Impact of Combining Syntactic and Semantic Similarities on Patch Prioritization while using the Insertion Mutation Operators

{Mohammad Raihan Ullah, Nazia Sultana Chowdhury, Fazle Mohammed Tawsif}

Institute of Information and Communication Technology
Shahjalal University of Science and Technology
Sylhet, Bangladesh

Abstract—Patch prioritization ranks candidate patches based on their likelihood of being correct. The fixing ingredients that are more likely to be the fix for a bug, share a high contextual similarity. A recent study shows that combining both syntactic and semantic similarity for capturing the contextual similarity, can do better in prioritizing patches. In this study, we evaluate the impact of combining the syntactic and semantic features on patch prioritization using the Insertion mutation operators. This study inspects the result of different combinations of syntactic and semantic features on patch prioritization. As a pilot study, the approach uses genealogical similarity to measure the semantic similarity and normalized longest common subsequence, normalized edit distance, cosine similarity, and Jaccard similarity index to capture the syntactic similarity. It also considers Anti-Pattern to filter out the incorrect plausible patches. The combination of both syntactic and semantic similarity can reduce the search space to a great extent. Also, the approach generates fixes for the bugs before the incorrect plausible one. We evaluate the techniques on the IntroClassJava benchmark using Insertion mutation operators and successfully generate fixes for 6 bugs before the incorrect plausible one. So, considering the previous study, the approach of combining syntactic and semantic similarity can able to solve a total number of 25 bugs from the benchmark, and to the best of our knowledge, it is the highest number of bug solved than any other approach. The correctness of the generated fixes are further checked using the publicly available results of CapGen and thus for the generated fixes, the approach achieves a precision of 100%.

Index Terms—patch prioritization, semantic similarity, syntactic similarity, automated program repair, mutation operation, contextual similarity

I. Introduction

Automated Program Repair is the automatic repair of software bugs without the intervention of a human programmer. Automatic debugging and repairing of the software defects can make a programmer's life a lot easier.

Manual Debugging is a very difficult, time-consuming, and painful task. Study shows that the global cost of debugging is 312 billion dollars annually and software developers spend 50 percent of their time fixing bugs [1].

Automated Program Repair (APR) reduces the bug-fixing effort by automatically generating patches that satisfy a specification e.g. Test cases [2]. A patch is a modification applied to a program for fixing a bug. A typical APR approach takes

a faulty program and a set of test cases as input (where the program fails at least one test case) and produces a patch to repair the fault (the patched program at least passes all the tests). This type of technique is called the generate-and-validate technique. Patches that are applied to a program could exist in the faulty program itself as well as in a non-local program. The study founds that up to 69% of the repaired code fragments (in the form of code lines) can be obtained from both the local program and non-local programs [3], [4].

Throughout the years, several Automated Program Repair (APR) techniques have been introduced. Such as GenProg, HDRepair, CapGen, ssFix, SimFix, Elixir, etc [7], [8], [9], [10], [11], [12], [13], [14]. Many of these techniques e.g. SimFix, ssFix, CapGen, Elixir, etc [9], [10], [12], [15], [16] are known as search-based APR. So, APR is often described as a search problem in a patch candidate space. In a candidate space, there are the fixing code fragments or patches that can be a possible fix for that particular bug. Testing the fixing codes earlier that are more likely to be the fix for the bug can reduce bug fixing time. Sorting candidate patches based on their probability of being the correct fix for the bug, is called patch prioritization.

The faulty program and the code fragment where the fix resides share a high contextual similarity. To capture these similarity, some method [18] [19] [20] uses Syntactic features and some method [22] uses Semantic features. A recent study [5] shows that combining Syntactic and Semantic features in patch prioritization can reduce bug fixing time and achieve higher precision. So in this study, we use the combination of Syntactic and Semantic features in Patch Prioritization and observe the impacts while working with *Insertion* as the mutation operation. We also evaluate some different ways of combining those feature scores to calculate the final score and analyze their impact on Patch Prioritization. We evaluate the techniques on IntroClassjava benchmark [23] which contains 297 real-life bugs. We were successfully able to solve six bugs with a precision of 100%.

II. Related Work

Testing the candidate patches early which, are more likely to be the fix can help to reduce the bug fixing time and give some

treatment to the search space explosion problem of search-based Automated Program Repair.

The most representative of the search-based APR category is GenProg [17], which searches for correct patches via genetic programming algorithm. GenProg constrains the genetic operations of mutation and crossover to operate only on the region of the program that is relevant to the error. GenProg repairs bugs with the help of the following components: Mutation Space, Crossover, Evaluation Function. GenProg presents a program as an Abstract Syntax Tree (AST) and modifies the faulty node using the mutation operations such as insertion, deletion, and Replacement operators. AST of the program is modified using mutation from the mutation space. As a result of the modification, variants of the faulty program are generated then they are passed to the evaluation function to see if those variants pass the test cases.

Elixir [18] generates patches based on generate-and-validate techniques. For a given bug, Elixir takes as input a buggy program, a test suite, and optionally a bug report, then produces a patch that passes all the test cases. Elixir works in the following ways: Bug Localization, Generating Candidate Patches, Ranking and Selection of Candidate Patches, Validating Selected Candidate Patches. Elixir uses four syntactical features to rank the patches. They are Distance score between faulty and fixing ingredients, Contextual Similarity, Frequency in the context, Bug Report Similarity. The approach introduces 8 templates and validates only the top 50 patches generated from each template.

SimFix [19] defines a method to obtain a search space from existing patches based on an abstract space definition on AST types. The approach in SimFix consists of - an offline mining stage, an online repairing stage. SimFix uses the Ochiai algorithm to localize fault which produces an ordered list of suspicious statements. It extracts a set of fixing ingredients using syntactic features, such as Structure Similarity, Variable Name Similarity, Method Name Similarity. Then it matches variables in the faulty and fixes ingredients and builds a variable mapping table for code adaptation. It calculates the difference between faulty and fixing ingredients and extracts concrete modification. Finally, it generates a list of patches and validates them using the test suite. These three features obtain low precisions 63.41%, 33.33%, and 60.7%, respectively.

ssFix [20] uses the syntactic features to calculate the similarity between faulty code and fixing ingredients. It uses the TF-IDF model to calculate the similarity score. They evaluated their technique on 357 bugs in the Defect4j [21] bug dataset and successfully solved 20 bugs from that dataset.

CapGen [22] uses semantic features to prioritize patches. It works on the fine-grained level to extract fixing ingredients. They selected 30 augmented mutation operators to generate operation space. For example: insert Simple Name under Method Invocation or delete Expression Statement under Method Declaration etc. CapGen uses three models to calculate the similarity score between faulty nodes and fixing ingredients. They are Genealogy Context, Accessed Variable, Semantic Dependency. CapGen combines the three model scores to calculate the final score. Based on that score, it prioritizes the patches and tests the patches which are more likely to be the fix for the bug. This approach achieves a precision of 84%.

A study [17] showed that combining syntactic and semantic features for patch prioritization can work better than using syntactic and semantic features alone. In the study, they took expression level bug and used replacement mutation operation for patch generation. To calculate the similarity between faulty code and fixing ingredients, it used Genealogical & variable similarity as semantic metrics and Normalized Longest Common Subsequence (Normalized LCS) as syntactic metrics. The approach was evaluated using 22 bugs from the IntroClassJava benchmark [23], fixed by CapGen applying replacement mutation [22]. It repaired all 22 bugs and achieved a precision of 100%.

III. METHODOLOGY

A recent study [5] shows that combining both syntactic and semantic similarity can improve patch prioritization. It works at *Expression* level and uses *Replacement* mutation operators to generate the patches. In this research, we evaluate the technique further to observe its impact on patch prioritization while using the *Insertion* as the mutation operation. The approach works at the *Expression* and *Expression-Statement* level as the *Insertion* mutation operation works better at these level [22]. It manipulates the source codes representing them in Abstract Syntax Tree (AST). Figure 1 shows some sample bug fixes using *Insertion* Mutation Operation. In figure 1a, the bug is fixed by inserting a *Simple Name* under the faulty *Method Invocation*. In figure 1c, the bug is fixed inserting a *Method Invocation* under the *If-Statement*.

The approach in this paper takes a source program and a test suite (positive and negative test cases) as input and generates some repaired variants. For generating patches, it uses *Insertion* mutation operation using *Expression* and *Expression-Statement* level code for fixing elements. It uses Cosine Similarity, Jaccard Similarity Index, Normalized Edit Distance, Normalized LCS as a syntactic feature, and Genealogical similarity and Anti-Pattern as a semantic feature. The repaired variants are thus tested with the test suite and the correct patches for the bug are validated. The approach works in the four following steps. The steps are quite similar to the approach of the previous study [5].

A. Fault Localization

This step identifies the faulty code using Ochiai [26] algorithm, a spectrum-based fault localization technique. It takes help from the GZoltar [25] tool for this purpose. GZoltar takes a source code and a test suite (including positive and negative test cases) and generates a list of the suspicious statements. For *Insertion* mutation operation, the approach identifies the faulty AST node of *Expression* and *Expression Statement*. It outputs a line number-wise suspicious value (probability of being a faulty node) of a program. If a node spans across multiple

```
//faulty code
72: - return solve (min, max);

// fixing ingredient
72: - return solve (f, min, max);
```

(a) A fixed bug (Math 70 from Defects4J) using *Insertion* mutation operation
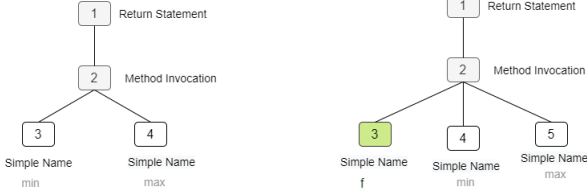
(b) AST difference of the fixed bug 1a

```
421: if (escapingOn && c[start] == QUOTE)  // Buggy statement
422: + next(pos); // Fixed statement
423: return appendTo == null ? null : appendTo.append(c);

// Fixing ingredient for the bug
170: if (c[pos.getIndex()] == START) {
171: format = parse(pattern, next(pos));
```

(c) A Fixed Bug (Lang 43 in Defects4J) Using the *Insertion* Mutation Operation and Fix Extracted From the Same File

Fig. 1: Sample Bug Fixes using Insertion Mutation Operation

lines, it is assigned the average of the suspicious values of those lines.

B. Patch Generation

In this step, the approach generates patches inserting fixing ingredients under the faulty node. For faulty nodes, it takes *Expression* and *Expression-Statement* level bug. Ambiguity arises while inserting a node under a block statement. This ambiguity can be addressed as *fix location selection* problem. To avoid ambiguity for fix location, it further splits the insertion operation into two more categories:

- Insert Before: Insert a subtree before node *N*.
- Insert After : Insert a subtree after node *N*.

The operation space induced by *Insertion* mutation operator is huge. To reduce this space, we use the top 10 augmented insertion mutation operations from *CapGen* [22].

C. Patch Prioritization

Several patches are produced at the patch generation level. To minimize the bug fixing time and cost, patches are prioritized. It also helps to maximize the precision of patches. For prioritizing patches, the genealogical similarity between is used to measure semantic similarity, and normalized Longest Common Subsequence (LCS) and Anti - Pattern [6] and Cosine Similarity is used to measure syntactic similarity.

- **Genealogical Similarity:** The genealogical structure of an AST node shows a node is frequently used under and together with which types of code elements by checking its ancestor & sibling node [22]. Traversal is done from the node to its ancestor until a method declaration is

found. And nodes of category Statement and Expression within the same Block are extracted for sibling nodes [22]. The Genealogical Similarity is measured as

$$gen\_s(f_n, f_e) = \frac{\sum_{t \epsilon k} \min(f_n(t), f_e(t))}{\sum_{t \epsilon k} f_n(t)} \quad (1)$$

Here, f_n and f_e are frequencies of different node types for faulty node and fixing ingredient respectively. K indicates a set of all distinct AST node types captured by f_n

- **Normalized LCS:** LCS measures the Longest Common Subsequences between two sets of sequence. To measure the LCS, we take string representation of faulty node (f_n) and fixing ingredient(f_e). Then the value of the LCS is normalized to get the value in the interval [0,1].

$$n\_lcs((f_n, f_e)) = \frac{LCS(f_n, f_e)}{\max(f_n, f_e)} \quad (2)$$

- **Cosine Similarity:** The cosine similarity of two ASTs representative vectors expresses the similarity between them. The cosine similarity measure uses two finite-dimensional vectors of the same dimension in which each vector represents a document. Given two string f_n (string representation of faulty node) and f_e (string representation of fixing ingredient), we construct the collection between the two elements. The collection of terms denotes C = $\{c_1, c_2,..., c_n\}$ with $c_i \in f_n \mid c_i \in f_e$ and each c_i is distinct. The strings are then represented in an n-dimensional vectors $\vec{V}_{f_n}$ and $\vec{V}_{f_e}$.

$$cos\_s(f_n, f_e) = \frac{\vec{V}_{f_n} \cdot \vec{V}_{f_e}}{(|\vec{V}_{f_n}| * |\vec{V}_{f_e}|)} \quad (3)$$

- **Normalized Edit Distance**: Edit distance refers to the minimum number of edit operations needed to convert from one string to another string. For measuring the score, we use an alternative version of Levenshtein distance where each Insertion or Deletion cost is 1 and Substitution cost is 2. The score is then normalized to get the score between the interval [0,1]. Finally, the score is subtracted from 1 as the approach in this paper only considers the similarities between the faulty nodes and fixing ingredients.

$$n\_ed(f_n, f_e) = 1 - \frac{Edit\_Distance(f_n, f_e)}{\max(f_n, f_e)} \quad (4)$$

Here, f_n and f_e is the string representation of faulty node and fixing ingredient.

- **Jaccard Similarity Index:** The technique computes Jaccard distance between string representation of faulty node (f_n) and fixing node (f_e). The input string is converted into a set of N-grams. In Comparing with Cosine similarity, Jaccard Similarity Index takes the number of common attributes is divided by the number of attributes that exist

in at least one of the two objects. In comparison, Cosine similarity divides the number of the common attributes by the dot product of two objects represented in vectors.

$$jac\_s(f_n, f_e) = \frac{|f_n \cap f_e|}{|f_n \cup f_e|} \tag{5}$$

Here, f_n and f_e is the string representation of faulty node and fixing ingredient.

- **Anti-Pattern:** Anti-patterns capture disallowed modifications to the buggy program [24]. If such a modification passes all tests in the given test-suite, they are not counted as repairs [24]. Because those modifications may later introduce some code smell in the program. We filter out such modifications as part of patch prioritization. We take account of the following anti-pattern [24] for Insertion mutation operation.
  - *Anti-append Early Exit*: This pattern disallows the insertion of return statements at any location except for after the last statement.

The utility of anti-pattern is not directly used with the semantic-based counterparts. Rather it is used to filter out the trivial solutions.

After calculating the syntactic and semantic features, they are then combined to calculate the final score. We keep Genealogical Similarity and Normalized LCS fixed and used Cosine Similarity, Normalized Edit Distance, Jaccard Similarity Index one at a time. So, we evaluated total 3 combinations of features. The first one was **Com-CS**, the combination of Genealogical Similarity, Normalized LCS and Cosine Similarity. The second one was **Com-NED**, the combination of Genealogical Similarity, Normalized LCS and Normalized Edit Distance. And the final one was **Com-JS**, the combination of Genealogical Similarity, Normalized LCS and Jaccard Similarity Index.

We calculate the score of the patches in the following way:

$$patch\_score = gen\_s(f_n, f_e) + n\_lcs(f_n, f_e) + simi(f_n, f_e) \tag{6}$$

Here, $simi(f_e, f_n)$ stands for Cosine Similarity, Normalized Edit Distance, Jaccard Similarity Index, and Jaccard Similarity Index which we plugged into the equation one at a time.

D. Patch Validation

Patch validation validates the patch that is the correct repair of the faulty node. This step executes negative and positive test cases for validating the patches.

IV. EXPERIMENT

We implemented the technique addressed in this paper in Java. It uses the Eclipse JDT parser for manipulating AST. It uses the GZoltar [25] tool to localize the faulty codes. GZoltar tool is used with Ochiai algorithm [26] to calculate the suspicious value of the *Expression* and *Expression-Statement*

for being a faulty node. The pseudocode of our technique is given in 1.

Algorithm 1: High-level pseudocode for our technique.

Input : Program P to be repaired; Set of positive test cases, *PosT*; Set of negative test cases, *NegT*; Set of augmented mutation operation, *AugOp*.

Output: Repaired program variant

1   $RepairedProgramVariants \leftarrow \emptyset$
2   $\phi \leftarrow localize\_fault(P, PosT, NegT)$
3   **forall** $faultyNode \in \phi$ **do**
4     $\theta \leftarrow getFixingElement()$
5     $N \leftarrow getParent(faultyNode)$
6   **forall** $fixingIngredient \in \theta$ **do**
7       $OP \leftarrow getMutationOperation(N, fixingIngredient)$
8   **end**
9   **end**
10   **forall** $fixingIngredient \in \theta$ & $OP \in AugOp$ **do**
11     $syntacticSimi \leftarrow calculateSyntacticSimi(faultyNode, fixingIngredient, OP)$
12     $semanticSimi \leftarrow calculateSemanticSimi(faultyNode, fixingIngredient, OP)$
13     $finalScore \leftarrow combine(syntacticSimi, semanticSimi)$
14     $candidatePatches \leftarrow \{faultyNode, fixingIngredient, finalScore\}$
15   **end**
16   $sortedCandidatePatches \leftarrow sort(candidatePatches)$
17   **forall** $patch \in sortedCandidatePatches$ **do**
18     $programVariant \leftarrow insert(faultyNode, fixingIngredient, AugOp)$
19     $RepairedProgramVariants \leftarrow validate(programVariant, PosT, NegT)$
20   **end**
21   **return** RepairedProgramVariants

A. Research Question

As we discussed in the earlier section, the existing patch prioritization technique can be categorized into 3 broad categories. Recent work [5] introduced the technique for combining syntactic and semantic similarity for patch prioritization, worked at the *Expression* level bug and used the *Replacement* mutation operation to generate patches. In this study, we evaluate the technique of combining syntactic and semantic similarity to observe its impact on patch prioritization using the *Insertion* mutation operation. Our evaluation aims to answer the following research questions:

- **RQ1:** What is the impact of combining syntactic and semantic similarity while using *Insertion* mutation operation?
- **RQ2:** How different syntactic and semantic similarity features impact patch prioritization?

B. Evaluation

We evaluated our technique in IntroClassJava Benchmark [23]. The benchmark contains 297 bugs from 6 projects. For evaluation, the following are examined:

- **Median Rank of the Correct Patch:** The lower the median rank, the better the approach is [10].
- **Precision:** If a technique can rank the correct solution before the incorrect plausible one, it achieves higher precision. [22]
- **Rank of the first correct solution:** The approach that ranks the first correct solution higher, considered as more efficient. [10].

C. Experiment Settings

To understand the impact of similarities, the approach need to compared with patch prioritization techniques that use semantic and syntactic similarity individually. Therefore, this study further implements semantic or syntantic similarity based patch prioritization approaches using metrics discussed in Section III-C. The techniques are described below:

- **Semantic Similarity based Approach (SSBA):** It uses only semantic similarity metrics namely genealogical to prioritize patches. Also the anti-pattern is also used to filter out the incorrect patches.
- **LCS based Approach (LBA):** It is a syntactic similarity based approach that prioritizes patches using only normalized LCS score.
- **Cosine Similarity based Approach (CSBA):** It is another syntactic similarity based approach that prioritizes patches using only normalized Cosine score.
- **Jaccard Similarity Index based Approach (JSBA):** It is also a syntactic similarity based approach that uses only Jaccard Similarity Index score
- **Normalized Edit Distance based Approach (NBA):** It is also another syntactic similarity based approach that prioritizes patches using only normalized Edit Distance score.

All of these five patch prioritization approaches follow the same repairing process as the combined ones. It ensures that the observed effects occurred due to varied similarities used in patch prioritization.

D. Result Analysis

The approach repairs 6 bugs from IntroClassJava benchmark [23]. For the repaired bugs, no plausible patch is generated before the first correct solution. Thus, the approach achieves a precision of 100%. The implementation is publicly available here.

In our study, we were able to repair 6 bugs from the IntroClassJava benchmark. The results are shown in the table I

Figure 2 shows the first correct patch rank of different approach. The lower the rank is, the better the approach. Figure 3 shows the rank distribution of Com-CS, Com-NED, Com-JS, SSBA, LBA, CSBA, JSBA, and NBA. Log transformation
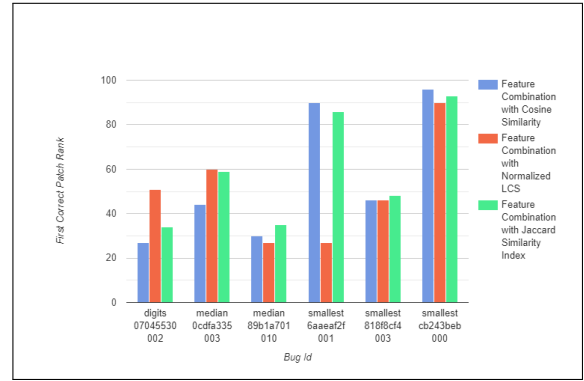

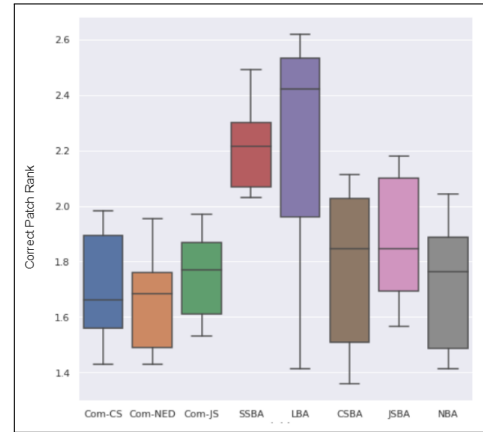Fig. 2: Impact of different combination of features on patch rank

Fig. 3: Comparison of Correct Patch Rank among various Approach

is used in this figure as the data range is high. In terms of median rank, Com-CS, Com-NED, and Com-JS outperform SSBA, LBA, CSBA, JSBA and NBA. The ranks are 45, 48.5, 53.5, 163.5, 177, 55, 55, 58.5 respectively. The reason is combination of features incorporate information from multiple domains (both textual similarity and code meaning). The correctness of the generated patches are checked using the publicly available results of CapGen [22] except for the bug *smallest_cb243beb_000*. CapGen doesn't contain the fix for that bug. The study [6] that shows that combining both semantic and syntactic similarities works better than the syntactic or semantic similarities alone, was able to fix 22 bugs using Replacement Mutation Operation. In this paper we evaluate the impact if we use Insertion Mutation Operation. So, in total 25 bugs have been solved from the *IntroClassJava* benchmark [23] using the approach of combining syntactic and semantic similarities.

V. CONCLUSION

In this paper, a patch prioritization technique is evaluated for combining syntactic and semantic similarity for automated program repair. Our approach uses different combinations of syntactic and semantic matrices to see their impact on patch

Bug Id	With Cosine Similarity		With Normalized Edit Distance		With Jaccard Similarity Index	
	Total Patches	Correct Patch Rank	Total Patches	Correct Patch Rank	Total Patches	Correct Patch Rank
digits_07045530_002.java	57	27	78	51	57	34
median_0cdfa335_003.java	137	44,68	170	60	137	59, 63
median_89b1a701_010.java	145	30	185	27	145	35
smallest_6aaeaf2f_001.java	197	90	226	27	197	86
smallest_818f8cf4_003.java	199	46	178	46	199	48
smallest_cb243beb_000.java	168	96	189	90	168	93

TABLE I: Results of the Approach of Combining Features on IntroClassJava Benchmark

Bug Id	SSBA		LBA		CSBA		JSBA		NBA	
	TP	CPR	TP	CPR	TP	CPR	TP	CPR	TP	CPR
digits_07045530_002.java	326	107	357	342	58	23	58	42	82	67
median_0cdfa335_003.java	652	110, 120	682	287, 340	138	42, 92	138	52, 85	175	81
median_89b1a701_010.java	660	299	645	26	146	26	146	37	189	26
smallest_6aaeaf2f_001.java	817	170	870	113	198	130	198	140, 151	233	26
smallest_818f8cf4_003.java	779	157, 175	851	48, 414	200	70	200	58	181	50
smallest_cb243beb_000.java	589	310	631	241	168	124	168	121	190	110

TABLE II: Results of the Approach using Only Semantic or Syntactic Features Alone on IntroClassJava Benchmark (TP = Total Patches, CPR = Correct Patch Ranks)

prioritization while using the Insertion mutation operation for patch generation. It uses *Genealogical Similarity*, *Normalized LCS*, *Cosine Similarity*, *Jaccard Similarity Index*, *Normalized Edit Distance* to calculate the context similarity score. It also considers *Anti-Patterns* to filter out the incorrect plausible patches. The approach observes that *Cosine Similarity* and *Normalized Edit Distance* along with *Genealogical Similarity* and *Normalized LCS* rank the correct patch higher. This approach has been able to solve 6 bugs from IntroClassJava benchmark [23]. It achieves a precision of 100% for solving those bugs.

REFERENCES

[1] Tom Britton, Lisa Jeng, Graham Carver, Paul Cheak, and Tomer Katzenellenbogen. 2013. Reversible debugging software. Judge Bus. School, Univ. Cambridge, Cambridge, UK, Tech. Rep (2013).

[2] M. Monperrus, "Automatic software repair: a bibliography," ACM Computing Surveys (CSUR), vol. 51, no. 1, p. 17, 2018.

[3] E. T. Barr, Y. Brun, P. Devanbu, M. Harman, and F. Sarro, "The plastic surgery hypothesis," in ESEC/FSE, 2014, pp. 306–317.

[4] S. Sumi, Y. Higo, K. Hotta, and S. Kusumoto, "Toward improving graftability on automated program repair," in ICSME, 2015, pp. 511–515.

[5] M. Asad, K. K. Ganguly and K. Sakib, "Impact Analysis of Syntactic and Semantic Similarities on Patch Prioritization in Automated Program Repair," 2019 IEEE International Conference on Software Maintenance and Evolution (ICSME), 2019, pp. 328-332, doi: 10.1109/IC-SME.2019.00050.

[6] Z. Chen and M. Monperrus, "The remarkable role of similarity in redundancy-based program repair," Computing Research Repository (CoRR), vol. abs/1811.05703, 2018.

[7] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "Genprog: A generic method for automatic software repair," IEEE Transactions on Software Engineering (TSE), vol. 38, no. 1, pp. 54 –72, 2012

[8] Y. Pei, Y. Wei, C. Furia, M. Nordio, and B. Meyer, "Code-based automated program fixing," in International Conference on Automated Software Engineering (ASE), 2011, pp. 392 –395.

[9] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. 2013. Automatic patch generation learned from human-written patches. In ICSE'2013. IEEE Press, 802–811.

[10] Xuan Bach D Le, David Lo, and Claire Le Goues. 2016. History Driven Program Repair. In SANER'2016, Vol. 1. IEEE, 213–224.

[11] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. SemFix: program repair via semantic analysis. In ICSE'2013. IEEE Press, 772–781.

[12] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. 2009. Automatically finding patches using genetic programming. In ICSE'2009. IEEE Computer Society, 364–374.

[13] V. Debroy and W. Wong, "Using mutation to automatically suggest fixes for faulty programs," in International Conference on Software Testing, Verification and Validation (ICST), 2010, pp. 65 –74.

[14] D Le Xuan-Bach, Chu Duc-Hiep, Lo David, Goues Claire, Le, and Willem Visser. 2017. S3: Syntax- and Semantic-Guided Repair Synthesis via Programming by Examples. In FSE'2017. ACM, to appear.

[15] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. 2012. A systematic study of automated program repair: Fixing 55 out of 105 bugs for 8 each. In ICSE'2012. IEEE, 3–13.

[16] Westley Weimer, Zachary P Fry, and Stephanie Forrest. 2013. Leveraging program equivalence for adaptive program repair: Models and first results. In ASE'2013. IEEE, 356–366.

[17] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "Genprog: A generic method for automatic software repair," IEEE Transactions on Software Engineering (TSE), vol. 38, no. 1, pp. 54 –72, 2012

[18] R. K. Saha, Y. Lyu, H. Yoshida, and M. R. Prasad, "Elixir: Effective object-oriented program repair," in Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 648–659, IEEE, 2017.

[19] J. Jiang, Y. Xiong, H. Zhang, Q. Gao, and X. Chen, "Shaping program repair space with existing patches and similar code," in Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA), pp. 298–309, ACM, 2018.

[20] Q. Xin and S. P. Reiss, "Leveraging syntax-related code for automated program repair," in Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 660–670, IEEE, 2017.

[21] R. Just, D. Jalali, and M. D. Ernst, "Defects4J: A database of existing faults to enable controlled testing studies for Java programs," in ISSTA, 2014, pp. 437–440.

[22] M. Wen, J. Chen, R. Wu, D. Hao, and S.-C. Cheung, "Context-aware patch generation for better automated program repair," in Proceedings of the 40th International Conference on Software Engineering (ICSE), pp. 1–11, ACM, 2018.

[23] T. Durieux and M. Monperrus, IntroClassJava: A Benchmark of 297 Small and Buggy Java Programs. PhD thesis, Universite Lille 1, 2016.

[24] Shin Hwei Tan, Hiroaki Yoshida, Mukul R Prasad, and Abhik Roychoudhury. 2016. Anti-patterns in search-based program repair. In International Symposium on Foundations of Software Engineering. ACM, 727–738.

[25] J. Campos, A. Riboira, A. Perez, and R. Abreu, "GZoltar: an eclipse plug-in for testing and debugging," in ASE, 2012, pp. 378–381.

[26] R. Abreu, P. Zoeteweij, and A. J. C. v. Gemund, "An evaluation of similarity coefficients for software fault localization," in Proceedings of the 12th Pacific Rim International Symposium on Dependable Computing, ser. PRDC '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 39–46.