JOURNAL OF INFORMATION SCIENCE AND ENGINEERING 35, 175-200 (2019)
DOI: 10.6688/JISE.201901_35(1).0010

Cost-Aware Clustering of Bug Reports by Using a Genetic Algorithm*

JAEKWON LEE[1], DONGSUN KIM[2] AND WOOSUNG JUNG[3,+]
[1]*Department of Computer Engineering*
Chungbuk National University
Cheongju, 28644 South Korea
[2]*Interdisciplinary Centre for Security Reliability and Trust*
University of Luxembourg
Kirchberg, 4365 Luxembourg
[3]*Graduate School of Education*
Seoul National University of Education
Seoul, 06639 South Korea
E-mail: exatoa@cbnu.ac.kr[1]; dongsun.kim@uni.lu[2]; wsjung@snue.ac.kr[3]

The inefficient distribution of bugs to developers is increasing the cost of software development and maintenance. In efforts to tackle this issue, various studies have been carried out to recommend suitable developers for specific bugs. These studies often leverage similarity between bug reports; for example, if a developer addressed a bug report similar to a newly incoming report, that developer can be suitable to fix the bug described in the new report. However, the existing studies have resulted in imbalanced distribution – a large number of bugs can be concentrated in a small number of developers. In this paper, we propose a novel approach to achieve a cost-aware distribution of bug reports to support workload balancing. Our approach is composed of two phases. First, a set of similar report groups composed of strongly related bugs is generated based on their similarity and dependency. Clusters are then created by grouping the similar report groups so that each cluster can have similar cost (*i.e.*, minimizing its standard deviation). Our approach leverages a genetic algorithm to find a near-optimal distribution of bug reports because it is an NP-hard problem. The experiments with 1,047 bug reports collected from Mozilla's Firefox were conducted to evaluate our approach. The results showed that our approach effectively provides an appropriate solution to achieve a cost-balanced distribution of bug reports. In addition, we carried out a user study targeting 30 developers from 15 companies to figure out the usefulness and effectiveness of our approach. Among the participants, 67% answered that our approach is useful for triaging their bugs to developers. This shows the possibility for use in cases of managing or triaging bugs from the project manager's perspective.

Keywords: bug report, mining software repositories, bug triage, genetic algorithm, assignment optimization

1. INTRODUCTION

The number of bugs in software projects is increasing, and they are overwhelming developers and maintainers. According to Shokripour's research [1], more than 300 new bugs are added to the Mozilla project per day. It is thus becoming harder for bug triagers

Received October 2, 2017; revised November 20, 2017; accepted December 28, 2017.
Communicated by Shyi-Ming Chen.
* This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (Ministry of Science, ICT & Future Planning) (No. 2015R1C1A1A01054994).
+ Corresponding author.

to distribute the bug reports to developers properly and in a cost-effective manner. To support the maintenance process, ITS (Issue Tracking System) is used to manage bugs effectively [2]. However, with regard to the issue of the distribution bugs, ITS accompanies problems from the perspective of workload balancing and the manual triaging process.

One of a bug triager's important tasks is to assign bugs to appropriate developers. When a new bug is reported, the triager checks if it is a duplicate bug with previously reported bugs. It is then validated in terms of whether it contains wrong information or it is reproducible. Finally, the bug is assigned to a developer who is considered the most qualified handler for it. However, this process is tedious and can increase the cost of fixing bugs because triagers should manually deal with the reports [3].

Researchers thus have proposed various methods to find the most suitable developers to fix given bugs, based on bug assignment histories collected from ITS [1, 4-7]. For example, Cubranic and Murphy [4] proposed suitable developers recommendation approach utilizing history information how those developers handled similar bug reports. They achieved 30% accuracy on the Firefox project by recommending 5 developers. However, most of researchers focused on the issue of finding a suitable developer for a specific bug, not considering the overall cost of tackling the bug reports or a balanced distribution from the assigner's perspective.

This could delay the project schedule and eventually increase the overall development cost. Developers who newly join a project are rarely assigned to fix bugs because they have no history data in the repositories. Furthermore, the estimated cost simply based on history data is not precise. Notably, the precision of semi-automated triaging [1] decreased to 50% in a case in which only one developer was recommended. Pack *et al.* focused on reducing the cost rather than increasing the precision of triaging [8]. However, they only used the developer's bug-holding time as their cost, which could include bug-tossing [3] time or doing-nothing time as well.

We thus analyzed developer's workload assuming that often a large proportion of bugs are assigned to a small number of specific developers. We extracted all bug reports of the Firefox project, and then compared the ratio of bugs that were being fixed by active developers (which are the top 10% of developers when sorting them according to the number of reports assigned). The results showed that more than 50% of the bugs were assigned to the top 10% of the active developers on average. For the worst case, more than 80% of the bugs were assigned to the top 10% of active developers. Finally, we realized that there is an imbalanced distribution of bug reports.

In this paper, we propose a GA-based approach to achieve a cost-aware optimal distribution of bugs to support cost-balanced triaging. We defined the repository schema for the bug reports, and then implemented a set of tools to extract, analyze, and generate clusters of bugs for a given number of developers. The clustering process is composed of two phases. The first phase is generating similar report groups (SRGs) that are composed of strongly related bugs that are considered to have high similarity or dependency. Thus, they are not split into different groups in the next phase of clustering. The second phase is to generate clusters based on the total costs of the SRGs. In this phase, the standard deviation of the cluster's cost is used as a fitness of our optimization process.

In short, our approach entails finding a near-optimal distribution of the given bugs, which the distribution enables a cost-balanced assignment from the manager's point of

view. Our method is thus effective when assigning a massive number of bugs to a development team in a cost-balanced manner.

Additionally, our method does not depend on the history data of the repositories, which also differentiates it from previous approaches that use such data for cost estimation [8, 9]. Thus, our approach can be applied for any given set of bug reports. Based on the document readability [10], we also defined a reading cost metric to calculate the reading cost of the bugs. The cost metric can be easily enhanced without changing the framework of the proposed method. The results of optimization demonstrate the effectiveness of our approach by showing that the standard deviation of the final bug clusters' cost can reach below 0.07. We received 67% positive responses to our approach from developers who participated in a survey.

We can summarize the contributions of our research as follows:

- A near-optimal assignment of bugs to a group of developers can be conducted via the concept of load balancing.
- A novel cost metric to comprehend a set of bug reports is defined by considering similarity and dependency of bug reports.
- A methodology including overall schema and process is provided in the case of managing or triaging bugs from the project manager's perspective.

The remainder of this paper is organized as follows. Section 2 describes the motivation of our research with the analyzed distribution data of bugs in open source projects. Section 3 describes related work. Section 4 presents the assumptions of our research and formally defines the balanced distribution of bug reports. Section 5 explains in detail our approach to generate balanced clusters of bug reports. Section 6 shows the results of experiments to evaluate the usefulness and effectiveness of our method and discusses our research including threats to validity. Finally, section 7 provides our conclusions.

2. MOTIVATION

Contemporary software projects including open source software (OSS) often employ issue tracking systems to efficiently deal with issues such as bugs and feature requests. Users can submit bug reports to the system when faced with abnormal behaviors on a software. The submitted bug reports are assigned to developers by a triager (*e.g.*, product manager). Triagers of OSS projects often assign bug reports based on developer expertise [5].

However, bug report assignment can be easily imbalanced. Jeong *et al.* [3] stated that a few developers can take too many bug reports after a small number of report tossing. This imbalanced distribution may delay overall software development since this can impose an excessive burden on a few developers. In particular, this problem can be more serious because more than 300 bug reports are filed every day in many recent OSS projects [1].

To determine whether an imbalanced distribution occurs in OSS projects, we investigated bug reports of Mozilla Firefox[†]. We collected 142,217 bug reports from Mozilla's issue tracking system from the beginning of the project to August 31, 2014. For each bug

[†] https://bugzilla.mozilla.org/

report, we examined changes in status and assignee fields.

Since some bug reports do not follow normal status transitions [2, 11] and change *assignee*, we collected *status* transitions based on the following rules:

- Rule 1: A user who changed the status into *ASSIGNED* is the assignee when no assignee information is provided.
- Rule 2: A user who changed the status into *RESOLVED* when no assignee information is provided.
- Rule 3: The last value of the *assignee* field is regarded as the actual assignee. Previous assignees and status changes are not used.
- Rule 4: If two different users changed the status of a single report into *ASSIGNED* and *RESOLVED*, respectively, the user who changed it into *RESOLVED* is regarded as the assignee.
- Rule 5: If no explicit *ASSIGNED* status is provided while the *assignee* field is available, the creation time of the bug report is regarded as the assigned time.
- Rule 6: If there is no explicit *RESOLVED* status, a user who changed the status into *VERIFIED* or *CLOSED* is regarded as the assignee.

We examined how many bug reports are assigned and owned by each developer. For each developer, we counted two numbers: (1) the number of reports that are newly assigned and (2) the number of reports that are being processed, respectively. Since our focus is on figuring out whether a few developers assume an imbalanced workload within a specific period rather than the entire life cycle, we investigated bug reports within a four-week window, as shown in Fig. 1. We slid this window by one week.

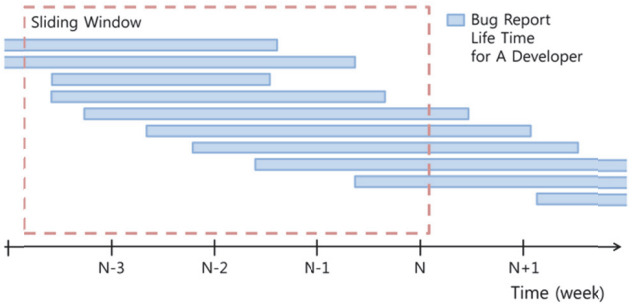


Fig. 1. Inspection of the number of bug reports for a certain developer within a sliding window (number of assigned bug reports = 7, the number of held bug reports = 9).

Fig. 2 shows the proportion of bug reports newly assigned to developers. We split the developers into two groups: (1) top 10% and (2) other 90%, after sorting them according to the number of reports assigned. As shown in Fig. 2 (a), on average more than 60% of bug reports were assigned to the top 10% of developers. Since there might be many highly inactive developers in the 90%, we split the 10% again into two subsets of 10 & 90% and examined the distribution. The results are shown in Fig. 2 (b). For active developers, more than 50% of reports were assigned to the top 10% of developers on average. This implies that the current practice distributes reports in an imbalanced manner.

(a) Two groups of developers in all developers. (b) Two groups of developers in top 10% developers.

Fig. 2. Rate of assigned bug reports into two groups after sorting them according to the number of reports assigned each week from August 2002 to August 2014 (window size = 4 weeks, sliding unit = 1 week).

Table 1. Bug resolution time between the top 10% at least once and the rest of developers (based on the results of Fig. 2 (b)).

	Count of Developers	Count of Resolved Bugs	Total Elapsed Time (day)	Average Elapsed Time per a Bug (day per a Bug)
Top 10%	134	95,119	19,135,767	201.2
Rest	170	14,612	2,050,431	140.3
Top 10% / Rest	0.8	6.5	9.3	1.4

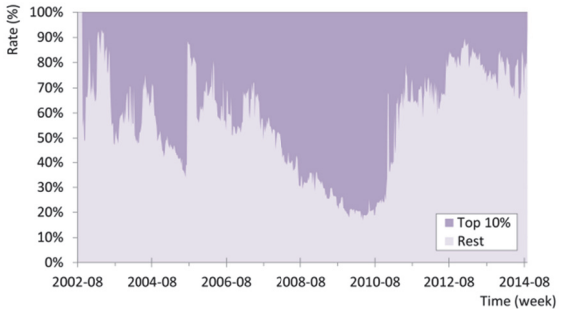


Fig. 3. Rate of held bug report each week from Aug 2002 to Aug 2014 (window size = 4 weeks, sliding unit = 1 week).

In addition, we investigated the number of reports that each developer held within a window. If the top 10% of developers can resolve bugs faster than others, they would hold numbers of reports similar to those of other developers. However, it turned out that the top 10% of developers held more than 40% of reports on average, as shown in Fig. 3. This is not significantly different from the results shown in Fig. 2.

This imbalanced distribution affected the bug resolution time. We examined bug reports resolved by developers who had been involved in the top 10% at least once and the rest of developers (based on the results shown in Fig. 2 (b)). We then measured the time between the assignment and resolution for each bug report. Table 1 shows the results. The top 10% of developers resolved 6 times more reports than other developers and spent 9 times longer time for them. For each report, the top 10% of developers spent 1.4 times longer time than others. This indicates that the top 10% developers may have an excessive burden to resolve bugs on time.

More than 60% of bug reports are assigned to 10% of developers on average while they spend 1.4 times longer time to resolve the reports than other developers.

Based on the above observations, in this paper we present a novel approach, CBAC (Cost-aware BAlanced Clustering), to balance the distribution of bug reports. This approach takes the reading cost of each bug report into account and uses a genetic algorithm to achieve a balanced distribution of reports.

3. RELATED WORK

3.1 Triaging Bug Reports

There have been various studies related to bug assignment or triaging. These studies initially focused on finding suitable developers by using triaging histories on who had handled what bug reports in the past.

Cubranic and Murphy proposed a semi-automatic bug triaging method [4] inspired by Fischer's research to extract the software features from code history and a set of bug reports [12]. They transformed textual summaries and descriptions into computable feature-vectors based on TF-IDF. The vectors being mapped to the bug assignees were then classified by using the naïve Bayesian technique. The experimental results showed 30% accuracy in triaging the bug reports of the Eclipse project.

Anvik *et al.* proposed a novel technique [5] using the support vector machine algorithm to improve the previous semi-automatic triaging [4]. They used a heuristic approach to build a word-vector from bugs and labeled them with related developers. The precisions of the algorithm applied to Eclipse and Firefox were 57% and 64%, respectively, which were 4~20% higher than in previous cases.

On the other hand, Matter *et al.* focused on extracting developer expertise from source codes to enhance the bug triaging process [13]. To represent developer expertise, they created a term-author matrix by extracting bag of words from all source code histories. The temporal decay was also considered by diminishing the weight of expertise as time passed since the code had been changed. When they have a new bug report, they build a term-vector of the bug report and compare it with the term-author matrix. Their approach showed 33.6% top-1 precision and 71% top-10 recall.

Bhattacharya extended Jeong's tossing graph [3] to exclude inactive developers by considering the period of their activeness [14]. Comparison experiments with various machine learning algorithms were conducted to find the best solution to bug assignment, naïve Bayes showed 86% accuracy.

Shokripour *et al.* proposed a two-phased location-based triaging method [1] to resolve this problem. They built a weighted noun-index using simple terms in order to predict the location of each bug and assign a proper developer. Their method showed 89% and 59% precisions for JDT and Firefox, respectively, a 70% improvement over the previous approach [15].

However, most of them focused on the issue of finding a suitable developer for a

specific bug, not considering the overall cost of tackling the bug reports or a balanced distribution from the assigner's perspective. As a result, it is difficult to provide a fully automated solution to those problems.

3.2 Cost Estimation of Handling Bug Reports

There have been several studies on estimating the cost of handling bug reports. Weiss *et al.* estimated the modification time of a newly given bug by using the average time previously to have similar bugs fixed [9]. Their approach showed 30% precision for the JBoss data. However, the method was only applicable for systems capable of recording the effort history.

Park *et al.* proposed a solution to this issue by considering the developer's bug-fixing cost [8]. The purpose of their work was to minimize the fixing cost and maximize the triaging precision at the same time. Their approach showed that a 10% decrease of the precision could cut the total fixing cost in half for open source projects such as Apache, Eclipse, Linux Kernel, and Mozilla. Their research showed the importance of a cost-balanced distribution of bugs in triaging. However, the time gap between bug-assigned time and bug-resolved time was considered as the fixing cost for a bug.

Xiao and Afzal also utilized the estimated effort to fix bugs to tackle bug reports with GA-based scheduling in the software testing phase [16]. They also defined models for the bug report and for the developer with fundamental information such as required skill, competency, experience, and workload. However, this required manual evaluations for all data in order to predict the fixing cost of each bug.

In most approaches, the time difference between the time of a bug being assigned and it being fixed was considered as the cost of the bug. For these approaches, however, It is necessary to acquire the fixing history of similar bugs, and it is difficult to estimate the cost of bug reports with short-term history. From this point of view, our approach uses the size, difficulty, and similarity of the documents to predict the cost of bugs.

3.3 Grouping Similar/Duplicate Bug Reports

Several studies have been conducted to detect bug reports describing the same problem, not only because duplicate bugs can compensate for insufficient information but also because they can confuse developers and lead to inefficiency [17]. Runeson *et al.* studied the detection of duplicate bug reports based on natural language processing for the first time [18]. They extracted summaries and descriptions from open source projects, and then generated a feature-vector based on TF-IDF. The cosine similarity algorithm was used to compare the similarity between each pair of bugs and a time frame was given to reduce the number of bugs to be compared. By investigating the top 5 high similarity bugs, they found 30% duplicate bugs Sony Ericsson's project.

Wang *et al.* extended Runeson's method [18] and improved the accuracy of the duplicate bug detection algorithm by using execution information as well as natural language data [19]. In particular, the execution information was a better source for detecting similar internal behaviors of defects than a text-based description, which was limited in terms of representing their external behaviors. Wang's approach showed better accuracy than Runeson's for the Eclipse and Firefox projects.

Sun *et al.* used an SVM-based machine learning algorithm to train a model using duplicate and non-duplicate pairs of bugs extracted from a repository [20]. Their experiment to recommend the top 5 duplicated bugs showed 50% recall for the Eclipse, Firefox, and OpenOffice projects. However, no obvious improvement was found in the case of excluding manually tagged information.

Nguyen *et al.* used the Latent Dirichlet Allocation as well as a text-based retrieval technique to detect duplicate bugs expressed in different terms [21]. They defined a topic model based on LDA, and combined it with BM25 to resolve the issue. The results of recall applying DBTM to the Eclipse project were 57%, 76%, and 82% for the top-1, top-5, and top-10 recommendations, respectively. These results were 20% higher than previous methods [22]. Somasundaram and Murphy used the Latent Dirichlet Allocation with Kullback-Leibler divergence to correctly assign bug reports that had been categorized into the wrong components [23]. Zhang and Lee grouped bug reports that had been included in the same component. They then compared the bug reports located in the same group to detect duplicate bugs [24].

Several approaches applied machine learning techniques based on similarity metrics to find duplicate bugs. In order to apply the machine learning techniques to duplicate bugs, the number of clusters or training sets should be provided. To address these constraints, we adopted threshold values for the similarity metric to group them into undivided SRGs. We also considered fuzzy inference that is utilized for document clustering [25], temperature prediction [26], and forecasting enrollments [27] to detect duplicate bugs. However, we chose a simple similarity algorithm based on TF-IDF because of their complexity.

3.4 Optimization Algorithms

There are many approaches to find optimal solutions such as Simulated Annealing [28], Ant Colony Optimization [29], Genetic Algorithm [30, 31], and fuzzy-related algorithms [32-35]. Since finding a solution from optimization problem is too complex to compute all the possible cases, these algorithms are used to obtain the near-optimal solution instead of the real-optimal solution.

Simulated Annealing [29] is a global search algorithm that mimics the physical annealing of solids. It starts from a random position and moves toward the optimal solution progressively. Jumping to a far distance from a current optimal solution with a small probability is used to prevent the algorithm from falling into local optimization.

Ant Colony Optimization [30] is a global search algorithm that mimics the way how ants find food. The initial goal was to find the optimal path in a graph. The algorithm moves randomly to the direction of a target node. When it reaches the target node, it moves back to the start node with the pheromone. After the several iterations, the best pheromone intensity path will be selected to determine the optimal solution. This approach is used as a global search algorithm in various fields such as routing [36] and scheduling [37].

Genetic Algorithm [31] is one of the evolutionary algorithms that is inspired by the process of natural selection. It creates a population that is an initial set of solutions and evolves the population through operators such as crossover, mutation, and selection iteratively. This process provides an effective way to find globally optimal solutions by keep-

ing wide-range of candidates in a set of solutions. To apply the Genetic Algorithm, it is not only necessary to be able to convert a solution of the problem into a chromosome but we also need to define a fitness function to evaluate the solution.

4. COST-AWARE BUG REPORT CLUSTERING

This section defines a balanced clustering problem of bug reports with respect to the cost of the bugs. As described in section 2, the overall defect resolution can be delayed if bug reports are assigned to developers in an imbalanced manner. It is thus necessary to efficiently distribute bug reports to developers so that they can spend similar amounts of time in resolving bugs.

In this paper, it is assumed that a balanced bug distribution is necessary from the perspective of production managers. If a few developers take too many bugs, vulnerability to production delay is greatly increased. Thus, bug triagers should focus on balancing the cost of bugs assigned to each developer. A balanced distribution can have a higher priority than expertise matching. Otherwise, there will be increased vulnerability to delayed project releases.

Our goal is to distribute bug reports to each developer in a balanced way based on the assumption described above. To formally define this problem, it is necessary to clarify several concepts in the bug assignment process. For a given n number of bug reports and m number of developers, which should be the same as the number of clusters to be generated, let $B = \{b_1, b_2, \ldots, b_n\}$ be a set of the bug reports. Bug assignment can be determined by the n-dimensional vector:

$$\boldsymbol{a} = \langle a_1, a_2, \ldots, a_n \rangle \tag{1}$$

where a_i is an index of a cluster to which a bug report b_i belongs ($1 \le i \le n$, $1 \le a_i \le m$). Then, a cluster with an index j, which is actually a set of bug reports, can be defined as $c_{\boldsymbol{a},j}$ ($1 \le j \le m$) for an assignment $\boldsymbol{a}$.

Let $C_{\boldsymbol{a}} = \{c_{\boldsymbol{a},1}, c_{\boldsymbol{a},2}, \ldots, c_{\boldsymbol{a},m}\}$ be a set of clusters made by an assignment $\boldsymbol{a} \in A$, where A is a universal set of assignments. Each bug reports are assigned to clusters. Later, each cluster will be assigned to a certain developer for a balanced report distribution.

For a certain assignment $\boldsymbol{a}$, it is necessary to compute the total cost of assigned bug reports for each cluster. Let $Cost_B$ be a function defined to compute the cost of bug reports as follows:

$$Cost_B: B \to \mathbb{R}. \tag{2}$$

Based on $Cost_B$, the total cost of bug reports in cluster $c_{\boldsymbol{a},j}$ for a certain assignment $\boldsymbol{a}$ can be defined as:

$$Cost_C(c_{\boldsymbol{a},j}) = \bigcup_{b \in c_{\boldsymbol{a},j}} Cost_B(b). \tag{3}$$

The cost of a set of bug reports is not just the sum of each bug report's costs because of their similarity and dependency. The details related to this issue are described in sec-

tion 5.

Our goal is to minimize deviation of estimated effort to fix bugs among developers. As shown in Eq. (4), it is necessary to find an optimal and balanced distribution $\boldsymbol{a}_{opt}$ in the set of assignments A when dividing n reports for m developers.

$$a_{opt} = \arg\min_{a \in A} \sigma_{j=1}^{m} \{Cost_C(c_{a,j})\} \tag{4}$$

Finding $\boldsymbol{a}_{opt}$ is a combinatorial optimization problem and its computation complexity class is NP-hard and intractable. For example, the number of cases to assign 20 documents to 5 developers is simply calculated as 5^{20} which is greater than 90 trillion. A meta-heuristic approach such as genetic algorithm provides effective optimal solutions for such complex problems. Therefore, local search techniques may not be suitable for this problem. In section 5, this paper presents a novel approach to balanced clustering of bug reports for a given set of developers.

5. APPROACH

This section describes our approach, CBAC (Cost-aware BAlanced Clustering), to cost-aware bug report clustering.

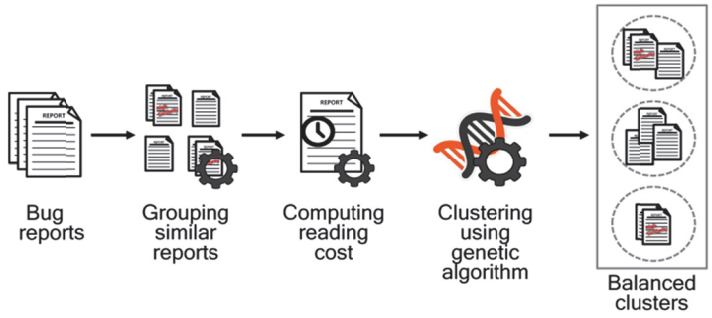


Fig. 4. Overview of our approach, CBAC (Cost-aware BAlanced Clustering).

5.1 Overview

CBAC is an approach to achieve cost-aware bug report clustering, as described in section 4. This approach leverages a genetic algorithm to find balanced clusters of bug reports for a given number of developers. This process prevents a few developers in a project from being overwhelmed by a large number of bug reports.

As shown in Fig. 4, CBAC has three sub-tasks. First, it measures the similarity of bug reports and groups similar reports together. It then computes the reading cost of individual bug reports and report groups. After obtaining the reading cost, the approach leverages a genetic algorithm to cluster bug reports with respect to a balanced distribution of cost.

CBAC uses only the summary (*i.e.*, title) and description of a bug report. These fields contain the initial information for a given bug that the reporters are faced with. Since the goal of this paper is to obtain balanced clusters of bug reports with respect to

reading cost rather than achieving better report assignment, we focused on textual information.

The remainder of this section describes the tasks of CBAC in detail.

5.2 Bug Report Reading Cost

To compute the cost of a given bug report, CBAC uses the document reading cost. This cost represents the effort to understand a specific document. Since bug reports are documents written in natural languages, reading and understanding bug reports take a significant amount of debugging effort. Thus, reading cost can be an effective indicator for measuring the cost of a bug report.

Our approach uses three metrics to calculate the reading cost of a bug reports: (1) word difficulty; (2) sentence complexity; and (3) document length. First, difficult or unfamiliar words can increase the cost of document reading since a document reader may need to search in a dictionary or on the web for the concept of the words. Complex sentences can make a reader take more time to understand the document. In addition, the size of a document (*e.g.*, the number of words or sentences) may inherently increase the effort needed to read and understand it.

To compute the word difficulty and sentence complexity, this approach uses the Flesch-Kincaid equation [10]. This equation improves *Flesch Reading Ease* [38]. The United States Department of Defense uses this equation to evaluate its documents [39]. As shown in Eq. (5), the *Flesch-Kincaid* equation measures the reading cost of a document b (*i.e.*, a bug report in this approach) by using the following two terms: (1) the average number of syllables in words to compute word difficulty and (2) the average number of words in sentences to compute sentence complexity.

$$FK(b) = 0.39\left(\frac{word\_count(b)}{sentence\_count(b)}\right) + 11.8\left(\frac{syllable\_count(b)}{word\_count(b)}\right) - 15.59 \quad (5)$$

CBAC defines the reading cost for a document b as $Cost_B(b)$ based on the *Flesch-Kincaid* equation and document size (the number of words) as shown in Eq. (6). $Cost_B(b)$ is a product of the *Flesch-Kincaid* value and the number of words in a document b. We set the coefficient of the first term to 10^{-1} since 10 is the minimum level of readability for ordinary documents based on the US government requirements [40]. As $FK(b)$ yields [0, 20], it is normalized into [1, 3]. Similarly, the approach normalizes the second term (*i.e.*, document size) by 100 so that it will not dominate the first term.

$$Cost_B(b) = \left(\frac{FK(b)}{10} + 1\right)\left(\frac{word\_count(b)}{100}\right) \quad (6)$$

5.3 Grouping Bug Reports

CBAC groups similar bug reports together before clustering the reports. Reading similar documents can reduce reading cost; after reading a bug report, the reading cost of another similar report can be lower than the cost of understanding the report independently. Thus, it is necessary to put similar bug reports into a group.

5.3.1 Bug report similarity

To compute the similarity of a pair of bug reports, CBAC leverages the Vector Space Model (VSM) [41]. This approach transforms text in a bug report (title and description) into a vector. The vector consists of TF-IDF (term frequency-inverse document frequency) [42] values of each word in the bug report; this is computed by Eq. (7). Let t be a term used in report b. $tf(t, b)$ is the scaled frequency of the raw term frequency $f(t, b)$. B is a set of bug reports and $N(t, B)$ is the number of documents in which the term t appears. $idf(t, B)$ is the inverse document frequency of the term t in B. $tfidf(t, b, B)$ is the TF-IDF value of the term t in the report $b \in B$.

$$\begin{aligned} tf(t,b) &= 1 + \log f(t,b) \\ idf(t,B) &= 1 + \log \tfrac{|B|}{N_t} \\ tfidf(t,b,B) &= tf(t,b) \cdot idf(t,B) \end{aligned} \tag{7}$$

Cosine similarity is used to compute the similarity between reports. CBAC performs a pair-wise comparison between all bug reports to determine similar report groups (SRGs). Eq. (8) shows the similarity equation used in our approach:

$$Sim(V_1, V_2) = \frac{V_1 \cdot V_2}{|V_1||V_2|} \tag{8}$$

where V_* is a term vector (its elements are computed using Eq. (7)) of bug reports.

CBAC uses the similarity values to identify SRGs. The approach defines a threshold value when identifying report groups. Once a subset of bug reports is classified into an SRG, it is dealt with by single developers so that the reading cost will be reduced.

To determine an appropriate threshold value, we conducted a preliminary user study. In this study, we asked five developers whether a specific threshold is effective to identify SRGs. For this study, we collected 1,047 bug reports of Mozilla Firefox submitted from July 1, 2013 to July 30, 2013. We then computed the pair-wise similarity of the reports. Among these reports, we selected 41 pairs of similar bug reports in which their similarity values were higher than 0.5.

We provided the pairs to the five developers and asked them to evaluate the effectiveness of each pair with respect to reducing reading cost. The effectiveness was evaluated by the five-level Likert scale as shown in Table 2.

Table 2. Likert scores to determine appropriate threshold.

Likert Score	Description
5	Very Helpful
4	Helpful
3	Normal
2	Rarely Helpful
1	Never Helpful

Fig. 5 shows the results of the similarity evaluation. The X-axis represents threshold values (for each 0.01 unit) and the Y-axis is the ratio of each answer shown in Table 2.

We supposed that report pairs with scores of 4 and 5 were positive to be dealt with together by a single developer. Based on the results, we selected 0.86 as the threshold value for report grouping since this value indicates that at least 90% of report pairs received more than a score of 4 according to the results shown in Fig. 5.

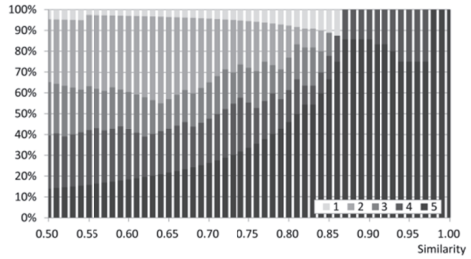


Fig. 5. Results of similarity as evaluated by developers.

5.3.2 Similar report groups and their reading costs

Once CBAC computes pair-wise similarity of bug reports, the approach put similar reports into a group. First, the approach builds a graph in which bug reports are nodes. Edges can be defined between a pair of bug reports (b_1 and b_2) if $Sim(V_{b_1}, V_{b_2}) > 0.86$. After connecting the nodes by edges, the approach identifies cliques (*i.e.*, SRGs) in the graph; these SRGs are the unit of clustering.

Since reading similar bug reports together may take less time, it is necessary to redefine the reading cost metric of bug reports in a group rather than computing the sum of the cost of all bug reports. CBAC uses a maximum spanning tree algorithm [43] to compute the reading cost of a report group. The algorithm identifies the minimum number of edges with maximum values (here, similarity), that connects all nodes in the group without any cycle.

After identifying the spanning tree of an SRG g, the approach uses Eq. (9) to compute the reading cost of the SRG. Let g be a set of reports $\{b_1, b_2, \ldots, b_k\}$:

$$Cost_G(g) = \sum_{b_i \in g} Cost_B(b_i) - \sum_{(b_i, b_j \in g) \wedge (i<j)} Sim(b_i, b_j) \times \min\{Cost_B(b), Cost_B(b_j)\} \quad (9)$$

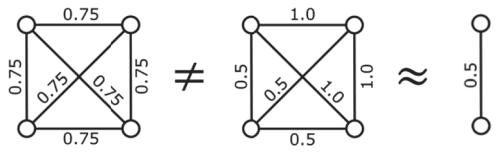
where $min\{x, y\}$ returns the minimum value between x and y ($x, y \in \mathbb{R}$).

Finally, because these SRGs are the unit of clustering, $c_{\boldsymbol{a},j}$ can be re-defined as a set of SRGs: $\{g_1, g_2, \ldots, g_r\}$. This makes it easier to calculate the reading cost of cluster $c_{\boldsymbol{a},j}$. Therefore, the total cost of bug reports in cluster $c_{\boldsymbol{a},j}$ for a certain assignment $\boldsymbol{a}$ can be re-defined as:

$$Cost_C(c_{\boldsymbol{a},j}) = \sum_{g \in c_{\boldsymbol{a},j}} Cost_G(g). \quad (10)$$

Fig. 6 shows three groups of bug reports, among which groups (a) and (b) have the same average similarity, 0.75. Every pair of bug reports in group (a) has 0.75 similarity. Group (b) includes three duplicate bug reports having 1.0 similarity in each pair while

the others have 0.5 similarity for each. However, group (b) should have a reading cost w similar to that of group (c), which has a 0.25 lower average similarity compared to groups (a) and (b). This is because the total reading cost of the three duplicate bug reports should be close to the cost of one bug report. The reading cost calculated from Eq. (9) produces exactly the same result as well.



(a) (b) (c)

Fig. 6. Two types of situation having the same average similarity but different reading cost.

5.4 Report Clustering

Based on the reading cost (section 5.2) and report groups (section 5.3), CBAC uses Algorithm 1 to search for nearly optimized clusters of bug reports. This is a genetic algorithm that takes a set of bug report groups and gives *m* clusters that contain a specific subset of report groups, where *m* is the potential developers. The user of this algorithm can provide the population size, maximum generations, crossover rate, and initial/dynamic mutation rate as parameters.

To execute Algorithm 1, we need five parameters: the size of the population, the maximum number of generations, crossover/mutation rates, and the dynamic mutation rate. In addition, the algorithm takes a set of bug reports and the number of developers as inputs. Using these parameters and input values, the algorithm returns a best-so-far chromosome in the population, which is the clustering information of the reports.

Algorithm 1 starts with *population_size* of arbitrary chromosomes generated by the *initial_population* function. These chromosomes are considered as base population P to produce children and to find the best solution. The algorithm then initializes *children_size* by multiplying r_c and *population_size*. The population, P, is reproduced until the number of generations reaches *generation_max*. In each generation, the algorithm produces *children_size* individuals by using the following three operators (see section 5.4.3 for more details): (1) the *selection* operator takes two individuals from P to be parent chromosomes, p_1 and p_2 (2) the *crossover* operator combines two parent chromosomes to reproduce *offspring* and (3) the *mutation* operator changes digits of the *offspring* chromosome with the probability r_m. r_m is adjusted by r_{dm} for each generation. All offspring replace the one of their parents with the lower fitness value [44].

By comparing the fitness values, the algorithm then puts the *offspring* into $P_{children}$ and a weak one of the parents to eliminate into $P_{deprecated}$. After generating offspring, the algorithm conducts a replace operation that deletes $P_{deprecated}$ from P and adds $P_{children}$ into P.

The remainder of this section explains the representation, fitness function, and mutation operators necessary for Algorithm 1.

Algorithm 1: Algorithm for cost-aware balanced clustering of bug reports

Input: R: a set of bug reports
Input: D: the number of developers (*i.e.*, the number of clusters)
Output: *solution*: best-so-far chromosome (*i.e.*, clusters)
Param: *population_size*: the size of population
Param: *generation_max*: the maximum number of generations
Param: r_c: crossover rate
Param: r_m: mutation rate

```
1  P ← initial_population (population_size, R, D);
2  children_size ← population_size × r_c; //The number of solutions will be created in a
       generation.
3  for generation ← 1 to generation_max do
4  |   P_children ← {};              //Solutions which is created in this generation
5  |   P_deprecated ← {};            //Solutions which will be removed in this generation
6  |   for cnt ← 1 to children_size do
7  |   |   p1, p2 ← selection(P) offspring ← crossover(p1, p2)
8  |   |   if random() < r_m then
9  |   |   |       //mutation will be applied as r_m probability.
10 |   |   |       offspring ← mutation(offspring)
11 |   |   end
12 |   |   P_children ← P_children ∪ offspring
13 |   |   deprecated ← min_fitness(p1, p2);
14 |   |   P_deprecated ← P_deprecated ∪ deprecated;
15 |   end
16 |   P←replace(P, P_children, P_deprecated);
17 |   r_m←r_m × r_dm;
18 end
19 solution ← best_individual(P);
20 return solution
```

5.4.1 Representation

Fig. 7 shows the chromosome and the clusters used in our approach. The chromosome can represent one possible assignment of SRGs to clusters, which is actually an assignment of bug reports to clusters, as described in Eq. (1) of section 4.1. The position and digit of a gene indicate the indexes of an SRG and the cluster to which the SRG is assigned, respectively. Thus, the bug reports in an SRG are always assigned to an identical cluster.

For example, the chromosome in Fig. 7 (a) indicates that the five SRGs are assigned to clusters 2, 1, 2, 4 and 3. Fig. 7 (b) is a graphical representation of clusters based on the chromosome shown in Fig. 7 (a). Note that reports in a single SRG are assigned to a single cluster.

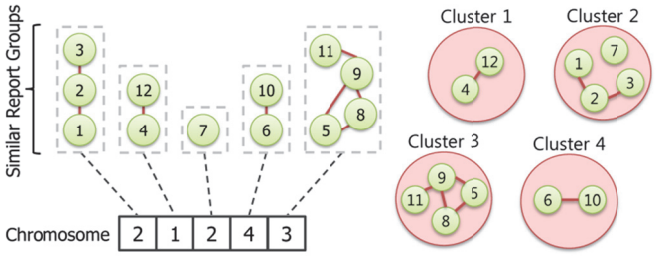


Fig. 7. Representation of a chromosome and its clusters (cluster size = 4).

5.4.2 Fitness

The fitness function in our approach evaluates whether a set of clusters represented in a chromosome is balanced with respect to the cost of bug reports in the clusters. The approach uses the following equation to compute the fitness value for a certain assignment $\boldsymbol{a}$:

$$f(a) = \sigma_{j=1}^{m} \{Cost_C(C_{a,j})\} \tag{11}$$

where $\boldsymbol{a}$ is a chromosome with length m while $Cost_C(C_{a,j})$ is the function described in Eq. (10). This fitness function calculates the standard deviation of the entire group reading cost for the report groups in $\boldsymbol{a}$. The goal of Algorithm 1 is to find a near-optimal chromosome (*i.e.*, m clusters) that minimizes the value of Eq. (11).

5.4.3 Operators

In this approach, we use selection, crossover, and mutation operators for the reproduction of chromosomes. Roulette selection [45] is used for the selection operator. This selects an individual based on its fitness value; an individual with a higher fitness value has a higher probability of being selected.

Uniform crossover [46] is used for the crossover operator in which each gene has the same probability of contributing to the reproduction of offspring. The mixing ratio is fixed at 0.5 so that one offspring has approximately half of the genes from the first parent and vice versa.

To avoid empty clusters, this approach repairs a chromosome if it can result in any empty cluster. Since the goal of this approach is to find m clusters with a balanced cost distribution, a chromosome with any empty cluster is obviously abnormal. Thus, CBAC repairs the chromosome by using parent chromosomes, as shown in Fig. 8. The repairing task searches for the leftmost gene of the first parent, which is assigned to the cluster that is empty in the offspring. We then copy it into the same location in the offspring's chromosome from the first parent. This task repeats the repairing until there are no empty clusters in the offspring. In the worst case, it might make one offspring identical to one of the parents. However, we regard this to be better than chromosomes with empty clusters. If this worst case occurs too frequently, the number of clusters m will be too large with respect to the number of reports n.

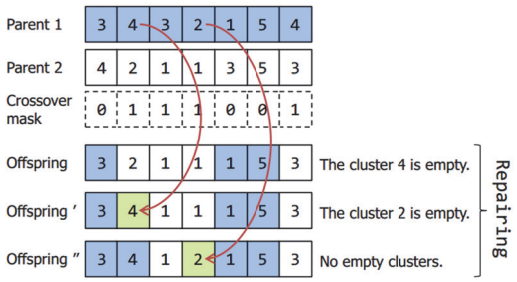


Fig. 8. Process of repairing an incorrect chromosome.

For the mutation operator, the approach utilizes the dynamic mutation rate [31]. To prevent radical mutation, the initial mutation rate in this approach is set to 0.043. For every generation, it is multiplied by 0.998 to dynamically change the mutation rate. For example, the rate is adjusted to 0.006 after 1,000 generations.

6. EVALUATION

This section describes the evaluation results of CBAC. The evaluation consists of (1) experiments; and (2) a user study. The experiments were conducted to figure out how CBAC performs when different parameters are given. The results of the experiments can guide users of our approach. The user study examined the effectiveness of the approach. The study designed and carried out a survey in which we asked 60 developers about the usefulness of our approach.

In particular, our evaluation addressed the following research questions:

- RQ1: What are appropriate values for parameters?
- RQ2: Is CBAC useful for a balanced distribution of bug reports with respect to cost?

RQ1 and RQ2 are answered in sections 6.1 and 6.2, respectively.

6.1 Experiments

This section describes the results of our experiments with various parameter values to evaluate our approach. In these experiments, we used 1,047 bug reports submitted from July 1, 2013 to July 30, 2013. They were collected from Mozilla's issue tracking system and the list of the reports is available at: https://sites.google.com/site/geneticcbac/.

Basically, we used the following parameter values as a baseline:

- # of clusters (*i.e.*, developers): 25
- threshold value for report grouping: 0.86
- maximum generation: 1,000
- size of population: 1,000
- selection operator: Roulette Wheel (selection pressure=4)
- crossover operator: uniform crossover (crossover rate=0.9)
- mutation operator: uniform mutation (initial rate=0.043)

First, we used different population sizes. As shown in Fig. 9, we varied the size (P) from 100 to 1,500. In addition, we evaluated six different numbers ranging from 5 to 30 for the number of clusters (m). When $m = 5$, $P > 200$ is sufficient to find appropriate clusters while $P > 700$, $P > 1{,}000$, and $P > 1{,}300$ are sufficient for $m = 15$, $m = 25$, and $m = 30$, respectively. This implies that it is necessary to proportionally increase the size of the population (P) when applying our approach to a larger number of clusters (m).

When the size of clusters is 25, an appropriate size of population is 1,000.

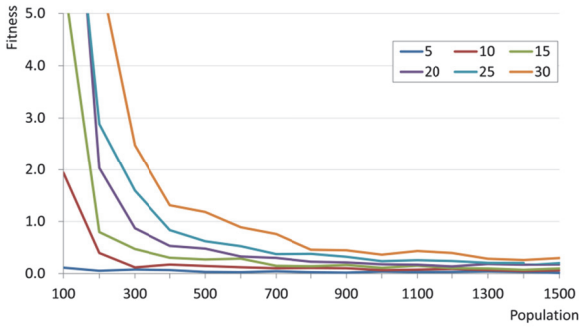
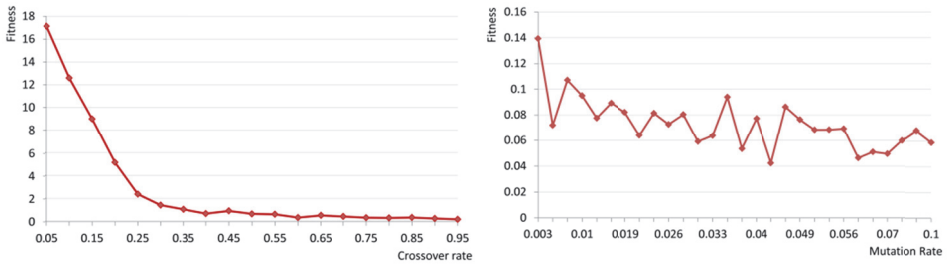


Fig. 9. Fitness values after 1,000 generations for each population size.

We then varied the crossover and mutation rates when reproducing offspring. The crossover rate determines the portion of individuals in the population that will be used as parent chromosomes of the crossover operator (see section 5.4.3). If this rate is too high, the operator takes a proportionally longer time.

As shown in Fig. 10 (a), the fitness value was saturated when the crossover rate is 0.40. We varied the rate from 0.05 to 0.95 and measured the fitness value of the elitist (out of 1,000 individuals) after 1,000 generations. From 0.05 to 0.40, the fitness value was improved rapidly but it did not significantly change after 0.45. This implies that our approach can produce appropriate clusters when the crossover rate is approximately 0.40. Note that this can be the minimum crossover rate. Certainly, a higher rate can lead to better solutions but could take more time.



(a) Crossover rate (b) Mutation rate

Fig. 10. Fitness values after 1,000 generations for each population size.

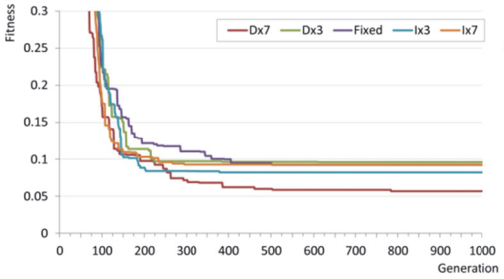


Fig. 11. Results of different settings used todynamically adjust the mutation rate.

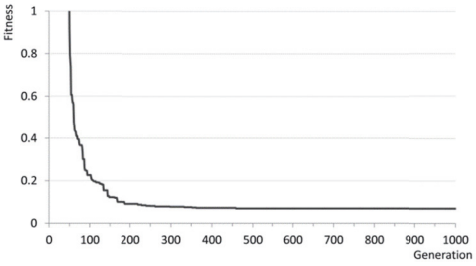


Fig. 12. Fitness value of each generation for appropriate parameters.

We measured the fitness value while varying the mutation rate from 0.003 to 0.1, as shown in Fig. 10 (b). Contrary to the crossover rate, the fitness value was not saturated. However, the fitness value showed the best performance when the mutation rate was 0.043. Using a higher rate does not result in better performance. Thus, users of our approach can use [0.03, 0.05] as the mutation rate to obtain an appropriate set of clusters.

When the crossover and mutation rates are 0.95 and 0.043, respectively, our approach shows its best performance.

In addition to the fixed mutation rate, we compared the performance of dynamic mutation rates as well, with results shown in Fig. 11. We tried five different settings: two decrement rates, two increment rates, and a fixed rate. For the dynamic decrement and increment rates, we used two different dynamic rates. For example, the mutation rate will be 0.116 if the dynamic rate is 1.001 and the initial rate is 0.043 after 1,000 generations. Since the final rate is approximately three times higher than the initial rate, we denoted this setting by **D×3**. Similarly, the final rate will be 0.317 if the dynamic rate is 1.002; this is denoted by **D×7**. We applied similar dynamic rates to the increment settings and denoted them by **I×3** and **I×7**, respectively. The fixed mutation rate (*i.e.*, 0.043) is denoted by Fixed.

Fig. 11 shows the results of applying different values to the dynamic mutation rate. For the increment settings (*i.e.*, **I×** and **I×**), their fitness values were quickly saturated. On the other hand, the decrement settings (*i.e.*, **D×3** and **D×7**) resulted in worse fitness values in early generations. However, they outperformed other settings in later generations. In particular, **D×** showed the best fitness value after 1,000 generations. This implies that gradually reducing the mutation rate can improve the performance of our approach.

Dynamically decreasing mutation rates can achieve better clusters.

After investigating appropriate parameters as described above, we applied our approach to the subjects explained in section 6.1: size of clusters = 25, population size = 1,000, maximum generation = 1,000, crossover rate = 0.95, initial mutation rate = 0.043, and dynamic mutation rate = **D×7** (= 0.998). Fig. 12 shows the fitness value of each generation (we repeated our approach 10 times and selected the best one). The final value of

fitness was 0.06936 and the experiment took 15 minutes (wall-time).

In addition, we visualized the results of bug report grouping and clusters after applying our approach. Fig. 13 shows a partial view of the results after grouping similar bug reports. The value of the grouping threshold was 0.86. White nodes represent independent reports that have no similar reports while red nodes are reports that have at least one similar report.

Fig. 14 represents the results of clustering using our approach. Each rectangle is a cluster of bug reports. The average of the total reading cost of each cluster was approximately 70. This indicates that our approach can uniformly and efficiently distribute bug reports to a specific number of developers.

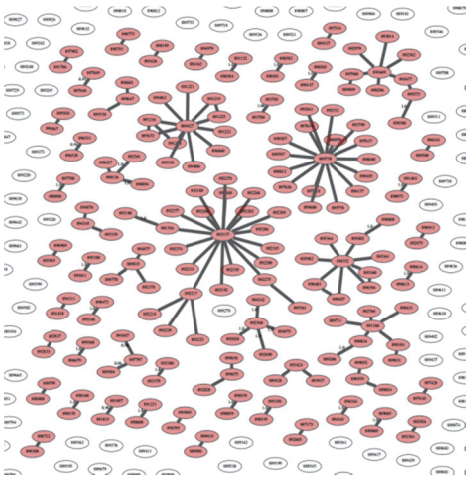


Fig. 13. Results of grouping for bug reports of Mozilla.

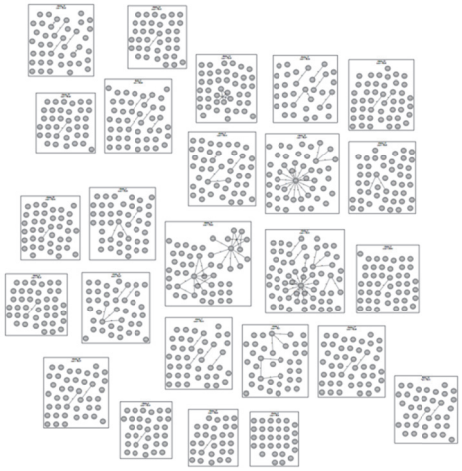


Fig. 14. Results of clustering for the bug reports of Mozilla.

6.2 User Study

This section presents the results of our user study. This study recruited 60 developers from 15 different companies. Among them, 30 developers responded. We asked them about their current practice of distributing bug reports and the qualitative evaluation results of our approach. The questions and distribution of the answers are listed in Table 3.

The analysis results are interpreted as follows: According to the responses to the first question (Q1), there are specific roles to distribute bug reports to developers. Only a small number of participants (13%) answered that developers perform this task. How identifying similar (or relevant) reports (Q2), some developers (20%) responded that they leverage systematic methods while most developers rely on manual identification.

For report distribution (Q3), more than half of the participants (53%) did not agree that their organizations uniformly distributed bug reports to developers. Although 33% of them stated that the current practice of distributing reports is likely to be uniform, the majority still answered that it is not sufficiently fair. 40% of participants stated that bug-fixing effort is the first criterion of report distribution (Q4). Reading cost is the second most frequent response.

Table 3. Five questions and answers for user study.

Q1: Who does distribute bug reports?		
Distributor (70%)	Developer (13%)	Other(17%)
Q2: How to identify similar bug reports?		
Manual identification by distributors (63%)		
Manual identification by developers (13%)		
Systematic identification (20%)		
Other (3%)		
Q3: Are bug reports uniformly distributed in the current practice?		
Strongly agree (10%)	Agree (23%)	
Neither agree nor disagree (53%)		
Disagree (10%)	Strongly disagree (3%)	
Q4: What are the criteria of uniform report distribution?		
Similar number of reports (3%)	Similar reading cost (27%)	
Similar bug fixing time (40%)	Other (30%)	
Q5: Is CBAC useful for bug report distribution?		
Strongly agree (10%)	Agree (57%)	
Neither agree nor disagree (20%)		
Disagree (13%)	Strongly disagree (0%)	

We then asked them about our approach. First, 67% of the participants stated that the results of CBAC were helpful for obtaining a balanced report distribution. Negative responses accounted for only 13%. In addition, the participants provided several suggestions: addressing bug-fixing cost [9] and considering developer expertise and authorship [16, 47].

> Developers stated that our approach can contribute to balanced report distribution.

6.3 Threats to Validity

- **Construct validity:** The reading cost of bug reports may not be strongly correlated with bug-fixing time in practice. However, contemporary estimation techniques [7, 8, 11] for bug-fixing effort highly depend on imprecise bug-fixing data because developers often do not accurately record their effort and time in resolving bugs. Document reading cost can provide at least a guide for determining estimated effort to understand bugs, and this is one of the major tasks in bug resolution.
- **Internal validity:** Our equation for reading cost estimation (Eq. 6) might not precisely represent the actual complexity of a document. Since bug reports have different characteristics compared to other types of documents such as technical manuals and legal statements, general-purpose methods for computing reading cost might not work.
- **External validity:** Our approach might show different performance depending on subjects or closed-source projects. Since our evaluation was performed only on Mozilla's Firefox project, applying CBAC to other projects may yield different results.

7. CONCLUSION

We presented an approach and tools aimed at distributing bug reports to developers

in a cost-effective manner, by considering their costs to provide workload balancing. First, we extracted bug reports from open source projects. Similar report groups that can be efficiently tackled by the same developer were then generated based on their similarity and dependency. Next, GA-based experiments to find a near-optimal set of clusters were conducted to provide a balanced distribution of bug reports. The standard deviation of the cluster costs was used as a fitness function in the optimization process.

The analyzed data from open source projects revealed that a large number of bug reports are concentrated on a small group of specific developers. However, overall bug-fixing time can be delayed if bug reports are distributed to developers in an imbalanced manner.

We also evaluated our approach by carrying out a survey targeting 30 developers from 15 companies. The results showed that 67% of the participants found our method helpful for triaging bugs to developers. The proposed approach, because it does not utilize the fixing history, can also provide balanced opportunity to both existing and new developers.

When a developer is organizing a development team, our approach can be effectively applied to assign a massive number of bugs to be fixed in a given limited time. From that point of view, our approach is also related to the optimized scheduling problem.

In summary of our contribution, we suggested a novel cost metric to comprehend a set of bugs by considering similarity and dependency of the bug reports. The bugs can be assigned to each developer in a balanced manner based on the metric. Thus, our approach can be effectively used in the development process, from the manager's perspective. Moreover, we used Kinciad's document readability [10] to calculate the reading cost of bug reports. This can be replaced with any other measure in the future because our method can be applied independently of the metric. And using our approach, the structure of the development team can also be generated from a set of bug reports, by recursively applying our method with a modified threshold or by using relations between the bugs to find larger clusters.

Recently, in the field of Granular Computing [48, 49], many researchers have been proposing new granular algorithms such as clustering [50, 51], classification [52, 53], rule-based algorithm [54, 55], and other various approaches [56-58]. In the future, we will improve the way to find optimal solutions by applying those algorithms.

REFERENCES

1. 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," in *Proceedings of the 10th Working Conference on Mining Software Repositories*, 2013, pp. 2-11.
2. C. R. Reis and R. P. Fortes, and M. Fortes, "An overview of the software engineering process and tools in the Mozilla project," in *Proceedings of Open Source Software Development Workshop*, 2002, pp. 155-175.
3. G. Jeong, S. Kim, and T. Zimmermann, "Improving bug triage with bug tossing graphs," in *Proceedings of the 7th Joint Meeting of European Software Engineering Conference and ACM Symposium on the Foundations of Software Engineering*, 2009, pp. 111-120.

4. D. Cubranic and G. C. Murphy, "Automatic bug triage using text categorization," in *Proceedings of the 16th International Conference on Software Engineering and Knowledge Engineering*, 2004, pp. 92-97.
5. J. Anvik, L. Hiew, and G. C. Murphy, "Who should fix this bug?" in *Proceedings of the 28th International Conference on Software Engineering*, 2006, pp. 361-370.
6. H. Naguib, N. Narayan, B. Brugge, and D. Helal, "Bug report assignee recommendation using activity profiles," in *Proceedings of the 10th Working Conference on Mining Software Repositories*, 2013, pp. 22-30.
7. A. Tamrawi, T. T. Nguyen, J. M. Al-Kofahi, and T. N. Nguyen, "Fuzzy set and cache-based approach for bug triaging," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, 2011, pp. 365-375.
8. J. W. Park, M. W. Lee, J. Kim, S. W. Hwang, and S. Kim, "COST RIAGE: A cost-aware triage algorithm for bug reporting systems," in *Proceedings of the 25th AAAI Conference on Artificial Intelligence*, 2011, pp. 139-144.
9. C. Weiss, R. Premraj, T. Zimmermann, and A. Zeller, "How long will it take to fix this bug?" in *Proceedings of the 4th International Workshop on Mining Software Repositories*, 2007, pp. 1-8.
10. J. P. Kincaid, L. R. P. Fishburne, R. L. Rogers, and B. S. Chissom, "Derivation of new readability formulas (Automated readability index, fog count, and flesch reading ease formula) for navy enlisted personnel," *Naval Technical Training: Naval Air Station Memphis*, 1975, pp. 8-75.
11. A. Zeller, *Why Programs Fail: A Guide to Systematic Debugging*, Morgan Kaufmann Publishers Inc., CA, 2005.
12. M. Fischer, M. Pinzger, and H. Gall, "Analyzing and relating bug report data for feature tracking," in *Proceedings of the 10th Working Conference on Reverse Engineering*, 2003, pp. 90-99.
13. D. Matter, A. Kuhn, and O. Nierstrasz, "Assigning bug reports using a vocabulary-based expertise model of developers," in *Proceedings of the 6th IEEE International Working Conference on Mining Software Repositories*, 2009, pp. 131-140.
14. P. Bhattacharya, I. Neamtiu, and C. R. Shelton, "Automated, highly accurate, bug assignment using machine learning and tossing graphs," *Journal of Systems and Software*, Vol. 85, 2012, pp. 2275-2292.
15. H. Kagdi, M. Gethers, D. Poshyvanyk, and M. Hammad, "Assigning change requests to software developers," *Journal of Software: Evolution and Process*, Vol. 24, 2012, pp. 3-33.
16. J. Xiao and W. Afzal, "Search-based resource scheduling for bug fixing tasks," in *Proceedings of the 2nd International Symposium on Search Based Software Engineering*, 2010, pp. 133-142.
17. N. Bettenburg, R. Premraj, and T. Zimmermann, "Duplicate bug reports considered harmful … really?" in *Proceedings of IEEE International Conference on Software Maintenance*, 2008, pp. 337-345.
18. 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*, 2007, pp. 499-510.
19. X. Wang, L. Zhang, T. Xie, J. Anvik, and J. Sun, "An approach to detecting dupli-

cate bug reports using natural language and execution information," in *Proceedings of the 13th International Conference on Software Engineering*, 2008, pp. 461-470.
20. C. Sun, D. Lo, X. Wang, J. Jiang, and S. C. Khoo, "A discriminative model approach for accurate duplicate bug report retrieval," in *Proceedings of the 32nd International Conference on Software Engineering*, 2010, pp. 45-54.
21. A. T. Nguyen, T. T. Nguyen, T. N. Nguyen, D. Lo, and C. Sun, "Duplicate bug report detection with a combination of information retrieval and topic modeling," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, 2012, pp. 70-79.
22. C. Sun, D. Lo, S. C. Khoo, and J. Jiang, "Towards more accurate retrieval of duplicate bug reports," in *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering*, 2011, pp. 253-262.
23. K. Somasundaram and G. C. Murphy, "Automatic categorization of bug reports using latent Dirichlet allocation," in *Proceedings of the 5th India Software Engineering Conference*, 2012, pp. 125-130.
24. T. Zhang and B. Lee, "A bug rule based technique with feedback for classifying bug reports," in *Proceedings of the 11th International Conference on Computer and Information Technology*, 2011, pp. 336-343.
25. Y. J. Horng, S. M. Chen, Y. C. Chang, and C. H. Lee, "A new method for fuzzy information retrieval based on fuzzy hierarchical clustering and fuzzy inference techniques," *IEEE Transactions on Fuzzy Systems*, Vol. 13, 2005, pp. 216-228.
26. N. Y. Wang and S. M. Chen, "Temperature prediction and TAIFEX forecasting based on automatic clustering techniques and two-factors high-order fuzzy time series," *Expert Systems with Applications*, Vol. 36, 2009, pp. 2143-2154.
27. S. M. Chen, N. Y. Wang, and J. S. Pan, "Forecasting enrollments using automatic clustering techniques and fuzzy logical relationships," *Expert Systems with Applications*, Vol. 36, 2009, pp. 11070-11076.
28. H. Szu and R. Hartley, "Fast simulated annealing," *Physics Letters A*, 1987, pp. 157-162.
29. M. Dorigo, M. Birattari, and T. Stutzle, "Ant colony optimization," *IEEE Computational Intelligence Magazine*, Vol. 1, 2006. pp. 28-39.
30. S. M. Chen and N. Y. Chung, "Forecasting enrollments using high-order fuzzy time series and genetic algorithms," *International Journal of Intelligent Systems*, Vol. 21, 2006, pp. 485-501.
31. Z. Michalewicz, *Genetic Algorithms + Data Structures = Evolution Programs*, 3rd ed., Springer-Verlag, London, UK, 1996.
32. S. M. Chen and T. H. Chang, "Finding multiple possible critical paths using fuzzy PERT," *IEEE Transactions on Systems*, *Man*, *and Cybernetics – Part B: Cybernetics*, Vol. 31, 2001, pp. 930-937.
33. S. M. Chen and C. Y. Chien, "Parallelized genetic colony systems for solving the traveling salesman problem," *Expert Systems with Applications*, Vol. 38, 2011, pp. 3873-3883.
34. P. W. Tsai, J. S. Pan, S. M. Chen, B. Y. Liao, and S. P. Hao, "Parallel cat swarm optimization," in *Proceedings of International Conference on Machine Learning and Cybernetics*, 2008, pp. 3328-3333.
35. P. W. Tsai, J. S. Pan, S. M. Chen, and B. Y. Liao, "Enhanced parallel cat swarm op-

timization based on the Taguchi method," *Expert Systems with Applications*, Vol. 39, 2012, pp. 6309-6319.
36. J. E. BellaPatrick and R. McMullen, "Ant colony optimization techniques for the vehicle routing problem," *Advanced Engineering Informatics*, Vol. 18, 2004, pp. 41-48.
37. D. Merkle, M. Middendorf, and H. Schmeck, "Ant colony optimization for resource-constrained project scheduling," *IEEE Transactions on Evolutionary Computation*, Vol. 6, 2002, pp. 333-346
38. R. Flesch, "A new readability yardstick," *Journal of Applied Psychology*, Vol. 32, 1948, pp. 221-233.
39. L. Si and J. Callan, "A statistical model for scientific readability," in *Proceedings of the 10th International Conference on Information and Knowledge Management*, 2001, pp. 574-576.
40. R. P. L. Buse and W. R. Weimer, "Learning a metric for code readability," *IEEE Transactions on Software Engineering*, Vol. 36, 2010, pp. 546-558.
41. G. Salton, A. Wong, and C. S. Yang, "A vector space model for automatic indexing," *Communications of the ACM*, Vol.18, 1975, pp. 613-620.
42. G. Salton and M. J. McGill, *Introduction to Modern Information Retrieval*, McGraw-Hill, Inc., NY, 1986.
43. J. B. Kruskal, "On the shortest spanning subtree of a graph and the traveling salesman problem," in *Proceedings of the American Mathematical Society*, Vol. 7, 1956, pp. 48-50.
44. D. J. Cavicchio, "Adaptive search using simulated evolution," Ph.D. Thesis, Department of Computer Science, University of Michigan, 1970.
45. A. Lipowski and D. Lipowska, "Roulette-wheel selection via stochastic acceptance," *Physica A: Statistical Mechanics and its Applications*, Vol. 391, 2012, pp. 2193-2196.
46. G. Syswerda, "Uniform crossover in genetic algorithms," in *Proceedings of the 3rd International Conference on Genetic Algorithms*, 1989, pp. 2-9.
47. F. Servant and J. A. Jones, "WhoseFault: Automatic developer-to-fault assignment through fault localization," in *Proceedings of the 34th International Conference on Software Engineering*, 2012, pp. 36-46.
48. Y. Yao, "A triarchic theory of granular computing," *Granular Computing*, Vol. 1, 2016, pp. 145-157.
49. D. Ciucci, "Orthopairs and granular computing," *Granular Computing*, Vol. 1, 2016, pp. 159-170.
50. G. Peters and R. Weber, "DCC: A framework for dynamic granular clustering," *Granular Computing*, Vol. 1, 2016, pp. 1-11.
51. P. Lingras, F. Haider, and M. Triff, "Granular meta-clustering based on hierarchical, network, and temporal connections," *Granular Computing*, Vol. 1, 2016, pp. 71-92.
52. M. Antonelli, P. Ducange, B. Lazzerini, and F. Marcelloni, "Multi-objective evolutionary design of granular rule-based classifiers," *Granular Computing*, Vol. 1, 2016, pp. 37-58.
53. H. Liu and M. Cocea, "Granular computing-based approach for classification towards reduction of bias in ensemble learning," *Granular Computing*, Vol. 2, 2016, pp. 131-139.
54. H. Liu, A. Gegov, and M. Cocea, "Rule-based systems: A granular computing perspective," *Granular Computing*, Vol. 1, 2016, pp. 259-274.

55. S. S. S. Ahmad and W. Pedrycz, "The development of granular rule-based systems: a study in structural model compression," *Granular Computing*, Vol. 2, 2017, pp. 1-12.
56. A. Skowron, A. Jankowski, and S. Dutta, "Interactive granular computing," *Granular Computing*, Vol. 1, 2016, pp. 95-113.
57. D. Dubois and H. Prade, "Bridging gaps between several forms of granular computing," *Granular Computing*, Vol. 1, 2016, pp. 115-126.
58. L. Livi and A. Sadeghian, "Granular computing, computational intelligence, and the analysis of non-geometric input spaces," *Granular Computing*, Vol. 1, 2016, pp. 13-20.

Jaekwon Lee (李在權) received the B.S. and M.S. degrees in Computer Engineering from Chungbuk National University, Korea, in 2013 and 2015, respectively. Currently, he is a Ph.D. student in the Department of Computer Engineering, Chungbuk National University. His research interests include mining software repositories and search-based software engineering (SBSE).

Dongsun Kim (金東鮮) received the B.E., M.S., and Ph.D. degrees in Computer Science and Eengineering from Sogang University, Seoul, Korea, in 2003, 2005, and 2010, respectively. He is currently a Research Associate at the University of Luxembourg. His research interests include mining software repositories, automatic patch generation, static analysis, search-based software engineering (SBSE).

Woosung Jung (鄭羽盛) received his B.S. and Ph.D. degrees in Computer Science and Engineering from Seoul National University, Korea, in 2003 and 2011, respectively. He was a Researcher in SK UBCare from 1998 to 2002. He was a Senior Research Engineer at Software Capability Development Center in LG Electronics from 2011 to 2012. He was an Associate Professor at the Department of Computer Engineering, Chungbuk National University from 2012 to 2016. He is currently an Associate Professor at the Graduate School of Education, Seoul National University of Education. His research interests include software education, software engineering, adaptive software system and data mining.