflect such a relationship into the 0-1 programming problem. Now we assume that the review of file s_i requires the review of another file s_j. Then, such dependent relationship between s_i and s_j can be expressed as "$x_i \leq x_j$." If you select s_i as your review target, i.e., set $x_i = 1$, then you have $x_i = 1$ and $x_i \leq x_j$, so they imply $x_j \geq 1$. Therefore, you obtain $x_j = 1$ because x_j can only be 0 or 1. Such an inequality constraint is referred to as a "logical constraint." If your review environment needs to consider such dependent relationships, they can be reflected with logical constraints.

Let us consider the example shown in Section II-C again to see how the proposed method works.

Case 1: $L = 400$.

In this case, all newly-developed source files cannot be reviewed within L. This case is corresponding to the step 4) in Def.2. Thus, formulate the review plan only for newly-developed source files as follows (see Eqs.(5),(6)).

$$\text{maximize} \quad 14x_1 + 8x_2 + 28x_3 + 22x_4 \;,$$
$$\text{subject to} \; 100x_1 + 80x_2 + 300x_3 + 220x_4 \leq 400 \;.$$

The solution is $(x_1, x_2, x_3, x_4) = (1, 1, 0, 1)$, so we will review s_1, s_2 and s_4 in this case. The total worth is 44, that is better than the conventional method's result $(= 42)$.

Case 2: $L = 1000$.

All newly-developed files can be reviewed within L in this case, so it is corresponding to the step 3) in Def.2. Since the total cost to review all newly-developed files is 700, formulate our review plan for the remaining files as follows (see Eqs.(3),(4)):

$$\text{maximize} \quad 3x_5 + 0x_6 + 5x_7 + 4x_8 + 1x_9 + 2x_{10} \;,$$
$$\text{subject to} \quad 100x_5 + 90x_6 + 230x_7 + 180x_8 + 50x_9 + 300x_{10} \leq 1000 - 700 \;.$$

The solution is $(x_5, x_6, x_7, x_8, x_9, x_{10}) = (1, 0, 0, 1, 0, 0)$. Thus, we will review s_5 and s_8 along with all newly-developed files. The total worth of selected files is 79, that is better than the result by the conventional method $(= 78)$.

□

In both cases $L = 400$ and $L = 1000$, the proposed method can produce better solutions than the conventional method in terms of the total value of bug-proneness. We conduct a case study to examine such effectiveness of the proposed method by using real bug data in the next section.

[1]http://sourceforge.net/projects/lpsolve/

IV. Case Study

To examine the usefulness of our 0-1 programming model-based method, this section presents a case study using two popular open source software and their bug fix histories.

A. Experimental Object

This study analyzes the development of SquirreL SQL Client[2](`sql`) and Angry IP Scanner[3](`scanner`). The main reasons why we used them are as follows: (1) They are popular and have been actively developed; they are highly placed on the popularity ranking at SourceForge.net[4] which is one of the largest open source community sites. (2) Their source files are written in Java; this requirement is from our data collection tools[5]. (3) They maintain files with Git; we extract bug fixing commits by using its built-in commands.

For each software, we produce our review plan for a certain version—ver.2.6 for `sql` and ver.3.0β1 for `scanner`, respectively. Then, we examined how many buggy files are included in our recommendation lists. If a bug fixing occurred in a file until the next version—ver.3.0 for `sql` and ver.3.1 for `scanner`—, we considered the fixed file to be a buggy one. During our review planning, we regarded the files which were born after the release of the previous version (ver.2.5 for `sql` and ver.3.0α1 for `scanner`) as the newly-developed files.

B. Experimental Procedure

We conducted our experiments in the following steps.

1) Extract the files to which a bug fixing was performed until the release of the next version: we picked up the commits whose logs contain specific keywords "bug," "fix," "defect," "Bug," "Fix" or "Defect."
2) Collect metrics data: we measured the physical SLOC and the cyclomatic complexity for each file.
3) Get the bug fix histories, and produce review plans by the conventional method and the proposed method, respectively: We conducted the plannings under $L = 5\%$ to 50% (at 5% intervals) of the total physical SLOC.

C. Result

Tables III, IV and Figs. 1, 2 show the results. Since the computation is independently done for each L, a different L would have a different set of files to be reviewed. Thus, the number of buggy files may not monotonically increase with increasing L as seen at $L = 25\%$ in Fig.1; Large-sized files might become candidates when $L = 25\%$, and then they pushed others away from the recommendation list.

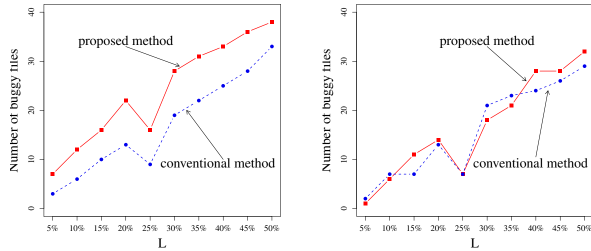
Generally speaking, the proposed method produced more effective review plans than the conventional one: more buggy

[2]http://squirrel-sql.sourceforge.net/
[3]http://angryip.org/w/Home
[4]http://sourceforge.net/
[5]http://se.cite.ehime-u.ac.jp/tool/{LOCCounter,CyclomaticNumberCounter}

Table III
NUMBER OF SOURCE FILES AND PHYSICAL SLOC.

	SquirreL SQL Client		Angry IP Scanner	
type	**#source files (#buggy ones)**	**physical SLOC**	**#source files (#buggy ones)**	**physical SLOC**
new	21(4)	2057	8(1)	1229
other	425(66)	75127	101(35)	10372
total	446(70)	77184	109(36)	11601

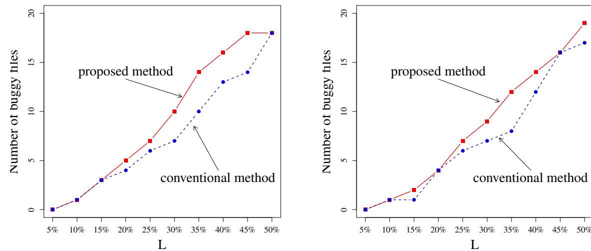
(new: newly-developed files)



(a) evaluated by Criterion 1 (number of bug fixings)

(b) evaluated by Criterion 2 (bugspot score)

Figure 1. Number of buggy files included in recommendation lists of review targets for SquirreL SQL Client.



(a) evaluated by Criterion 1 (number of bug fixings)

(b) evaluated by Criterion 2 (bugspot score)

Figure 2. Number of buggy files included in recommendation lists of review targets for Angry IP Scanner.

files are included in the recommendation lists to review produced by the proposed method. When we used Criterion 1 in which the bug-proneness is computed with the number of bug fixings, the proposed method recommended about 42% and 21% more buggy files[6] for `sql` and `scanner`, respectively. On the other hand, Criterion 2 (using bugspot score) did not work as well—the proposed method was only slightly better by about 4% and 17%, respectively[7].

In terms of the total bug-proneness, the proposed method was much better than the conventional one[8]: in Criterion 1, about 49% and 53% larger for `sql` and `scanner`, respectively; in Criterion 2, about 57% and 14% larger for them.

[6] $239/168 \simeq 1.42$ and $92/76 \simeq 1.21$ (see Table IV)
[7] $166/159 \simeq 1.04$ and $84/72 \simeq 1.17$ (see Table IV)
[8] We omit the detailed data for the lack of space.

Table IV
NUMBER OF BUGGY FILES INCLUDED IN RECOMMENDATION LISTS.

	SquirreL SQL Client				Angry IP Scanner			
	Criterion 1		**Criterion 2**		**Criterion 1**		**Criterion 2**	
L	**con**	**pro**	**con**	**pro**	**con**	**pro**	**con**	**pro**
5%	3	7	2	1	0	0	0	0
10%	6	12	7	6	1	1	1	1
15%	10	16	7	11	3	3	1	2
20%	13	22	13	14	4	5	4	4
25%	9	16	7	7	6	7	6	7
30%	19	28	21	18	7	10	7	9
35%	22	31	23	21	10	14	8	12
40%	25	33	24	28	13	16	12	14
45%	28	36	26	28	14	18	16	16
50%	33	38	29	32	18	18	17	19
total	168	239	159	166	76	92	72	84

(con: conventional method; pro: proposed method)

While Criterion 1 succeeded to recommend more buggy files as many as the total bug-proneness, Criterion 2 did not work as well. Originally, Criterion 2 (bugspot scoring) is designed to distill the source files that are suspected to be buggy. Thus, maximizing bug-proneness might not be appropriate for the bugspot score. Since there are many other promising metrics and models in MSR world, we plan to do further examinations with those metrics/models as our future work.

D. Threats to Validity

The newly-developed files were only about 5% and 7% in `sql` and `scanner`, respectively. Thus, the policy that all newly-developed files should be reviewed prior to the others might not affect our results. If there were more newly-developed files, the results might show different trend.

Since our bug data collection method was simple, incorrect data might mix in the bug fix history. While such a simple way has been commonly used in the past studies, it would be better to associate repositories with bug tracking systems. Moreover, our data of buggy files are from the commits until the next release version, so we might see a different empirical result if we use a longer bug fix history.

The bugs may include the ones that are hard to reveal by code review. While the case study showed that the proposed method can make a more cost-effective review plan than the conventional method under the same condition, the effectiveness depends on how to evaluate the bug-proneness. More detailed investigations about revealable bugs would be needed to more clearly discuss the usefulness.

Since the 0-1 programming problem is an NP-hard problem, the computation of the proposed method may need a long time and be impracticable as the number of files increases. In this study, however, all computations were completed within one second by using an average computer[9], so the proposed method would work for many practical cases.

[9] CPU: Intel Core 2 Duo 1.86GHz, Memory: 2GB, OS: Linux 3.0.77, Solver: lpsolve 5.5.2.0.

V. Related Work

Bugspots [11] quantifies the degree of hot spots for source files with bug fix history, and ranks the files for the code review. Criterion 2 of our method uses the scoring model as well. Kim et al. [13] proposed a prioritization of test cases using the exponential weighted moving average based on the test history. While the basic ideas to prioritize source files or test cases are common to our approach, our method focuses on not only such a prioritization but also the cost to review.

Mende et al. [14] and Kamei et al. [10] discuss effort-aware bug prediction models that also consider not only bug-proneness but also effort to review: for instance, they evaluate the risk of module as the number of errors divided by the effort. While they have the same research target as this study, there is an essential difference in dealing with the effort (cost). Their models use the cost as a factor of the risk evaluation, but our model uses the cost as a constraint of the 0-1 programming model (not for the objective function).

Aman [15] proposes to apply the 0-1 programming model to code review, that is the previous work of this study. However, the previous model had a lack of consideration about the file type—whether the file is newly-developed or not. Thus, the model assumes that all files are newly-developed, and evaluates them with only product metrics, so its assumption lacked a reality. The new model in this paper is designed to consider newly-developed files separately, and uses both product metrics and process metrics.

VI. Conclusion and Future Work

This paper focused on the code review based on the bug fix history. As studied in MSR world, the bug fix history is a useful information source to extract bug-prone files to be preferentially reviewed. In the context, files are prioritized by using process metrics (e.g., the number of bug fixings), then they are reviewed in decreasing order of priority.

While such a straightforward strategy seems to be useful, it can not be the optimal strategy when our available time and effort are limited. Then, we proposed a 0-1 programming model-based method to produce the optimal review plan. Moreover, the proposed model is designed to consider the newly-developed files and the others (source files having been maintained since the previous version) separately.

We conducted a case study using two popular open source software and their bug fix histories, and showed that the proposed method is up to 42% more effective than the conventional method. Since many interesting process metrics and models have been studied in MSR world, further examinations of our proposal with those process metrics and models will be our significant future work.

Although we prioritized newly-developed files, recently-fixed or largely-modified files may also be noteworthy ones in predicting bugs. We would like to make comparative studies of source files to be prioritized in the future.

Acknowledgment

The author would like to thank the anonymous reviews for their helpful comments on an earlier version of this paper.

References

[1] K. Wiegers, *Peer Reviews in Software: A Practical Guide*. Boston, MA: Addison-Wesley Professional, 2002.

[2] T. J. McCabe, “A complexity measure,” *IEEE Trans. Softw. Eng.*, vol. SE-2, no. 4, pp. 308–320, Dec. 1976.

[3] S. Chidamber and C. Kemerer, “A metrics suite for object oriented design,” *IEEE Trans. Softw. Eng.*, vol. 20, no. 6, pp. 467–493, June 1994.

[4] V. R. Basili, L. C. Briand, and W. L. Melo, “A validation of object-oriented design metrics as quality indicators,” *IEEE Trans. Softw. Eng.*, vol. 22, no. 10, pp. 751–761, Oct. 1996.

[5] T. M. Khoshgoftaar and E. B. Allen, “Controlling overfitting in classification-tree models of software quality,” *Empirical Software Engineering*, vol. 6, no. 6, pp. 59–79, 2001.

[6] H. Hata, O. Mizuno, and T. Kikuno, “Fault-prone module detection using large-scale text features based on spam filtering,” *Empirical Software Engineering*, vol. 15, no. 2, pp. 147–165, 2010.

[7] A. E. Hassan, “The road ahead for mining software repositories,” in *Proc. Frontiers of Software Maintenance (FoSM)*, 2008, pp. 48–57.

[8] N. Nagappan and T. Ball, “Use of relative code churn measures to predict system defect density,” in *Proc. 27th International Conference on Software Engineering*, 2005, pp. 284–292.

[9] A. E. Hassan and R. C. Holt, “The top ten list: Dynamic fault prediction,” in *Proc. 21st IEEE International Conference on Software Maintenance*, 2005, pp. 263–272.

[10] Y. Kamei, S. Matsumoto, A. Monden, K. ichi Matsumoto, B. Adams, and A. E. Hassan, “Revisiting common bug prediction findings using effort-aware models,” in *Proc. 2010 IEEE International Conference on Software Maintenance*, 2010, pp. 1–10.

[11] Google. (2011) Bugspots - bug prediction heuristic. [Online]. Available: https://github.com/igrigorik/bugspots

[12] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of Np-Completeness*. New York: W. H. Freeman and Coompany, 1979.

[13] J.-M. Kim and A. Porter, “A history-based test prioritization technique for regression testing in resource constrained environments,” in *Proc. 24th International Conference on Software Engineering*, 2002, pp. 119–129.

[14] T. Mende and R. Koschke, “Effort-aware defect prediction models,” in *Proc. 14th European Conference on Software Maintenance and Reengineering*, 2010, pp. 107–116.

[15] H. Aman, “Planning of priority review using 0-1 programming model with logical constrraints,” *Computer Software*, vol. 29, no. 3, pp. 115–120, Aug. 2012, in Japanese.