A Team Allocation Technique Ensuring Bug Assignment to Existing and New Developers Using Their Recency and Expertise

Afrina Khatun
Institute of Information Technology
University of Dhaka
Dhaka, Bangladesh
Email: bit0411@iit.du.ac.bd

Kazi Sakib
Institute of Information Technology
University of Dhaka
Dhaka, Bangladesh
Email: sakib@iit.du.ac.bd

***Abstract*—Existing techniques allocate a bug fixing team using only previous fixed bug reports. Therefore, these techniques may lead to inactive team member allocation as well as fail to include new developers in the suggested list. A Team Allocation approach for ensuring bug assignment to both Existing and New developers (TAEN) is proposed, which uses expertise and recent activities of developers. TAEN first applies Latent Dirichlet Allocation on previous bug reports to determine the possible bug types. For new developers, TAEN identifies their preferred bug type, and adds them to the list of other developers, grouped under the identified bug types. Upon the arrival of a new bug report, TAEN determines its type and extracts the corresponding group of developers. A heterogeneous network is constructed using previous reports to find the collaborations among the extracted developers. Next, for each developer, a TAEN score is computed combining the expertise and recency of their collaborations. Finally, based on the incoming report's severity, a team of *N* members is allocated using the assigned TAEN score and current workloads. A case study conducted on Eclipse Java Development Tools (JDT), shows that TAEN outperforms K-nearest-neighbor Search And heterogeneous Proximity based approach (KSAP) by improving the team allocation recall from 52.88 up to 68.51, and showing the first correct developer on average at position 1.98 in the suggested list. Besides, a lower standard deviation of workloads, 30.05 rather than 46.33 indicates balanced workload distribution by TAEN.**

***Keywords*—Bug Assignment; Team Allocation; Bug Report; Latent Dirichlet Allocation (LDA).**

I. INTRODUCTION

With the increasing size of software systems, bug assignment has become a crucial task for software quality assurance. For example studies reveal that, near the release dates, about 200 bugs were reported daily for Eclipse [1]. As developers generally work in parallel, this turns bug resolution into a collaborative task as well. It is reported that Eclipse bug reports involve on average a team of 10 developers contributions. However, due to large number of bug reports, manually identifying developer collaboration is error-prone and time-consuming. Besides, industrial projects have reported the need for collaborative task assignments to utilize both existing and new developers [2]. It is common that new developers join the company or project during software development. Random bug report assignment to new developers always results in unnecessary bug reassignments, and increases the time needed for the bug to be fixed. In this context, an automatic approach can facilitate bug assignment by allocating teams utilizing both existing and new developers.

In order to assign newly arrived bugs to appropriate developers, available information sources such as bug reports, source code and commit logs are analysed. Recent commits generally exhibit developer's recent activities and previous bug report represent their expertise on fixing particular types of bugs. Team assignment is generally done by analysing previously fixed bug reports, which can help to recommend experienced developers. With the passage of time, developers may switch projects or company, therefore inactive members may be recommended. On the other hand, developers who joined recently, do not own any fixed bug reports or commits. So, the approaches which learn from these information sources, fail to assign tasks to new developers. Existing developers get overloaded with a queue of bug reports, whereas new developers are ignored in the allocation procedure. This leads not only to prolonged bug fixing time, but also to improper workload distribution.

Understanding the importance of bug assignment, various techniques have been proposed in the literature. BugFixer, a developer allocation method has been proposed by Hao et. al [3]. This method constructs a Developer-Component-Bug (DCB) network using past bug reports, and recommends developers over the network. This allocated list becomes less accurate with the joining of new developers. Baysal et al. have proposed a bug triaging technique using the user preference of fixing certain bugs [4]. The technique combines developer's expertise and preference score for ultimate suggestion. However, this technique also considers only historical activities. Afrina et. al [5] have proposed an Expertise and Recency based Bug Assignment (ERBA) approach that considers both fixed reports and commit history for recommendation. This technique is applicable for single developer recommendation, and it cannot allocate tasks to new developers. A team assignment approach using K-nearest-neighbor Search And heterogeneous Proximity (KSAP) has been proposed by Zhang et al. [6]. It creates a heterogeneous network from the past bug reports, and assigns a team based on their collaboration over the network. The main limitation of this technique is that it over-prioritizes previous activities.

A Team Allocation technique for ensuring bug assignment to both Existing and New developers (TAEN), using expertise and recency of developers has been proposed. TAEN allocates a team in five steps. The *Bug Solving Preference Elicitation* step takes bug reports, and applies Latent Dirichlet Allocation (LDA) model on these reports to determine the possible types of bug reports. For new developers, TAEN first elicits their bug solving preference by presenting them with main representative terms of each bug type, and groups them under the corresponding type. The *New Bug Report Processing* step extracts the *summary, description* and *severity* of incoming reports, and determines their bug types to identify the potential fixer group. Next, the *Developer Collaboration Extraction* step generates a heterogeneous network using attributes (four types of nodes and eight types of edges) extracted from previous bug reports, and finds collaborations among the identified fixer group members over the network. The *Expertise and Recency Combination* step then assigns a TAEN score to each developer by combining the number and recency of their extracted collaboration. Finally, based on the *severity* of the incoming report, the *Team Allocation* step suggests a team of *N* developers using the TAEN score and current workloads. After each reported bug is fixed, this step also updates developers contribution status.

A case study on an open source project, Eclipse Java Development Tools (JDT) has been conducted for assessment of TAEN. To evaluate compatibility, TAEN has been compared with an existing technique, KSAP [6]. A total of 2500 *fixed* and 676 *open* bug reports have been taken under consideration [7]. A test set of 250 *fixed* and 30 *open* bug reports have been applied on both techniques. The results showed that TAEN improved the recall of the allocated team from 52.88 up to 68.51. A decrease in the average position of the first correct developer from 3.1 to 1.98 indicates the increased effectiveness of TAEN. Besides, a lower standard deviation (30.05 instead of 46.33) of developer workloads shows more balanced task distribution by TAEN.

The remainder of the paper is organized as follows. Section II describes the existing efforts in the field of automated bug assignment. Section III presents the overall team allocation procedure of TAEN by discussing the detailed processing of each step. Section IV shows a case study on Eclipse JDT while applying TAEN. Lastly, Section V concludes the paper by summarizing its contribution and possible future directions.

II. Related Work

Due to the increased importance of automatic bug assignment, a number of techniques have been proposed. A survey on various bug triaging techniques has been presented by Sawant et. al [8]. The survey divided bug triaging techniques into text categorization, reassignment, cost aware and source based techniques etc. Studies focusing on industrial needs of bug assignment have also been proposed in literature [2], [9]. Significant related works are outlined in this section.

Text categorization based techniques build a model that trains from past bug reports to predict the correct rank of developers [1], [3], [4], [10], [11]. Baysal et al. have enhanced these techniques by adding user preference in the recommendation process [4]. The framework performs its task using three components. The *Expertise Recommendation* component creates a ranked developer list using previous expertise profiles. The *Preference Elicitation* component collects and stores a rating score regarding the preference level of fixing certain bugs through a feedback process. Lastly, knowing the preference and expertise of each developer, the *Task Allocation* component assigns bug reports. The applicability of this technique depends on user ratings, which can be inconsistent. Besides, for recommendation the technique does not take new developers into account. As a result, imbalanced workload distribution among developers may occur.

Reassignment based techniques have also been developed by researchers [12], [13], [14]. The main focus of these techniques is to reduce the number of passes a bug report goes through due to incorrect assignment. In such techniques, a graph is constructed using previous bug reports [13], [14]. As mentioned above, consideration of these past activities fail to accommodate the new developers in final recommendation. A fine grained incremental learning and multi feature tossing graph based technique has been proposed by Bhattacharya et. al [12]. It is an improvement over previous techniques because it considers multiple bug report features, such as product and component, when constructing the graph. Because it considers previous information, the technique results in search failure in case of new developers arrival.

CosTriage, a cost aware developer ranking algorithm has been developed by Park et. al [15]. The technique converts bug triaging into an optimization problem of accuracy and cost, which adopts Content Boosted Collaborative Filtering (CBCF) for ranking developers. As the input to the system is only previous bug history, the technique contains no clue regarding new developers to assign tasks.

Source based bug assignment techniques have also been proposed. Matter et. al have suggested DEVELECT, a vocabulary based expertise model for recommending developers [11]. The model parses the source code and version history to index a bag of words representing the vocabulary of source code contributors. For new bug reports, the model checks the report keywords against developer vocabularies using lexical similarities. The highest scored developers are taken as fixers. Another source based technique has been proposed in [16]. The technique first parses all the source code entities (such as name of class, attributes, methods and method parameters) and connects these entities with contributors to construct a corpus. In case of new bug reports, the keywords are searched in the index and given a weight based on frequent usage and time metadata. The main limitation of these techniques is, it suggests novice developers without considering their experience and preference. As these techniques require minimum one source commits, these also fail to include new developers in final suggestion.

Vaclav et al. have presented a study to compare the trend of bug assignment in the open source and industrial fields [2]. The study applies Chi-Square and t-test for evaluating the variability of those two fields dataset, and reports identical trends in terms of distribution. Most importantly, it concludes with some findings highlighting the need for balanced task assignment to individuals and team recommendation. Zhang et al. developed a team assignment technique called KSAP [6]. It initially constructs a heterogeneous network using existing bug reports. When a new bug report arrives, the technique applies cosine similarity between the document vectors of new and existing bug reports, and extracts the K nearest similar bug reports. Next, the commenters of these K similar bugs are taken as the candidate list. Finally, the technique computes a proximity score for each developer based on their collaboration on the network. The top scored Q number of developers are recommended as fixer team. Although this technique can meet the need of team recommendation, it fails to cover the requirement of balanced task distribution due to ignoring new developers in the assignment process.

Various techniques for automatic bug fixer suggestion have been proposed in the literature. Most of the techniques learn from previous fix or source history of software repositories. Consideration of only one of these information sources leads to inactive or inexperienced developer recommendation. Again, both of the sources lack information regarding the newly joined developers. As a result, all of these techniques fail to delegate tasks to newly joined developers resulting in unequal workload distribution.

III. Methodology

In order to allocate teams by ensuring task allocation to both existing and new developers, a technique called TAEN is proposed. Most of the existing techniques learn from previous fixed reports for recommending expert developers. Due to ignoring recent activities, these approaches may suggest inactive developers. Using only expertise information cannot satisfy the required information provided by the source contributions. Both information sources need to be considered to allocate expert and recent group of developers. Therefore, an expert and recent team allocator capable of allocating tasks to both existing and new developers is required. TAEN allocates a team in five steps which are described below.

A. Bug Solving Preference Elicitation of New Developers

As bug tracking and version control systems do not contain any record regarding the activities of new developers, existing approaches fail to recommend new developers. In this case, bugs are assigned randomly to these developers regardless of their abilities and preferences in solving the bugs, which always results in reassignment and prolonged fixing time. This step determines the bug solving preference of new developers in two phases - *Developer Group Creation* and *Preference Elicitation*.

1) Developer Group Creation: This phase groups developers based on the types of bugs they have worked on. In this context, first, the possible types of bugs needs to be determined. Therefore, this step takes bug reports as input in Extensible Markup Language (XML) format. A bug report generally contains a number of attributes such as *id, status, resolution, fixer, commenter, severity, summary, description, activity history* etc. For training and evaluation purpose, the bug reports which have *resolved* and *verified* as status, and *fixed* as resolution property are taken into consideration. Besides, in order to determine developers current workloads, the bug reports which have bug status either of *new*, *reopened* and *started* are collected.

Next, the *summary* and *description* property of each report are extracted and processed to represent its vocabulary. The processing steps are discussed in Subsection III-C. For identifying the type of bug reports, LDA modeling is used. Given a list of documents having mixtures of (latent) topics, LDA tends to determine the most relevant topic of the document. So, the bug reports are represented as documents, and fed into the LDA model to be divided into *n* types. At the end, the LDA model determines the most relevant type for each bug report. Each bug type is represented with the probabilities of each word to be in the type.

Once all the bug reports are labeled with one of the *n* types, the developers who have worked on similar types of bugs are grouped together. Hence, the algorithm in Figure 1 is proposed for creating developer groups. The *GroupDevelopers* procedure of Figure 1 takes the processed bug reports as input. This procedure keeps the grouped developers in a complex data structure called *bugTypes*, as shown in line 2. The outer map of *bugTypes* links each type to developers who have contributed to that specific type of bugs. The inner map connects each developers name to their contribution frequency on that type of bugs.

A *for* loop is defined at line 4 for iterating on the inputted bug reports. Each iteration of the loop first extracts the bug report's type determined by the LDA model. This task is done by calling a method, *GetBugType*, as shown in line 5. The method takes the *summary* and *description* of the report, and returns its *type*. The *GroupDevelopers* procedure also extracts and stores the contributor's name of each bug report in a *Set* of strings named *contributors*. Here, the contributors refers to the reporter and fixers of the bug report.

```
1:  procedure GROUPDEVELOPERS($List < BugReport >$
        $BugReports$)
2:      $Map<String, Map<String, Integer> > bugTypes$
3:      $Map<String, Integer> developers, String\ type$
4:      for each $b \in BugReports$ do
5:          $type \leftarrow$ GETBUGTYPE($b.summary, b.description$)
6:          $Set < String > contributors \leftarrow b.contributors$
7:          $developers \leftarrow bugTypes.get(type)$
8:          if $developers == null$ then
9:              $developers \leftarrow new\ Map<String, Integer>()$
10:             for each $c \in contributors$ do
11:                 $developers.put(c, 1)$
12:             $bugTypes.put(type, developers)$
13:         else
14:             for each $c \in contributors$ do
15:                 if $devlopers.contains(c)$ then
16:                     $developers.replace(c, developers[c]{+}1)$
17:                 else
18:                     $developers.put(c, 1)$
19:             $bugTypes.replace(type, developers)$
```

Figure 1: The Algorithm of Developer Group Creation

Next, the procedure gets the list of developers mapped against the identified bug *type* (line 7). If no developers are yet mapped against this *type*, a new instance of inner map named *developers* is initialized (line 8-9). All the *contributors* are then populated into the *developers* map which links each developer to their initial contribution frequency (line 10-11). This *developers* map is then put against the identified bug *type* (line 12). On the other hand, if a list of *developers* is already mapped against the identified *type*, another *for* loop is declared for updating the *developers* list (line 14). The loop then checks whether the *developers* list already contains the *contributors* and updates the contribution frequency of each contributor, *c* (line 15-18). Finally, the procedure updates the outer map *bugType* with the changed *developers* list (line 19).

2) Preference Elicitation: This step focuses to elicit the bug solving preference of new developers for ensuring their inclusion in the allocated team. When a new developer arrives, the list of most representative words of each bug type is offered to the developer. The chosen bug types are initially considered as the types of bugs the developer can contribute to. So, the developer is then grouped with the developers who have worked on similar bugs determined by the previous step.

B. New Bug Report Processing

On arrival of a new bug report *B*, the type of the report needs to be identified for extracting the developer group to which it can be assigned. The *summary, description* and *severity* properties of the report are extracted and processed. As these property values generally contain irrelevant and noisy terms, pre-processing is done. The processing step includes identifier decomposition based on CamelCase letter and separator character, number and special character removal, stop word removal and stemming. A score for each bug type is computed using (1) similar to [15], as follows-

$$typeScore(i) = \sum_{\forall w \in B} (Probability_i(w) * Distribution_i(w)) \quad (1)$$

where, *i* represents the *i-th* type in the LDA model, *w* represents each word in *B*, *Probability$_i$(w)* is the probability of *w* in the *i-th* bug type, and the distribution of *w* in the new bug report is indicated by *Distribution$_i$(w)*. Finally, the bug type which gets the highest score having most similar vocabulary with the new bug report, is determined as the type of the new bug report. Thus, the developer's group associated with the determined bug type is selected. The top-*K* members of this group, who have higher contributions are considered as the developers from which a bug fixing team needs to be allocated.

C. Developer Collaboration Extraction

It is mentioned above that bug resolution is a collaborative task. To allocate a team, the collaboration among the developers needs to be considered. So, this step extracts the collaboration among the developers of the identified group. A heterogeneous directed network is constructed from the previous *fixed* bug reports [6]. The four types of nodes include - Bug (B), Developer (D), Component (C) and Comment (T). The eight types of possible relations among these nodes are listed in Table I. For example, Type 1 relationship connects a D node to a B node depicting the developer (D) has *worked on* the bug report (B). The term *work* refers assignment, report, reassignment, reopening, fixing, verifying or tossing event of a bug. Similarly, Type 4 and Type 5 relations denote that a comment (T) *is contained by* a bug (B), and a developer (D) *has written* the comment (T), respectively.

Developer collaboration can be identified by factors such as how frequently two developers contribute to the same bugs and components of the system. Keeping these factors in mind, six types of paths similar to [6] are extracted from the network each of which connects two developers using combinations of the above relation edges. The paths are listed in Table II. For example - 'D-B-T-D' represents that a developer (D) has worked on a bug (B), which has a comment (T) written by another developer (D). Similarly, 'D-B-C-B-D' depicts that a developer (D) has worked on a bug (B) of a component (C), having another bug (B), which was worked on by another developer (D).

TABLE I. EIGHT TYPES OF RELATIONSHIPS AMONG NODES

Type No.	Specification
1	D *works on* B
2	B *is worked on by* D
3	B *contains* T
4	T *is contained by* B
5	D *writes* T
6	T *is written by* D
7	B *contains in* C
8	C *is contained by* B

TABLE II. SIX TYPES OF DEVELOPER COLLABORATION

Path Type	Collaboration on	Path
1	Same Bug	D-B-D
2	Same Bug	D-B-T-D
3	Same Bug	D-T-B-T-D
4	Same Component	D-B-C-B-D
5	Same Component	D-B-C-B-T-D
6	Same Component	D-T-B-C-B-T-D

D. Expertise and Recency Combination

As mentioned before, ignorance of recent activities may result in inactive developer assignment. So, this step adds recency information with the extracted developer's collaboration. The more recent developers work or comment on a bug, the higher the priority of that developer. For combining the recent activities, the time when the developers collaborate on the bug, is considered. The algorithm in Figure 2 is proposed to compute a score called TAEN score for each developer, by combining the expertise and recency of collaboration.

The *CalculateScore* procedure of Figure 2 takes a complex data structure, named *devInfos* as input. This data structure maps the developers to their identified collaboration information of type *DeveloperCollaboration.*

```
procedure CALCULATESCORE($Map < String,$
        $DeveloperCollaboration > devInfos$)
    $Map < String, Double > devScores$
    for each $d \in devInfos$ do
        for each $path \in d.sameBugs$ do
            ADDSCORE($path.firstEdge,$
                            $BugReport.date$)
            ADDSCORE($path.lastEdge,$
                            $BugReport.date$)
procedure ADDSCORE($Edge\ e, Date\ date$)
    if $e.srcNode = D$ then
        $dev \leftarrow e.srcNode$
    else
        $dev \leftarrow e.destNode$
    if $!devScores.keys.contains(dev)$ then
        $devScores \leftarrow 1/(date - e.Date)$
    else
        $devScores+ \leftarrow 1/(date - e.Date)$
```

Figure 2: The Algorithm of Expertise and Recency Combination

Each instance of *DeveloperCollaboration* contains two properties - *sameBugs* and *sameComponents*. The former property contains a list of paths which depicts the associated developers collaboration on same bugs. Similarly, the later one represents developers collaboration on same components. The *CalculateScore* procedure represents the partial score calculation process based on the same bugs only. A similar approach is also used for calculating the collaboration score of same components. The procedure starts with defining a data structure called, *devScores* which connects the developers to their calculated TAEN score (line 2). An outer *for* loop is defined for iterating on each developer and an inner loop is defined for iterating on their collaborated paths (line 3-4). Each collaboration path generally connects two developers. Therefore, for each collaborated path the score of the two developers needs to be added or updated (line 5,6). To perform this task, another procedure, *AddScore* is declared (line 7).

This procedure takes an edge and a date as input. It is seen from Table I that developers directly collaborate by working or commenting on bugs. So, the collaboration edge is sent as parameter for the *AddScore* function. Besides, for adding the recency information of these activities, the collaboration date is also sent to this function. It first extracts the developer node from the inputted edge (line 8-11). It then checks whether the developer has a TAEN score already assigned (line 12). Based on this checking, it adds or updates the score (line 13-15). The score for each collaboration path is initially considered as 1. However, this score is divided by the date difference between the reporting date of the new bug report (*date*) and the collaboration date of the developer (*e.date*). The smaller the difference, the more recent the developer collaborated on the bug, thus the higher the score the developer gets. Lines 13-15 ensure the effect of recency information on the developer's TAEN score.

E. Team Allocation

Finally, for allocating a team consisting of *N* developers where *N*<*K*, the technique first checks the *severity* property of the newly arrived bug report. The *severity* property refers to how severe the bug is, or whether it is an enhancement request. If the *severity* value is any of *blocker*, *critical* and *major* [7], the reported bug is considered as one that needs to be handled by existing developers. So, TAEN considers only the top-*K* contributors and sorts the developers based on their TAEN score. The top scored *N* developers are allocated as the fixer team.

If the *severity* value contains *normal*, *minor*, *trivial* or *enhancement* [7], it can be handled by new developers. In this case, TAEN considers the new developers along with the top-*K*, and counts their current workload (assigned bugs). The *N* developers with least workload are included in the team. This step ensures bug assignment to new developers based on their bug solving preference. If two developers have the same workloads, the tie is resolved using the TAEN score. When a bug report is fixed by a developer, this contribution is updated in the groups of Subsection III-A. This update ensures incremental contribution enhancement of developers as well as their participation in bug resolution.

IV. Case Study

For initial assessment of compatibility, TAEN was applied on an open source project, Eclipse JDT [17]. This project was chosen because this has been used for evaluation in various related approaches [15]. Secondly, the bug repository of JDT is available in open source. A total of 2500 *fixed* bug reports between years 2009 and 2015, and 676 *open* bug reports between 2015 and 2016 have been collected for experimental analysis of TAEN.

As stated before, TAEN first takes system bug reports in XML format [7]. The *summary* and *description* properties are collected from the <*short_desc*> and <*thetext*> tags respectively. It then applies LDA on the properties to determine the most relevant type of each bug report. Various techniques are available in the literature for identifying the natural number of topics when applying LDA [15]. The case study divides the bug reports into *n*=17 distinct types, as 17 has already been used as number of topics in Eclipse [15]. The contributors, severity, reporting time, activity properties are also extracted from different XML tags in a similar manner. The contributors are then grouped against the corresponding bug types identified by LDA.

Now, on arrival of new developers, they are presented with the most representative words of each bug type. Table III shows a few top most representative words of bug Type-2 and 11. The table depicts that the representative keywords give an idea about the corresponding type. For example, the enlisted keywords against Type-2 indicates User Interface (UI) related terms as well as bugs. If a developer selects Type-2, the developer is added to the group of developers associated with Type-2 bugs.

TABLE III. Few top representative words of bug Type-2 and 11

Type 2	Click	Editor	Select	Display	Dialog	Event
Type 11	Mozilla	Agent	Gecko	Build	User	Windows

For comparative analysis, TAEN is compared with a team assignment approach, KSAP [5]. A randomly selected test dataset containing 250 *fixed* and 30 *open* bug reports have been used for checking the allocation validity. The experimental analysis allocates a team of *N* developers, where *N* is set to 10. The reason behind setting *N* to 10 is that it is reported that Eclipse bug reports include on average 10 developers contributions [6]. Besides, for ensuring validation consistency between KSAP and TAEN, *K*=50 top most contributors are taken from both techniques for processing of *Developer Collaboration Extraction* step.

The compatibility of TAEN is evaluated using the following metrics - recall, effectiveness and workload distribution. Recall@N refers to whether the top N allocated developers contain the actual developers who fixed the report. The higher number of actual developers included in the top N places, the more correct the allocation is. The recall is calculated using (2) similar to [6] -

$$Recall@N = \frac{|\{dev1, dev2, ..., devN\} \cap \{GroundTruth\}|}{|GroundTruth|} \tag{2}$$

Here, *{dev1,dev2,...,devN}* is the set of *N* allocated developers, and *{GroundTruth}* refers to the set of actual fixers containing the reporter, fixer and commenters. Table IV illustrates the average Recall@10 achieved by TAEN and KSAP. TAEN had a higher average recall (68.51) than KSAP. Consideration of both recent and previous activities enabled TAEN to improve the recall from 52.88 to 68.51.

TABLE IV. COMPARISON OF AVERAGE TEAM ALLOCATION RECALL@10

Approaches	Average Recall@10
KSAP	52.88
TAEN	68.51

Effectiveness refers to the position of the first *GroundTruth* developer in the allocated list. Not all the members of a team generally play similar roles in an assigned task. Therefore, the ranking in the suggested team plays a vital role in determining task division. Approaches that allocate relevant developers at the top of the list are considered more effective. A lower value of this metric indicates higher effectiveness of the allocated list. The values in Table V shows allocation effectiveness of TAEN and KSAP. They also show the percentage of suggesting the first relevant developer at Position 1 to 3. In 66.36% cases TAEN shows the first revelant developer at Position 1 whereas KSAP shows that in only 2.73% cases. The consideration of recent activities enables TAEN to prioritize active developers at top of the list. The percentage of Position 2 and 3 for TAEN is less than KSAP because, TAEN covers most of the cases at Position 1. The last column shows TAEN shows the relevant developers on average near position 1.98 which is lower than KSAP (3.1).

TABLE V. COMPARISON OF AVERAGE EFFECTIVENESS

Approaches	Average No. of Cases (%)			Average Effectiveness
	Position 1	Position 2	Position 3	
KSAP	66.36	7.27	10.91	1.98
TAEN	2.73	57.27	11.82	3.1

In order to validate the task assignment to new developers, current workload among the developers are counted from *open* bug reports of 2016. The developers who do not have any past history, i.e. they are not grouped under any of the bug types are considered as new developers in the experimentation. The bug solving preference of these new developers is determined by the type of bugs they are currently assigned to. Based on this preference, these developers are initially grouped with one of the bug types. The 30 above mentioned *open* bug reports are analyzed on both KSAP and TAEN. A partial view of workload distribution among developers preferring bug Type-15 is shown in Figure 3. The bars of Figure 3 clearly show that KSAP assigns no tasks to the 6 new developers plotted at the right end of the graph. However, TAEN successfully allocates the new developers based on their preference.

For better understanding the variability of task assignment, standard deviation of the workloads is calculated. Standard deviation of a dataset depicts the variability of the data from their mean point. A lower value of this metric represents less variability i.e. equal workload distribution among developers. The average standard deviation of workloads assigned by the two techniques are enlisted in Table VI. TAEN has a

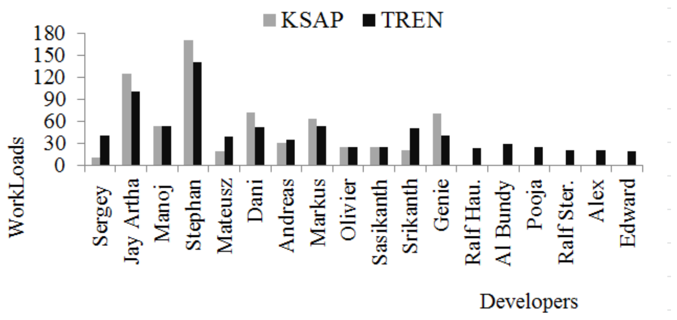

Figure 3: Workload Distribution of KSAP and TAEN

lower standard deviation of 30.05 than KSAP (46.33). The preference based inclusion of new developers in the assignment process, enabled TAEN to achieve lower standard deviation. This significant decrease in the value of standard deviation represents higher consistency of resource utilization by TAEN.

TABLE VI. COMPARISON OF VARIABILITY IN WORKLOAD DISTRIBUTION

Approaches	Average Standard Deviation
KSAP	46.33
TAEN	30.05

V. CONCLUSION

Team allocation is generally done from previous fixed reports. Due to ignoring recent activities, these approaches may allocate inactive fixers. Both previous reports and recent commits do not contain any information regarding the newly joined developers. Not considering new developers in the final allocation leads to improper workload distribution. To overcome these limitations, TAEN is proposed, which assigns bugs to both existing and new developers combining the expertise and recency information.

The *Bug Solving Preference Elicitation* step first determines new developer's choice of fixing certain types of bugs, and adds them to the group of developers of the chosen type. The *New Bug Report Processing* step identifies the type of the incoming reports to extract the corresponding grouped developers. Next, the *Developer Collaboration Extraction* step generates a heterogeneous network from the previous reports to find the collaboration of the extracted developers over the network. The *Expertise and Recency Combination* step then assigns a TAEN score to each developer based on their collaboration expertise and recency. After checking the severity of incoming reports, the *Team Allocation* step allocates a fixer team by using TAEN score and current workloads.

For performing a case study on Eclipse JDT, 2500 *fixed* and 676 *open* bug reports were collected. A test set of 250 *fixed* and 30 *open* bug reports were used for comparison with an existing technique, KSAP. The result shows that TAEN improves recall from 52.88 to 68.51, and achieves increased effectiveness by identifying the correct bug fixer near position 1.98. The results also depict a significant decrease of standard deviation from

46.33 to 30.05 which indicates equal workload distribution. In future, bug reports which are not previously handled by any developers should be observed to check TAEN's performance.

ACKNOWLEDGMENT

This work is supported by the fellowship from ICT Division, Bangladesh. No - 56.00.0000.028.33.065.16 (Part-1)-772 Date 21-06-2016.

REFERENCES

[1] S. Banitaan and M. Alenezi, "Tram: An approach for assigning bug reports using their metadata," in *Proceedings of the 3rd International Conference on Communications and Information Technology (ICCIT), June 19–21, 2013, Beirut, Lebanon*. IEEE, 2013, pp. 215–219, URL: http://info.psu.edu.sa/psu/cis/malenezi/pdfs/TRAM.pdf [accessed: 2016-10-30].

[2] V. Dedík and B. Rossi, "Automated bug triaging in an industrial context," in *Proceedings of the 42th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), August 31–September 2, 2016, Limassol, Cyprus*. IEEE, 2016, pp. 363–367, URL: https://www.researchgate.net/profile/Bruno_Rossi2/publication/308417176_Automated_Bug_Triaging_in_an_Industrial_Context/links/57e3e3df08ae8d5908c1617b.pdf [accessed: 2016-12-01].

[3] H. Hu, H. Zhang, J. Xuan, and W. Sun, "Effective bug triage based on historical bug-fix information," in *Proceedings of the 25th International Symposium on Software Reliability Engineering (ISSRE), October 23–26, 2014, Toulouse, France*. IEEE, 2014, pp. 122–132, URL: https://hal.inria.fr/hal-01087444/document [accessed: 2016-05-20].

[4] O. Baysal, M. W. Godfrey, and R. Cohen, "A bug you like: A framework for automated assignment of bugs," in *Proceedings of the 17th International Conference on Program Comprehension (ICPC), May 17–19, 2009, British Columbia, Canada*. IEEE, 2009, pp. 297–298, URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.143.7283&rep=rep1&type=pdf [accessed: 2016-10-29].

[5] A. Khatun and K. Sakib, "A bug assignment technique based on bug fixing expertise and source commit recency of developers," in *Proceedings of the 19th International Conference on Computer and Information Technology (ICCIT), December 18–20, 2016, Dhaka, Bangladesh*. IEEE, 2016, pp. 592–597, URL: http://sci-hub.cc/10.1109/iccitechn.2016.7860265 [accessed: 2017-03-05].

[6] W. Zhang, S. Wang, and Q. Wang, "Ksap: An approach to bug report assignment using knn search and heterogeneous proximity," *Information and Software Technology*, vol. 70, pp. 68–84, 2016, ISSN: 0950-5849.

[7] "Afrina/TREN," Jan. 2017, URL: https://github.com/Afrina/TREN/blob/master/TeamAssignMSTestProject/Data/TeamData/bug_data_2009_2015.xml [accessed: 2017-01-10].

[8] V. B. Sawant and N. V. Alone, "A survey on various techniques for bug triage," *International Research Journal of Engineering and Technology*, vol. 2, pp. 917–920, 2015, ISSN: 2395-0056.

[9] R. V. Sangle and R. D. Gawali, "Auto bug triage a need of software industry," *International Journal of Engineering Science*, vol. 6, pp. 8668–8670, 2016, ISSN: 2321-3361.

[10] M. Linares-Vásquez, K. Hossen, H. Dang, H. Kagdi, M. Gethers, and D. Poshyvanyk, "Triaging incoming change requests: Bug or commit history, or code authorship?" in *Proceedings of the 28th International Conference on Software Maintenance (ICSM), September 23–30, 2012, Trento, Italy*. IEEE, 2012, pp. 451–460, URL: http://www.cs.wm.edu/~mlinarev/pubs/ICSM'12-DevRecAuthorship.pdf [accessed: 2016-10-20].

[11] D. Matter, A. Kuhn, and O. Nierstrasz, "Assigning bug reports using a vocabulary-based expertise model of developers," in *Proceedings of the 6th International Working Conference on Mining Software Repositories (MSR), May 16-17, 2009, Vancouver, Canada*. IEEE, 2009, pp. 131–140, URL: http://flosshub.org/system/files/131AssigningBugReports.pdf [accessed: 2016-02-26].

[12] P. Bhattacharya and I. Neamtiu, "Fine-grained incremental learning and multi-feature tossing graphs to improve bug triaging," in *Proceedings of the 26th International Conference on Software Maintenance (ICSM), September 12–18, 2010, Timisoara, Romania*. IEEE, 2010, pp. 1–10, URL: http://www.cs.ucr.edu/~pamelab/icsm10bhattacharya.pdf [accessed: 2016-02-26].

[13] G. Jeong, S. Kim, and T. Zimmermann, "Improving bug triage with bug tossing graphs," in *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering (ESEC/FSE), August 24–28, 2009, Amsterdam, Netherlands*. ACM, 2009, pp. 111–120, URL: http://143.89.40.4/~hunkim/images/6/65/Papers_jeong2009fse.pdf [accessed: 2016-05-30].

[14] L. Chen, X. Wang, and C. Liu, "An approach to improving bug assignment with bug tossing graphs and bug similarities," *Journal of Software*, vol. 6, pp. 421–427, 2011, ISSN: 1796-217X.

[15] J.-W. Park, M.-W. Lee, J. Kim, S.-W. Hwang, and S. Kim, "Cost-aware triage ranking algorithms for bug reporting systems," *Knowledge and Information Systems*, vol. 48, pp. 679âĂŞ–705, 2015, ISSN: 0219-3116.

[16] R. Shokripour, J. Anvik, Z. M. Kasirun, and S. Zamani, "A time-based approach to automatic bug report assignment," *Journal of Systems and Software*, vol. 102, pp. 109–122, 2015, ISSN: 0164-1212.

[17] "JDT Core Component - Eclipse," Jan. 2017, URL: https://eclipse.org/jdt/core/ [accessed: 2017-01-10].