A Statement Level Bug Localization Technique using Statement Dependency Graph

Shanto Rahman, Md. Mostafijur Rahman and Kazi Sakib
Institute of Information Technology, University of Dhaka, 1205, Bangladesh

Keywords: Statement Level Bug Localization, Search Space Minimization, Statement Dependency, Similarity Measurement.

Abstract: Existing bug localization techniques suggest source code methods or classes as buggy which require manual investigations to find the buggy statements. Considering that issue, this paper proposes Statement level Bug Localization (SBL), which can effectively identify buggy statements from the source code. In SBL, relevant buggy methods are ranked using dynamic analysis followed by static analysis of the source code. For each ranked buggy method, a Method Statement Dependency Graph (MSDG) is constructed where each statement acts as a node of the graph. Since each of the statements contains few information, it is maximized by combining the contents of each node and its predecessor nodes in MSDG, resulting a Node Predecessor-node Dependency Graph (NPDG). To identify relevant statements for a bug, similarity is measured between the bug report and each node of the NPDG using Vector Space Model (VSM). Finally, the buggy statements are ranked based on the similarity scores. Rigorous experiments on three open source projects named as Eclipse, SWT and PasswordProtector show that SBL localizes the buggy statements with reasonable accuracies.

1 INTRODUCTION

In automatic software bug localization, finding bugs in granular levels i.e., statement of the source code is needed because it reduces maintenance effort. On top of this, many bug localization techniques have already been proposed which ranked buggy classes (Zhou et al., 2012) or methods (Poshyvanyk et al., 2007). The suggestion of buggy classes provide a large problematic solution search space where a bug can stay. Among the suggested class list, it is almost impossible to find buggy statements because a class may contain numerous statements. Although suggesting buggy methods are better than suggesting classes, this hardly reduces the maintenance effort because a method may also have large number of statements.

Although statement level bug localization is required, there may exist several limitations such as the availability of large irrelevant information within a project for a bug and a small valid information within a statement. A software project often contains a large number of statements having massive amount of irrelevant information for a bug. For example, Bug Id- 31779 of Eclipse is related to *src.org.eclipse.core.internal.localstore.UnifiedTree.java*, whereas the total number of files in Eclipse is 86206 and except the aforementioned buggy file, other files are irrelevant for this bug. As bug localization using bug report follows probabilistic approach, the consideration of irrelevant information can mislead to rank the buggy statements. Therefore, it is needed to discard the irrelevant source code as much as possible. Meanwhile, a single statement contains very few information about a bug. For example, for Eclipse Bug Id- 31779, line 122 (i.e., $child = createChildNodeFromFileSystem(node, parentLocal Location, localName)$;) is one of the buggy statements which contains few information about the bug. Only using this information, suggesting buggy statement is another challenging issue.

Fault localization is closely related to the bug localization, however the main difference is, fault localization does not consider bug report whereas bug localization does (Zhou et al., 2012). In real life projects, user or Quality Assurance (QA) team reports against a faulty scenario, and the bug is fixed using that report. Although several researches address bug localization, to the best of the authors knowledge, still no research has been conducted to suggest buggy statements using bug report. Zhou et al. propose a class level bug localization technique by considering whole source code as a search space. As a result,

Rahman, S., Rahman, M. and Sakib, K.
A Statement Level Bug Localization Technique using Statement Dependency Graph.
DOI: 10.5220/0006261901710178
In *Proceedings of the 12th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2017)*, pages 171-178
ISBN: 978-989-758-250-9

biasness may be introduced. As this technique suggests classes, it demands manual inspection into the source code files to find buggy statements. Considering this issue, comparatively more granular level i.e., method level suggestion is addressed. Poshyvanyk et al. propose PROMISER which suggests methods as buggy (Poshyvanyk et al., 2007). Authors consider whole source code which may degrade the accuracy of bug localization. To improve the accuracy, recently Rahman et al. introduce MBuM where irrelevant source code for a bug is discarded (Rahman and Sakib, 2016). Still, developers need manual investigations to find the buggy statements. Interestingly, the minimized search space which is extracted in MBuM can be used for reaching further granular level.

In this paper, Statement level Bug Localization (SBL) is proposed where statements are suggested as buggy. At first, SBL extracts source code methods, generated from (Rahman et al., 2016). As the statements of the methods' are related to each other, for each suggested method, a dependency relationship is developed among the statements, which is named as Method Statement Dependency Graph (MSDG). The information of each node is processed to produce corpora. Later, a Node Predecessor-node Dependency Graph (NPDG) is developed where the corpora of each node (i.e., statements of the source code) and its predecessor nodes in MSDG are combined. This is because a statement often contains few information about a bug. The combination of each node with its predecessors increases the valid information of each statement. The similarity between the bug report and each node of NPDG is measured using VSM. These statement similarity scores are weighted by the corresponding method similarity score. Finally, a list of buggy statements are suggested using the descending order of the similarity scores.

The effectiveness of SBL has been measured using 230 bugs from three open source projects namely Eclipse, SWT and PasswordProtector where Top N Rank, Mean Reciprocal Rank (MRR) and Mean Average Precision (MAP) are used as the evaluation metrics. In all the projects, SBL ranks more than 28.33% buggy statements at the top 1 and more than 58.33% within top 5. In case of MAP, 38.5%, 44.2% and 61% accuracies are gained for Eclipse, SWT and PasswordProtector respectively. For MRR, 45.8%, 51.3% and 66% accuracies have been achieved in the aforementioned projects respectively.

2 LITERATURE REVIEW

Since statement level bug localization is new, this section focuses on researches that are conducted to suggest buggy classes or methods.

2.1 Class Level Bug Localization

Zhou et al. propose BugLocator where class level buggy locations are suggested (Zhou et al., 2012). Here, two sets of corpora have been generated; one for bug report and another for source code using several text-processing approaches such as stop word removal, multi-word identifiers and stemming. Similarity is measured between these two sets of corpora by revised Vector Space Model (rVSM). This technique considers whole source code for static analysis which may degrade the accuracy. An improved version is addressed by Ripon et al. where special weights are assigned on structural information including class names, method names, variable names and comments of the source code (Saha et al., 2013). Due to considering whole source code for a bug, the accuracy of the technique may get biased.

Another class level bug localization technique is proposed by Tantithamthavorn et al. (Tantithamthavorn et al., 2013). Here, an intrinsic assumption is that when a bug is fixed, a set of classes are changed altogether. On top of this, co-change score is calculated and a list of co-change files are identified. Large number of changes holds high score for those files to be buggy. After finding the co-change score, these results are adjusted with BugLocator. As this technique suggests classes as buggy, manual searching is needed to find the actual buggy statements from the source code which consumes lots of time.

Sisman and Kak incorporates version histories (Sisman and Kak, 2012) to suggest buggy locations. Similar to this, Wang et al. introduce a technique by considering similar bug reports, version history and the structure of the source code (Wang and Lo, 2014). This technique also suggests class level buggy locations and so, developers have to spend time to find buggy statements. Based on this version histories and structural information, Rahman et al. also propose a class level bug localization technique (Rahman et al., 2015). Here, authors identify suspicious score for each class by combining the scores of rVSM (Zhou et al., 2012) with the frequently changed file information. Another score is generated from prioritization component and these two scores are combined. However, as these techniques suggest classes, it demands manual investigation to find buggy locations in more granular levels (e.g., buggy methods or statements).

2.2 Method Level Bug Localization

Since the suggestion of more granular level (i.e., methods or statements) bug localization technique is needed, few techniques are available to suggest methods as buggy. In method level bug localization, methods are identified as the unit of suggestions.

Lukins et al. propose a method level bug localization technique (Lukins et al., 2008) where each method of the source code is considered as the unit of measurement. The source code is processed using stop words removal, language specific keywords removal, multi-word identifiers and stemming. Later, Latent Dirichlet Allocation (LDA) model is generated and is queried using bug report to get the buggy methods. Another technique is proposed by considering whole source code where semantic meanings of each method have been extracted (Nichols, 2010). Authors gather extra information from the previous bug history. When a new bug arrives, Latent Semantic Indexing (LSI) is applied on the method documents to identify the relationships between the terms of the bug report and the concepts of the method documents. Based on that, a list of buggy methods is suggested.

A feature location based bug localization is introduced in (Alhindawi et al., 2013) where source code corpus is enhanced with stereotypes. Stereotypes represent the details of each word which are commonly used in programming. For example, the stereotype 'get' means a method returns a value. These stereotype information are derived automatically from the source code via program analysis. After adding stereotype information with the source code methods, IR technique is used to execute the queries. However, as the technique suggests method as buggy, it requires lots of time to find the buggy statements.

The first dynamic analysis based bug localization technique is proposed by Wilde et al. where minimization of source code has been performed by considering passing and failing test cases of the program (Wilde et al., 1992). However, due to using passing test cases, irrelevant features may be included in the domain of search space. As a result, the accuracy of bug localization may be hampered. An improved version of this is proposed by Eisenbarth et al. where both dynamic and static analysis of the source code are combined (Eisenbarth et al., 2003). Here, static analysis identifies the dependencies among the data to locate the features in a program while dynamic analysis collects the source code execution traces for a set of scenarios. Poshyvanyk et al. propose PROMESIR where authors also use both static and dynamic analysis of the source code. Through dynamic analysis, executed buggy methods are extracted for a bug. Initially, the two analysis techniques produce bug similarity scores differently without interacting with each other. Finally, these two scores are combined and a weighted ranking score for each method is measured. Although this technique uses dynamic information of the source code, it fails to minimize the solution search space during static analysis.

Rahman et al. introduce MBuM which focuses on the search space minimization by applying dynamic analysis of the source code (Rahman and Sakib, 2016). Executed methods are identified by reproducing the bug, and during static analysis only the contents of those executed methods are extracted. As a result, irrelevant source code is removed from the search space. The remaining texts are processed to measure the similarity between the bug report and source code method using a modified Vector Space Model (mVSM). In mVSM, the length of the method is incorporated with the existing VSM.

From the above discussion, it is clear that the accuracy depends on the identification of valid information domain by removing irrelevant source code. Existing approaches suggest either classes or methods as buggy which also demand manual inspections to find more granular buggy locations i.e., buggy statements.

3 THE PROPOSED APPROACH

This section proposes Statement level Bug Localization (SBL) which consists of two major phases such as irrelevant search space minimization but maximizing relevant data domain, and the ranking of buggy statements from that relevant data domain. The details are described in the followings.

3.1 Minimizing Irrelevant Search Space

Since MBuM provides better ranking of buggy methods than others, that ranked list of buggy methods is used here as minimized search space (Rahman and Sakib, 2016). To do so, irrelevant source code is discarded by considering source code execution traces. Only the relevant method contents are processed using several text processing techniques such as stop word removal, programming language specific keyword removal, multi-word identifications, semantic meaning extraction and stemming which produce code corpora. Similarly, bug report is also processed to generate bug corpora. Finally, textual similarity is measured between code and bug corpora by mVSM. The overall procedure is demonstrated in Figure 1.

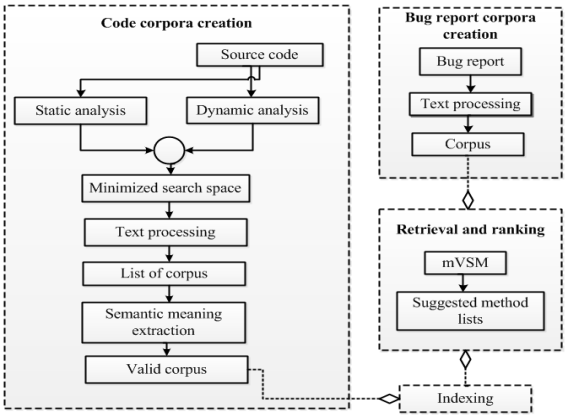

Figure 1: The overall procedure of MBuM.

3.2 Statement Level Bug Localization using Minimized Search Space

In this section, source code statements are suggested as buggy by only considering the ranked buggy methods described in Section 3.1. For each ranked method, a Method Statement Dependency Graph (MSDG) is generated where each statement acts as a node. For each node, a super node is generated by combining the node and its predecessors, because the execution of a node may depend on its predecessor nodes. This process maximizes the valid information of a statement. This graph is mentioned as Node Predecessor-node Dependency Graph (NPDG). To suggest a list of buggy statements, node similarity score (N_s) is measured between each node of NPDG and the bug report using VSM. For each node, N_s score is weighted by the score of the method (M_s) (obtained from MBuM) which contains the node.

3.2.1 Generation of Method Statement Dependency Graph (MSDG)

MSDG is developed because one statement may depend on other statements, and it is very often defects propagate from one to another. As a result, the statement which is executed first may affect the statements which use the result of that statement.

For a better understanding regarding the generation of MSDG, a *Java* source code class is considered (see Figure 2). Here, two methods namely *calculateProfit* and *calculateOperationCost* are available within class *IncomeProfit*. Each method introduce some local, global and method call variables. For example *calculateProfit* contains sell, buy, etc. as local variables where profit is a global variable. Except this, *calculateProfit* also depends on the result of *calculateOperationCost* method. So, *operationCost* at line 7 is a method call variable.

```
1  package bank;
2  public class IncomeProfit {
3  double profit= 0.0;
4  public void calculateProfit(int sell1, int buy1){
5        int sell=sell1;
6        int buy=buy1;
7        double operationCost=calculateOperationCost();
8        profit+=sell-buy-operationCost;
9        if(profit>0.0){
10             System.out.println("Yes you gain profit");
11       }
12       else{
13             System.out.println("Sorry you lose");
14       }
15 }
16 private double calculateOperationCost() {
17       double electricityCost=50;
18       double waterCost=10;
19       double gasCost=15;
20       double cost=electricityCost+waterCost+gasCost;
21       return cost;
22 }
```

Figure 2: Source code.

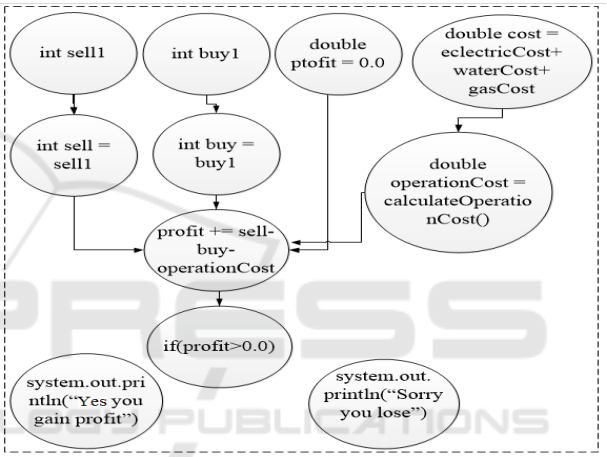

Figure 3: MSDG for *calculateProfit*.

For *calculateProfit* method, an MSDG is depicted in Figure 3 which is a directed graph. Let, $G = (V,E)$ is a graph where V is the set of vertices or statements of the method. Each vertex stores some properties of the source code (i.e., package name, class name, method name, method score and line number of the statement). Figure 4 shows these properties of node *int sell* = *sell*1 of MSDG in Figure 3. Similar to this node, all other nodes also contain these types of information. E is the set of directed edges between statements which represents the data flow. If there exists an edge (v_1, v_2) from node v_1 to v_2, v_1 is said to be a predecessor of v_2. In source code, the predecessors can be defined as follows: if a statement contains an assignment such as $x = y + z$, the last modified statements of y and z are the predecessors of x. For arithmetic assignment operators such as $x+ = y$, the last modification of x is also considered as a predecessor. In case of increment (x++), decrement (x- -) and conditional statements (e.g., if $(x > 0)$), statement containing the last modification

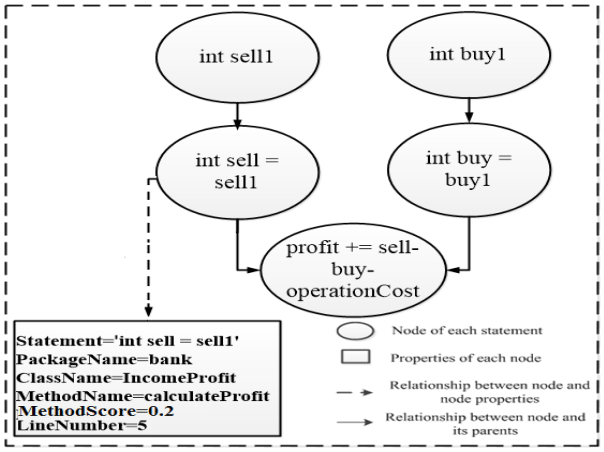

Figure 4: MSDG containing the details of a node.

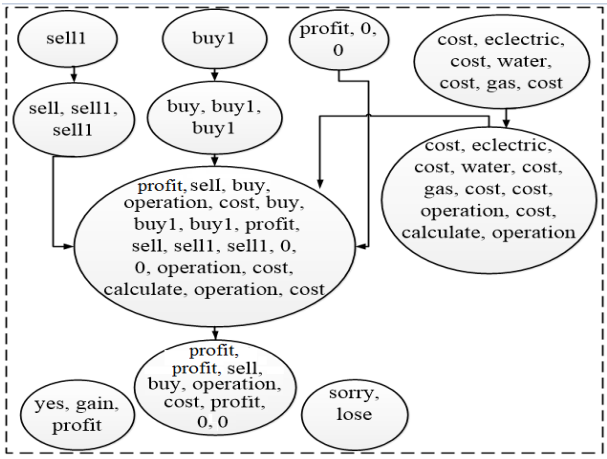

Figure 5: Node Predecessor-node Dependency Graph (NPDG) corresponding to Figure 3.

of x is a predecessor. These are needed to maximize the valid information of each statement which can improve the accuracy of bug localization technique. To find the last modification of a variable, three cases are identified based on the types of predecessor variables such as Local Variable Usage (LVU), Global Variable Usage (GVU) and Method Return Variable Call (MRVC).

1. In LVU, scopes of predecessor variables are bounded within that method. In *calculateProfit* method of Figure 2, *int sell1* affects *sell*, hence *int sell1* is the predecessor of *sell*. Similarly, *int buy1* is the predecessor of *buy* (see Figure 2) and so on.
2. In GVU, the predecessor variable is globally declared and multiple methods may use that variable. The declaration of the $double\ profit = 0.0$ in line 3 affects line 8 where $profit$ is calculated using $sell$, buy, $operationCost$ and $profit$. Therefore, line 3 is the predecessor of line 8.
3. MRVC indicates that a statement calls a method which returns a value. In this case, the last modification of that called method's return statement is the predecessor of the current node. In Figure 2, line 7 shows that $operationCost$ calls a method (i.e., $calculateOperationCost$), which returns $cost$ variable (see line 20). Therefore, line 20 is the predecessor of the calling node (i.e., line 7), as shown in Figure 3.

From Figure 3 it is found, the last two nodes such as *system.out.println("Yes you gain profit")* and *system.out.println("Sorry you lose")* have no edge with other nodes because these do not use any variables or call any methods. As a result, the above mentioned cases (i.e., LVU, GVU and MRVC) are not satisfied.

3.2.2 Generation of Node Predecessor-node Dependency Graph (NPDG)

In this phase, textual information is maximized by combining the corpora of each node with its predecessors (i.e., only the direct predecessors). At first, each node of MSDG containing a statement of the source code is processed. To do so, text are processed using the techniques, described in Section 3.1. Finally, a list of valid code corpora is obtained.

As SBL depends on the similarity between the bug report and statements of the source code, increasing valid information of a statement is an implicit demand. Hence, a super node is generated by combining each node with its predecessor nodes' corpora because predecessors may affect its successors, results in an NPDG. Figure 5 shows an NPDG which is derived from Figure 3 by combining each node with its predecessor nodes. In Figure 3, $int\ sell = sell1$ is a node while in NPDG, this node is incorporated with its predecessor node (i.e., $int\ sell1$). As a result, the number of corpus is increased for statement $int\ sell = sell1$.

After the generation of NPDG, similarity score is measured between the bug report and each node of NPDG. This graph is traversed to find the frequencies of shared terms between the bug corpora and each node corpora of NPDG. As several kinds of term-frequency (tf) have already been available (e.g., logarithm, boolean variants), logarithm variant performs better than others (Croft et al., 2010). $tf(t,n)$ and $tf(t,b)$ are calculated according to Equation 1 and 2.

$$tf(t,n) = \log f_{tn} + 1 \quad (1)$$

$$tf(t,b) = \log f_{tb} + 1 \quad (2)$$

f_{tn} and f_{tb} are the frequencies of a term of NPDG node and bug report respectively, which are used for

measuring the similarity between bug report and each statement of the source code (see Equation 3).

$$N_s(n,b) = VSM(n,b) = cos(n,b) = \sum_{t \in b \cap n}((\log f_{tn}+1) \times (\log f_{tb}+1)) \times \frac{1}{\sqrt{\sum_{t \in n}(\log f_{tn}+1)^2}} \times \frac{1}{\sqrt{\sum_{t \in b}(\log f_{tb}+1)^2}} \quad (3)$$

Here, $N_s(n,b)$ denotes the similarity score of each node. A statement containing large score (N_s) represents that the statement has large similarity with the bug report. Similarly, low score of a statement indicates that the bug has minimum effect on that node.

The score of each statement is weighted by the score of the method using Equation 4. This is because, a statement contains a few number of words compared to a method (Moreno et al., 2013). So, it is more obvious that the large ranking scored methods' are more likely liable for being buggy.

$$SBL_{score} = N_s \times M_s \quad (4)$$

M_s and N_s represent the score of a method containing statement s and the statement similarity score respectively. SBL_{score} denotes the ranking score of each statement, and based on that, statements are ranked.

4 RESULT ANALYSIS

To measure the effectiveness of SBL, several bug reports from three open source projects named as Eclipse, SWT and PasswordProtector are considered. The experiments are conducted for validating the ranking of buggy statements. To measure the accuracy of SBL, Top N Rank, Mean Reciprocal Rank (MRR) and Mean Average Precision (MAP) are used as metrics which are also commonly used in bug localization. The data collection followed by the experimental details are discussed in the followings.

4.1 Data Collection

Eclipse, SWT and PasswordProtector are used as the subject of evaluation. Eclipse is a widely used open source Integrated Development Environment (IDE) which is written in Java. SWT is a widget toolkit which is integrated with Eclipse. PasswordProtector[1] is also an open source project used to encrypt passwords for accessing multiple websites.

Different versions of Eclipse and SWT (e.g., version 2.1.0, 3.0.1, 3.0.2 and 3.1.0) are chosen which contain large volume of source code. For example, Eclipse 3.0.2 contains 12,863 classes, 95,341 methods and 1,86,772 non-empty statements, SWT contains 489, 18,784 and 32,032 classes, methods and non-empty statements respectively. These benchmark projects lack the available patches with statement level bug fixing because, none of the researches are conducted on statement level. Hence, SBL is evaluated using 230 bugs from three projects (i.e., 120 from Eclipse, 100 from SWT and 10 from PasswordProtector). These bugs are manually collected from Bugzilla for Eclipse and SWT. For each bug, corresponding patched files are also collected to validate the results provided by SBL. Meanwhile, the bug reports and fixed files of PasswordProtector are collected from the developers of the projects (Rahman, 2016).

[1]https://raw.githubusercontent.com/shanto-Rahman/SBL/master/PasswordProtector.zip

Table 1: The performance of SBL by considering 230 bugs.

Project name	Top 1	Top 5	Top 10	Top 20	MRR	MAP
Eclipse	34 (28.33%)	70 (58.33%)	79 (65.83%)	98 (81.66%)	45.8%	38.5%
SWT	38 (38%)	63 (63%)	72 (72%)	85 (85%)	51.3%	44.2%
Password Protector	4 (40%)	9 (90%)	10 (100%)	10 (100%)	66%	61%

The details of each bug are available in (Rahman, 2016) where bug id, description and buggy locations are given. It is noteworthy that stack trace is omitted from the bug description as it may bias the evaluation.

4.2 Research Questions and Evaluation

In this section, SBL is validated by addressing two research questions **RQ1** and **RQ2**. **RQ1** states how many bugs are successfully located by suggesting buggy statements. And **RQ2** demonstrates the effect of considering predecessors of each node. The following discussion holds the detail description of each research question along with their evaluation.

4.2.1 RQ1: How Many Bugs are Successfully Located at Statement Level?

For each bug, the ranked list of buggy statements suggested by SBL are compared with the original patched buggy statements. If the buggy statements are ranked at the top 1, top 5, top 10 or top 20, the bug is considered effectively localized. MRR and MAP metrics are also used to prove the efficacy of SBL.

Table 1 presents SBL locates 28.33% buggy statements at the top in Eclipse. This refers that developers may not traverse any more statements for 28.33% bugs. Besides, 58.33% and 65.83% bugs are located within top 5 and top 10 respectively. In case of SWT, SBL locates 38%, 63% and 72% at the top 1, top 5 and top 10. For PasswordProtector, 40% bugs are localized at the top 1, 90% and 100% bugs are correctly localized within top 5 and top 10 suggestions. For Eclipse, SWT and PasswordProtector, MRR of SBL is 45.8%, 51.3% and 66% while MAP is 38.5%,

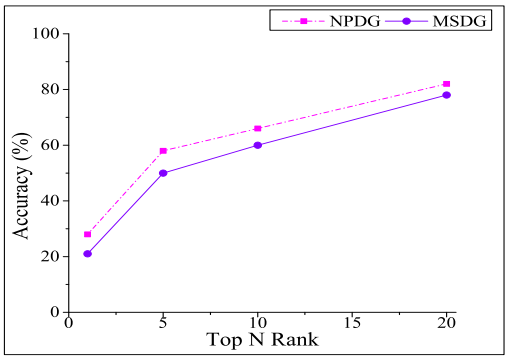

Figure 6: Demonstrating the effects of predecessor nodes in 120 bugs of Eclipse using Top N Rank.

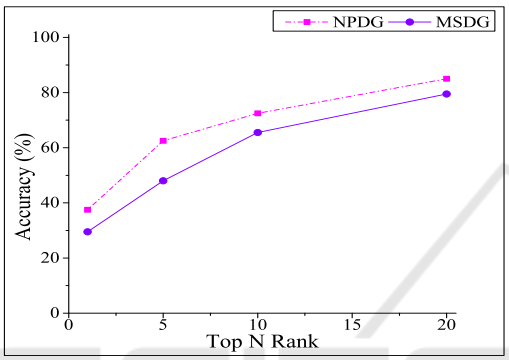

Figure 7: Demonstrating the effects of predecessor nodes in 100 bugs of SWT using Top N Rank.

44.2% and 61% respectively. These results indicate, SBL can effectively localize the buggy statements.

4.2.2 RQ2: Does NPDG Improve the Accuracy of Bug Localization?

SBL creates super node by combining each node and its predecessor nodes. Hence, it is an intrinsic demand to show whether the consideration of these predecessor nodes improve the bug localization accuracy or not. To validate this, a comparison is made between without considering predecessor (i.e., MSDG) and with consideration of predecessors (i.e., NPDG).

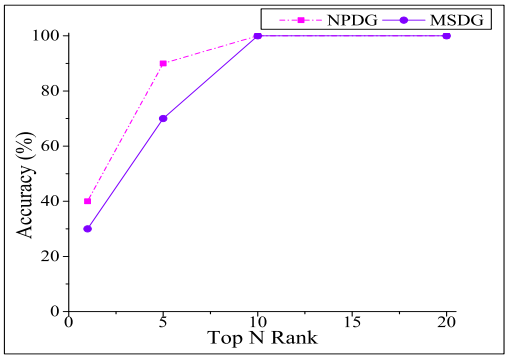

Figure 8: Demonstrating the effects of predecessor nodes in 10 bugs of PasswordProtector using Top N Rank.

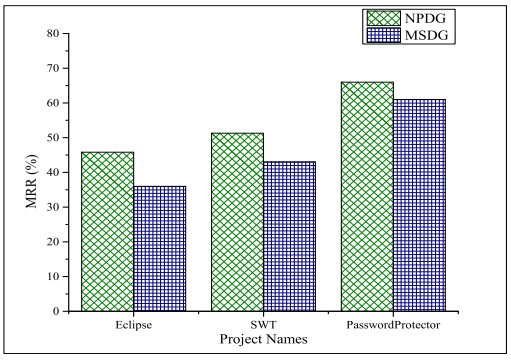

Figure 9: Impact of the predecessor node using MRR.

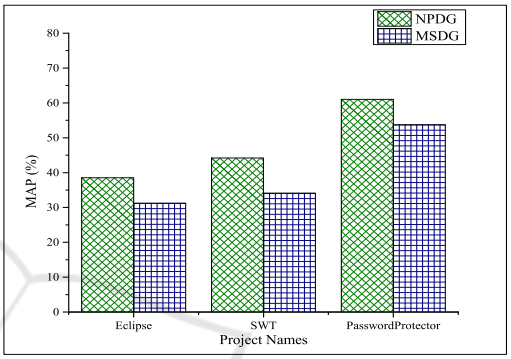

Figure 10: Impact of the predecessor node using MAP.

Figure 6-8 show the accuracy of ranking buggy statements in Top N Rank by MSDG which is comparatively lower than that of NPDG for all the studied projects. Because buggy statements depend on its previously executed statements. And NPDG is more effective when the bug is not stayed in a statement rather it propagates from one statement to another.

Figure 6 holds a comparative study between the results of considering NPDG and MSDG in case of Eclipse. The ranking of buggy statements in Eclipse shows that 21.66% and 28.33% bugs are located at the 1^{st} position in case of MSDG and NPDG respectively. For SWT, 29% and 38% bugs are located at the 1^{st} position for MSDG and NPDG. Although PasswordProtector is comparatively low volume project than other two, here also NPDG performs better than MSDG, i.e., 40% and 30% bugs are located at the 1^{st} position respectively. Thus NPDG improves 6.67%, 9% and 10% performance in case of Eclipse, SWT and PasswordProtector respectively over MSDG.

The effects of using predecessors are shown in Figure 9 and 10 where MRR and MAP are considered as metrics respectively. NPDG also shows higher values than MSDG. In case of MRR, 9.8%, 8.3% and 5% (Figure 9) improvements are found by conside-

ring NPDG instead of MSDG for Eclipse, SWT and PasswordProtector respectively. Meanwhile, 7.3%, 10.1% and 7.3% (Figure 10) accuracy improvements have been found (on MAP) by NPDG over MSDG in the aforementioned three projects respectively.

5 CONCLUSION

In this paper, a novel statement level bug localization technique named as SBL is proposed where irrelevant search space is discarded using the source code dynamic analysis. The relevant search space is further minimized by ranking the buggy methods. Later, for each suggested method, a Method Statement Dependency Graph (MSDG) is generated which holds the relationship among the statements. For invoking predecessor node information, a Node Predecessor-node Dependency Graph (NPDG) is generated for each method where bag of words of each node and its predecessor nodes are combined. The similarity between each node (i.e., statement) of NPDG and the bug report is measured to rank the buggy statements. Effectiveness of SBL has been evaluated on three open source projects. The experimental results show that SBL can successfully rank buggy statements. It is also evident from the results, the consideration of predecessor nodes improve the accuracy.

Since SBL performs well in different types of projects, in future it can be applied in industrial projects to assess its effectiveness in practice.

ACKNOWLEDGEMENT

This research is supported by the fellowship from ICT Division, Bangladesh. No-56.00.0000.028.33.028.15-214 Date 24-06-2015.

REFERENCES

Alhindawi, N., Dragan, N., Collard, M. L., and Maletic, J. I. (2013). Improving feature location by enhancing source code with stereotypes. In *2013 IEEE International Conference on Software Maintenance*, pages 300–309. IEEE.

Croft, W. B., Metzler, D., and Strohman, T. (2010). *Search engines: Information retrieval in practice*. Addison-Wesley Reading.

Eisenbarth, T., Koschke, R., and Simon, D. (2003). Locating features in source code. *IEEE Transactions on Software Engineering*, 29(3):210–224.

Lukins, S. K., Kraft, N., Etzkorn, L. H., et al. (2008). Source code retrieval for bug localization using latent dirichlet allocation. In *15th Working Conference on Reverse Engineering, WCRE'08*, pages 155–164. IEEE.

Moreno, L., Bandara, W., Haiduc, S., and Marcus, A. (2013). On the relationship between the vocabulary of bug reports and source code. In *International Conference on Software Maintenance (ICSM)*, pages 452–455. IEEE.

Nichols, B. D. (2010). Augmented bug localization using past bug information. In *48th Annual Southeast Regional Conference*, page 61. ACM.

Poshyvanyk, D., Gueheneuc, Y.-G., Marcus, A., Antoniol, G., and Rajlich, V. C. (2007). Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. *IEEE Transactions on Software Engineering*, 33(6):420–432.

Rahman, S. (2016). shanto-rahman/source code: 2016. https://github.com/shanto-Rahman/SBL.

Rahman, S., Ganguly, K., and Kazi, S. (2015). An improved bug localization using structured information retrieval and version history. In *18th International Conference on Computer and Information Technology (ICCIT)*, pages 190–195. IEEE.

Rahman, S., Rahman, M. M., and Sakib, K. (2016). An improved method level bug localization approach using minimized code space. *Communications in Computer and Information Science (Accepted)*.

Rahman, S. and Sakib, K. (2016). An appropriate method ranking approach for localizing bugs using minimized search space. In *Proceedings of the 11th International Conference on Evaluation of Novel Software Approaches to Software Engineering*, pages 303–309.

Saha, R. K., Lease, M., Khurshid, S., and Perry, D. E. (2013). Improving bug localization using structured information retrieval. In *28th International Conference on Automated Software Engineering (ASE)*, pages 345–355. IEEE.

Sisman, B. and Kak, A. C. (2012). Incorporating version histories in information retrieval based bug localization. In *9th IEEE Working Conference on Mining Software Repositories*, pages 50–59. IEEE Press.

Tantithamthavorn, C., Ihara, A., and Matsumoto, K.-i. (2013). Using co-change histories to improve bug localization performance. In *Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD), 2013 14th ACIS International Conference on*, pages 543–548. IEEE.

Wang, S. and Lo, D. (2014). Version history, similar report, and structure: Putting them together for improved bug localization. In *22nd International Conference on Program Comprehension*, pages 53–63. ACM.

Wilde, N., Gomez, J. A., Gust, T., and Strasburg, D. (1992). Locating user functionality in old code. In *Software Maintenance, 1992. Proceerdings., Conference on*, pages 200–205. IEEE.

Zhou, J., Zhang, H., and Lo, D. (2012). Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports. In *34th International Conference on Software Engineering (ICSE)*, pages 14–24. IEEE.