Impact of Similarity on Repairing Small Programs: A Case Study on QuixBugs Benchmark

Moumita Asad
Institute of Information Technology,
University of Dhaka
Dhaka, Bangladesh
bsse0731@iit.du.ac.bd

Kishan Kumar Ganguly
Institute of Information Technology,
University of Dhaka
Dhaka, Bangladesh
kkganguly@iit.du.ac.bd

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

ABSTRACT

Similarity analysis plays an important role in automated program repair by finding the correct solution earlier. However, the effectiveness of similarity is mostly validated using common benchmark Defects4J which consists of 6 large projects. To mitigate the threat of generalizability, this study examines the performance of similarity in repairing small programs. For this purpose, existing syntactic and semantic similarity based approaches, as well as a new technique of combining both similarities, are used. These approaches are evaluated using QuixBugs, a dataset of diverse type bugs from 40 small programs. These techniques fix bugs faster by validating fewer patches than random patch selection based approach. Thus, it proves the effectiveness of similarity in repairing small programs.

KEYWORDS

semantic similarity, syntactic similarity, automated program repair, quixbugs

ACM Reference Format:
Moumita Asad, Kishan Kumar Ganguly, and Kazi Sakib. 2020. Impact of Similarity on Repairing Small Programs: A Case Study on QuixBugs Benchmark. In *IEEE/ACM 42nd International Conference on Software Engineering Workshops (ICSEW'20), May 23–29, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3387940.3392182

1 INTRODUCTION

Automated program repair finds the correct solution of a bug based on a specification, e.g., test cases [1]. Since the solution space is infinite, numerous patches are generated. In addition, a plausible solution - patch that passes all the test cases, can be incorrect. To overcome these problems, existing approaches such as [4], [5], use similarity between faulty code and fixing ingredient (code used to fix the bug). These techniques are mostly validated using Defects4J (a dataset of 395 bugs from 6 large projects) since it is the first Java benchmark of real bugs [2]. Therefore, the impact of similarity on small programs has not been explored thoroughly.

To ensure generalizability, similarity analysis should be effective in both large and small projects. Existing studies [4], [5] reveal the importance of similarity for repairing large programs. However, small programs may incur any of the two problems: (1) they may not contain similar code due to their size (2) all the code fragments may be similar due to limited number of statements, expressions or identifiers (e.g, most of the statements operating on the same variable), resulting in incorrect plausible patches. To deepen the understanding of the strengths and weaknesses of different similarities, their performance needs to be analyzed using small programs.

To the best of the knowledge, only a few studies [1], [5] have evaluated their similarity based approach on small programs. Wen et al. have executed CapGen on IntroClassJava, a dataset of 297 bugs from 6 small projects [5]. Although they have reported that CapGen correctly repairs 25 bugs, the detailed analysis (e.g., number of incorrect plausible patches) is not reported [2]. Asad et al. have evaluated their combined similarity based approach using only 22 bugs from IntroClassJava [1]. Besides, the used bugs lack diversity-most of these are related to incorrect condition and variable.

This work is mainly an extension of [1]. It investigates the performance of different similarity metrics on small programs. Apart from considering the metrics (genealogical, variable similarity and normalized Longest Common Subsequence (LCS)) used in [1], this paper introduces a new syntactic similarity metric named token similarity to overcome the limitation of normalized LCS. Normalized LCS is sensitive to text order, whereas token similarity ignores the ordering. It only inspects whether a token (e.g., *identifiers*) exists regardless of its position. Thus, 5 patch prioritization techniques are developed by using syntactic, semantic similarity or a combination of both. All of these approaches follow the same repairing process. For evaluation, QuixBugs benchmark is selected since it contains diverse bugs including incorrect method call, missing arithmetic expression etc [6]. Furthermore, space and time complexity of these bugs are significant (e.g., 14 programs contain recursion).

To assess the correctness of the plausible patches, both automated test cases from [6] and manual inspection are used. On average, each approach generates 16 plausible patches out of which 10 are correct. Moreover, these techniques fix bugs faster than baseline approach that selects patches randomly.

2 METHODOLOGY

This study works in four steps namely fault localization, patch generation, patch prioritization and patch validation.

(1) Fault Localization: It identifies the faulty Abstract Syntax Tree (AST) node of type *Expression* using GZoltar tool (version 1.6.1) with Ochiai algorithm [1]. GZoltar outputs line-number wise suspicious score of a program, which are next mapped to AST nodes.

(2) Patch Generation: After identifying fault nodes, they are replaced with fixing ingredients to generate patches. Similar to [5],

ICSEW'20, May 23–29, 2020, Seoul, Republic of Korea

ACM ISBN 978-1-4503-7963-2/20/05…$15.00
https://doi.org/10.1145/3387940.3392182

Expression nodes from the buggy file are used as fixing ingredients.
(3) Patch Prioritization: To prioritize the generated patches, this study uses 4 types of similarity that capture syntactic or semantic similarity. Genealogical and variable similarity are used to measure semantic similarity. Normalized LCS and token similarity are considered for calculating syntactic similarity.

- **Genealogical Similarity (GS):** It checks whether faulty node and fixing ingredient are frequently used with the same type of code elements (e.g., inside *if* statement) [5]. To extract the genealogy contexts of a node, the type of it's ancestor and sibling nodes are inspected through AST traversal. The ancestors are traversed until a method declaration is found. For sibling nodes, *Expressions* or *Statements* type nodes within the same block of the specified node are examined.
- **Variable Similarity (VS):** To measure variable similarity, names and types of variables used by the faulty node and the fixing ingredient are compared using Jaccard index [5].
- **Normalized LCS (NL):** Normalized LCS between faulty code and fixing ingredient is computed by considering them as a sequence of characters.
- **Token Similarity (TS):** To calculate token similarity, the faulty node and fixing ingredient are tokenized. Similar to [4], camel case identifiers are further split and converted into lower-case (e.g., *isTrue* is converted into *is* and *true*). Next, token similarity is computed using Jaccard index.

Using these metrics, 5 patch prioritization approaches are developed, as shown in Table 1. These techniques follow the same repairing process (only the patch ranking score calculation varies). Patches are sorted in descending order based on the score provided by these approaches.

Table 1: Patch Ranking Score Calculation for Different Approaches

Approach	Score Calculation
Semantic Similarity Based Approach (*SSBA*)	FL*(GS+VS)
Combined (LCS) Approach (*Com-L*)	FL*(GS+VS+NL)
Combined (Token) Approach (*Com-T*)	FL*(GS+VS+TS)
LCS Based Approach (*LBA*)	FL*NL
Token Based Approach (*TBA*)	FL*TS

[1] FL denotes the suspicious score of a faulty node.

(4) Patch Validation: It checks the correctness of a patch by executing test cases. This step continues until all the patches are checked or a predefined time-limit (90 minutes) exceeds [5].

3 RESULT ANALYSIS

To understand the effectiveness of the similarity based approaches, these are compared with baseline technique that selects patch randomly [3]. All of these methods were executed on QuixBugs. The baseline technique was run 100 times due to its stochastic nature. The experiment was run on a Ubuntu server with Intel Xeon E5-2690 v2 @3.0GHz and 64GB physical memory. The results and the implementations are publicly available at GitHub[1]. Among 40 bugs, GZoltar fails to produce output for 3 programs- *bitcount*, *find_first_in_sorted* and *sqrt* since these bugs generate infinite loop. For the rest 37 bugs, these approaches generate plausible patch for 16, as shown in Table 2. The correctness of a patch is both manually and automatically analyzed using the test cases provided by [6].

All similarity based methods except *SSBA* can correctly fix more bugs and achieve better precision than baseline technique. For *SSBA*, there is a tie between incorrect plausible and correct patch for program *flatten* and *next_palindrome*. However, all the similarity based methods can fix bugs faster by examining fewer patches than baseline technique. It proves that similarity analysis is effective even for small programs [3]. Moreover, incorporating both similarities do not increase the bug fixing time. The reason is combined techniques rank the correct patch higher and thus, save time in patch validation. For the 9 bugs fixed by all the 5 techniques, the median rank of the correct patch is 126.5 and 146 for *Com-L* and *Com-T*, whereas it is 168.5, 305 and 260 for *SSBA*, *LBA* and *TBA* respectively.

Table 2: Performance of the Approaches on QuixBugs Benchmark

Approach	Correct Patches	Incorrect Patches	Precision (%)	Median Time (Seconds)
SSBA	9	7	56.25	39.61
Com-L	11	5	68.75	14.35
Com-T	10	6	62.50	14.69
LBA	11	5	68.75	22.67
TBA	10	6	62.50	25.90
Baseline	9	7	56.25	76.18

4 CONCLUSION

This paper analyzes the impact of similarity metrics on small programs. Hence, 5 techniques using existing syntactic and semantic similarity based approaches, as well as a new method of integrating both similarities, are developed. To measure semantic similarity, genealogical and variable similarity are used. To capture syntactic similarity, normalized LCS and token similarity are used. These approaches are evaluated on QuixBugs dataset. These techniques fix bugs faster by examining fewer patches than random patch selection based approach, which indicates the effectiveness of similarity in fixing small programs [3]. In future, these methods will be further explored using other small project datasets (e.g., IntroClassJava [2]).

ACKNOWLEDGMENTS

This research is supported by the fellowship from Information and Communication Technology Division, Bangladesh. No-56.00.0000.028 .33.093.19-427; Dated 20.11.2019. The virtual machine facility is provided by the Bangladesh Research and Education Network (BdREN).

REFERENCES

[1] Moumita Asad, Kishan Kumar Ganguly, and Kazi Sakib. 2019. Impact Analysis of Syntactic and Semantic Similarities on Patch Prioritization in Automated Program Repair. In *ICSME'2019*. IEEE, 328–332.
[2] Thomas Durieux, Fernanda Madeiral, Matias Martinez, and Rui Abreu. 2019. Empirical review of Java program repair tools: a large-scale experiment on 2,141 bugs and 23,551 repair attempts. In *FSE'2019*. 302–313.
[3] Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziying Dai, and Chengsong Wang. 2014. The strength of random search on automated program repair. In *ICSE'2014*. ACM, 254–265.
[4] Ripon K Saha, Yingjun Lyu, Hiroaki Yoshida, and Mukul R Prasad. 2017. Elixir: Effective object-oriented program repair. In *ASE'2017*. IEEE, 648–659.
[5] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. 2018. Context-aware patch generation for better automated program repair. In *ICSE'2018*. ACM, 1–11.
[6] He Ye, Matias Martinez, Thomas Durieux, and Martin Monperrus. 2019. A comprehensive study of automatic program repair on the QuixBugs benchmark. In *IBF'2019*. IEEE, 1–10.

[1] https://github.com/mou23/Impact-of-Combining-Sytactic-and-Semantic-Similarity-on-Patch-Prioritization