Impact of Combining Syntactic and Semantic Similarities on Patch Prioritization

Moumita Asad, Kishan Kumar Ganguly and Kazi Sakib
Institute of Information Technology, University of Dhaka, Dhaka, Bangladesh

Keywords: Patch Prioritization, Semantic Similarity, Syntactic Similarity, Automated Program Repair.

Abstract: Patch prioritization means sorting candidate patches based on the probability of correctness. It helps to minimize the bug fixing time and maximize the precision of an automated program repair technique by ranking the correct solution before incorrect one. Recent program repair approaches have used either syntactic or semantic similarity between faulty code and fixing ingredient to prioritize patches. However, the impact of combined approach on patch prioritization has not been analyzed yet. For this purpose, two patch prioritization methods are proposed in this paper. Genealogical and variable similarity are used to measure semantic similarity since these are good at differentiating between correct and incorrect patches. Two popular metrics namely normalized longest common subsequence and token similarity are considered individually for capturing syntactic similarity. To observe the combined impact of similarities, the proposed approaches are compared with patch prioritization techniques that use either semantic or syntactic similarity. For comparison, 246 replacement mutation bugs from historical bug fixes dataset are used. Both methods outperform semantic and syntactic similarity based approaches, in terms of median rank of the correct patch and search space reduction. In 11.79% and 10.16% cases, the combined approaches rank the correct solution at first position.

1 INTRODUCTION

Patch is the modifications applied to a program for fixing a bug. Automated program repair finds the correct patch based on a specification, e.g., test cases (Monperrus, 2018). It works in three steps namely fault localization, patch generation and patch validation (Tan and Roychoudhury, 2015), (Liu et al., 2019a). Fault localization identifies the faulty code where the bug resides. Patch generation modifies the faulty code to fix the bug. Finally, patch validation examines whether the bug has been fixed or not. Since the solution space is infinite, numerous patches can be generated (Jiang et al., 2018). In addition, a plausible solution - patch that passes all the test cases, can be incorrect. It is known as overfitting problem (Wen et al., 2018). To limit the search space, most of the program repair techniques such as (Le Goues et al., 2012), (Kim et al., 2013), (Qi et al., 2014), rely on redundancy assumption (Chen and Monperrus, 2018).

According to the redundancy assumption, the solution of a bug can be found elsewhere in the application or other projects (Chen and Monperrus, 2018), (White et al., 2019). This assumption has already been validated by existing studies such as (Martinez et al., 2014), (Barr et al., 2014). Martinez et al. found that 3-17% of the commits are redundant at the line level, whereas it is 29-52% at token level (Martinez et al., 2014). Another study on 15,723 commits reported that approximately 30% fixing ingredients (code used to fix the bug) exist in the same buggy file (Barr et al., 2014). Although the redundancy assumption limits the search space, in practice it is too large for exploring exhaustively (Chen and Monperrus, 2018). For example, if a technique collects line wise fixing ingredients at application level, the number of patches generated will be total LOC of the application. Therefore, potentially correct patches need to be validated earlier.

Sorting candidate patches, based on its probability of correctness, is called patch prioritization (Xiong et al., 2018). It can help to minimize the bug fixing time and maximize the precision of a repair technique. To prioritize patches, some information need to be considered such as similarity between faulty code and fixing ingredient or patterns derived from existing patches (Jiang et al., 2018). The information should be able to minimize overfitting problem by ranking the correct patch higher. Furthermore, it should validate the correct solution earlier since patch validation

Asad, M., Ganguly, K. and Sakib, K.
Impact of Combining Syntactic and Semantic Similarities on Patch Prioritization.
DOI: 10.5220/0009411301700180
In *Proceedings of the 15th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2020)*, pages 170-180
ISBN: 978-989-758-421-3

is a time-consuming task (Saha et al., 2017), (Chen and Monperrus, 2018).

To the best of the knowledge, existing approaches use either syntactic or semantic similarity between faulty code and fixing ingredient for patch prioritization (Saha et al., 2017), (Xin and Reiss, 2017b), (Jiang et al., 2018), (Wen et al., 2018). Elixir uses contextual and bug report similarities for capturing syntactic similarity (Saha et al., 2017). To calculate syntax-similarity score, ssFix uses TF-IDF model (Xin and Reiss, 2017b). For finding top fixing ingredients, syntactically similar to the faulty code, SimFix uses three metrics - structure, variable name and method name similarity (Jiang et al., 2018). CapGen uses three models based on genealogical structures (e.g., ancestors of an Abstract Syntax Tree (AST) node), accessed variables and semantic dependencies to measure semantic similarity (Wen et al., 2018). However, none of these approaches analyze the impact of integrating the strengths of both similarities to prioritize patches.

This paper presents an empirical study on patch prioritization to analyze the impact of combining syntactic and semantic similarities. It extends the author's initial work (Asad et al., 2019). Similar to (Asad et al., 2019), it uses genealogical and variable similarity between faulty node and fixing ingredient for measuring semantic similarity. However, it is found that syntactic similarity metric named normalized longest common subsequence does not perform well when the character level difference between faulty code and fixing ingredient is high (Asad et al., 2019). Therefore, to calculate syntactic similarity, normalized longest common subsequence as well as a new metric named token similarity are considered separately. Thus, two patch prioritization approaches namely *Com-L* and *Com-T* are proposed respectively. Genealogical similarity checks whether faulty node and fixing ingredient are frequently used with same type of code elements (e.g., inside *if* statement) (Wen et al., 2018). Variable similarity inspects the name and type of variables accessed by the faulty node and fixing ingredient. Normalized longest common subsequence calculates maximum similarity at character level. Token similarity measures to what extent same tokens (e.g., *identifiers*) exist in faulty node and fixing ingredient, regardless of its position.

For analysis, 246 out of 3302 bugs from historical bug fixes dataset are selected through preprocessing (e.g., removing duplicate bugs) (Le et al., 2016). These bugs are collected from over 700 large, open-source, popular Java projects such as Apache Commons Math, Eclipse JDT Core. Each bug is associated with a buggy and a fixed version file, corresponding commit hashes and project url. From the difference between the buggy and fixed version files, the faulty line is identified. Next, AST nodes of type *Expression* are extracted from that line. This work focuses on expression level since it increases the probability of including the correct patch in the search space (Wen et al., 2018). To generate patches, the faulty nodes are replaced with fixing ingredients. Following other repair techniques (Le Goues et al., 2012), (Wen et al., 2018), the fixing ingredients are collected from the source file where the bug resides. Lastly, patches are prioritized based on the proposed techniques. Similar to (Chen and Monperrus, 2018), ASTs of each patch and the correct solution are matched to assess its correctness. The correct patches of the bugs are provided with the dataset.

For evaluation, the combined methods are compared to techniques using only syntactic or semantic similarity. For comparison, three metrics namely median rank of the correct patch, average search space reduction and perfect repair (the percentage of bug fixes for which the correct solution is ranked at first position) are inspected as well as Wilcoxon Signed-Rank test is conducted. Results show that the proposed approaches outperform syntactic or semantic similarity based techniques in terms of median rank of the correct patch and average search space reduction. Using *Com-L* and *Com-T*, 96.52% and 96.62% of the total search space can be avoided to find the correct patch. These methods can rank the correct patch significantly higher than semantic or syntactic similarity based approaches. The mean rank of the correct patch is significantly better in *Com-T* than *Com-L*. Furthermore, *Com-L* and *Com-T* rank the correct patch at first position in 11.79% and 10.16% cases respectively. It indicates that these methods are capable of ranking correct patch prior to incorrect plausible ones.

2 RELATED WORK

Recently, automated program repair has drawn the attention of researchers due to its potentiality of minimizing debugging effort (Gazzola et al., 2017). Most of the earlier approaches (Le Goues et al., 2011), (Le Goues et al., 2012), (Kim et al., 2013), (Qi et al., 2014) rely on redundancy assumption to limit the number of generated patches. GenProg (Le Goues et al., 2012) and PAR (Kim et al., 2013) use genetic programming to find the correct patch. GenProg randomly modifies the faulty code using three mutation operators (insert, replace, delete) (Le Goues et al., 2012). On the other hand, PAR uses ten predefined templates (e.g., null pointer checker) to generate patches (Kim et al., 2013). These templates are extracted from manually inspecting 62,656 human-written patches. Another approach RSRepair randomly searches among the candidate patches to find the correct one (Qi et al., 2014).

Although the redundancy assumption has limited the number of generated patches, in practice it is too large for exploring exhaustively (Chen and Monperrus, 2018). All the patches need to be compiled and validated by executing test cases, which is time consuming (Saha et al., 2017), (Chen and Monperrus, 2018). Therefore, new techniques are prioritizing patches to validate potentially correct patches earlier (Le et al., 2016), (Saha et al., 2017), (Xin and Reiss, 2017b), (Jiang et al., 2018), (Wen et al., 2018).

Approaches using patch prioritization can be broadly divided into two categories based on the information used. The first category uses historical bug-fix patterns such as HDRepair (Le et al., 2016). HDRepair uses patterns obtained from 3000 bugs fixes over 700 large, popular GitHub projects. To generate patches, it uses 12 mutation operators such as replace statement, boolean negation. If a patch passes all the test cases, it is added to a set of possible solutions. Lastly, a predefined number of patches, ranked by their frequency in the historical bug-fixes, are presented to the developer. Although HDRepair solves more bugs than prior techniques GenProg (Le Goues et al., 2012), PAR (Kim et al., 2013), it obtains low precision (56.50%). It indicates that considering only historical bug fix patterns is not sufficient for eliminating incorrect plausible patches (Wen et al., 2018).

The second category uses similarity between faulty code and fixing ingredient to prioritize patches. This category can further be classified into two groups based on the type of similarity used (syntactic or semantic similarity). ELIXIR (Saha et al., 2017), ssFix (Xin and Reiss, 2017b), SimFix (Jiang et al., 2018) use syntactic similarity between faulty code and fixing ingredient. Syntactic similarity focuses on textual alikeness such as similarity in variable names. On the other hand, CapGen (Wen et al., 2018) uses semantic similarity between faulty code and fixing ingredient. It focuses on code meaning such as data type of variables (Nguyen et al., 2013).

Elixir, one of the first such approaches, introduces 8 templates for generating patches, e.g., changing Expression in return statement (Saha et al., 2017). It uses four features including contextual and bug report similarities to prioritize patches. Contextual similarity measures the syntactic similarity between fixing ingredient and surrounding code of the faulty location. Bug report similarity calculates the syntactic similarity between fixing ingredient and bug report. For assigning different weights to these similarities, logistic regression model is used. The approach validates only the top 50 patches generated from each template. It can repair more bugs compared to contemporary techniques (Xin and Reiss, 2017b), (Xiong et al., 2017). However, it yields low precision particularly 63.41%.

ssFix is the first approach to perform syntactic code search from a codebase containing the faulty program and other projects (Xin and Reiss, 2017b). At first, ssFix extracts the faulty code along with its context (code surronding the faulty location) using a LOC based algorithm. It is called target chunk (*tchunk*). A similar process is followed to retrieve fixing ingredients and their contexts from the codebase. These are called candidate chunks (*cchunks*). The *tchunk* and *cchunks* are tokenized after masking project-specific code (e.g., variable names). Next, *cchunks* are prioritized based on its syntax-relatedness to the tchunk, calculated using TF-IDF. Currently, ssFix uses maximum 100 top *cchunks* for generating patches. This approach obtains 33.33% precision, which indicates it is not good at differentiating between correct and incorrect patches.

SimFix uses three metrics - structure, variable name and method name similarity to capture syntactic similarity between faulty code and fixing ingredient (Jiang et al., 2018). Structure similarity extracts a list of features related to AST nodes (e.g., number of *if* statements). Variable name similarity tokenizes variable names (e.g., splitting studentID into student and ID) and calculates similarity using Dice coefficient (Thada and Jaglan, 2013). Method name similarity follows the same process as variable name similarity. To generate patches, SimFix selects top 100 fixing ingredients based on the similarity score. To further limit the search space, only fixing ingredients found frequently in existing human-written patches are considered. Similar to Elixir and ssFix, this approach yields low precision which is 60.70%.

To generate patches, CapGen defines 30 mutation operators such as insert *Expression* statement under *if* statement (Wen et al., 2018). It uses three models based on genealogical structures, accessed variables and semantic dependencies to capture context similarities at AST node level. These models mainly focus on semantic similarities between faulty code and fixing element to prioritize patches. The precision of this approach is higher (84.00%), however, it relies on program dependency graph to calculate semantic dependency which does not scale to even moderate-size programs (Gabel et al., 2008).

The above discussion indicates that various approaches have used either syntactic or semantic similarity as a part of the repairing process. However, techniques using syntactic similarity such as Elixir, ssFix, yields low precision. On the other hand, some semantic similarity metrics such as semantic dependency, suffer from scalability problem. Although both of these similarities have limitations, these are effective in program repair. Nevertheless, the impact of combining the strengths of both similarities to prioritize patches has not been explored yet.

3 METHODOLOGY

This study considers finding the correct patch in automated program repair as a ranking problem. It takes source code and faulty line as input and outputs a ranked list of patches. For generating the ranked list, patches are sorted using a combination of syntactic and semantic similarity.

3.1 Dataset Preprocessing

In this paper, historical bug fixes dataset is used which comprises more than 3000 real bug fixes from over 700 large, popular, open-source Java projects such as Apache Commons Lang, Eclipse JDT Core etc (Le et al., 2016). This dataset has been adopted by previous studies as well (Le et al., 2017), (Wen et al., 2018). Each bug in this dataset is associated with a buggy and a fixed version file, corresponding commit hashes and project url.

This study analyzes how combining syntactic and semantic similarity impacts patch prioritization. Similar to state of the art approaches (Le Goues et al., 2012), (Kim et al., 2013), (Le et al., 2016), (Wen et al., 2018), it focuses on redundancy based program repair. It particularly studies replacement mutaion bugs, as followed in (Chen and Monperrus, 2018). For this purpose, bugs that fulfill following criteria, are selected from the dataset.

- **Unique:** Duplicate bugs will bias the result. Two bugs are considered as duplicate if their corresponding buggy and fixed version files are same. The dataset contains some duplicate bugs (e.g., bug *Lollipop_platform_frameworks_base_18* and *AICP_frameworks_base_24*), which are filtered out.
- **Satisfy Redundancy Assumption at File Level:** Similar to (Le Goues et al., 2012), (Wen et al., 2018), (Liu et al., 2019b), this study focuses on file level redundancy assumption (patches of bugs are found in the corresponding buggy files). Only bugs satisfying this requirement are chosen.
- **Fixed by Applying Replacement Mutation:** The faulty code is more likely to be syntactically and semantically similar to the fixing ingredient for replacement mutation bugs (Chen and Monperrus, 2018). Hence, only bugs that can be solved by applying replacement mutation are selected.
- **Require Fixing at Expression Level:** Existing study found that redundancy is higher at finer granularity and therefore, increases the probability of including the correct patch in the search space (Wen et al., 2018). Only bugs that require fix at expression level are selected.
- **Having Available Project and Dependency Files:** To measure similarity, variables within a file need to be identified. For this purpose, the corresponding project and dependency files of a bug are needed. However, some project urls (e.g., *apache_james* project) do not exist anymore. Additionally, some bugs have missing dependency files (e.g., bug *baasbox_baasbox_6*), which are removed.

After filtering, it results in 246 replacement mutation bugs.

3.2 Approach

This paper analyzes the impact of combining syntactic and semantic similarities on patch prioritization. Figure 1 shows overview of the technique. It works in three steps namely fault localization, patch generation and patch prioritization. For a given buggy line, fault localization extracts corresponding *Expression* nodes. Patch generation produces patches by replacing the faulty node by fixing ingredients. For each patch, a score is calculated using syntactic and semantic similarity between faulty code and fixing ingredient. Based on this score, patches are prioritized. The details of these steps are given below:

1. **Fault Localization:** It identifies the faulty AST node of type *Expression*. Similar to (Chen and Monperrus, 2018), this study assumes that fault localization outputs the correct faulty line since the main focus is on patch prioritization. For each bug, the faulty line is identified from the difference between the buggy and fixed version files. Next, *Expression* type nodes (both buggy and non-buggy) residing in that line are extracted from AST. Figure 2 shows a sample bug *fasseg_exp4j_4* from project exp4j. Here, line 68 is faulty. All the expressions from this line such as *Character*, *Character.isDigit(next)*, *next* etc, are extracted.

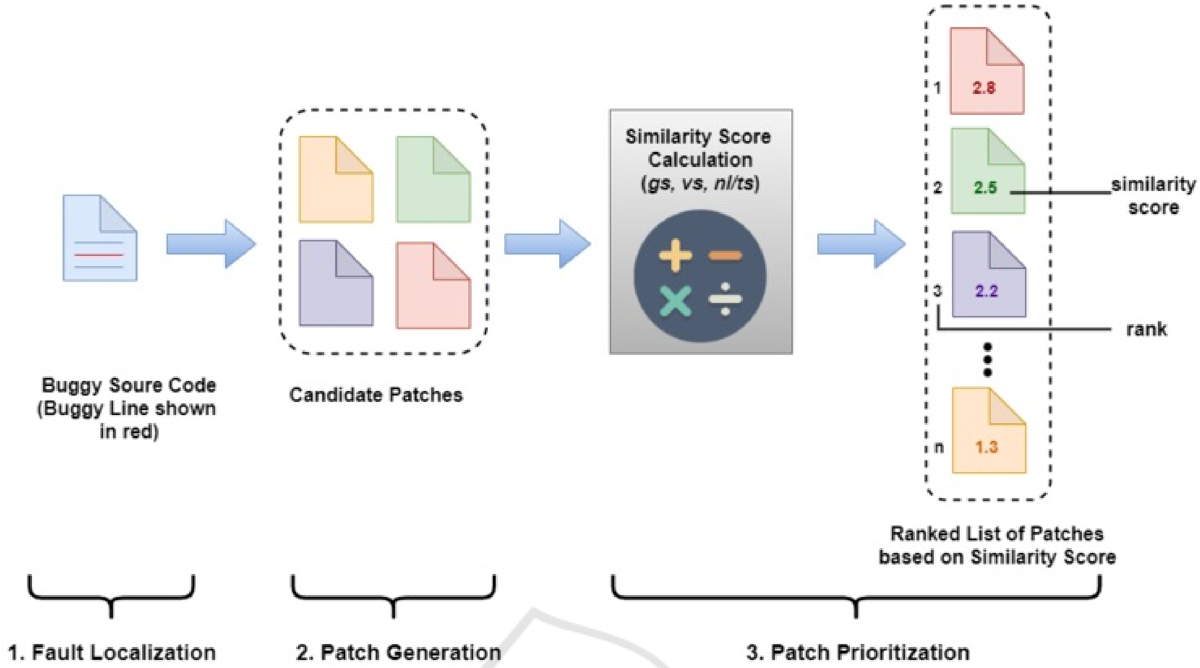

Figure 1: Overview of the Technique.

2. **Patch Generation:** This step modifies the source code to generate patches. After identifying the faulty nodes, fixing ingredients are collected. This study uses nodes from the buggy source file that have a category of *Expression* as fixing ingredients, as followed in (Wen et al., 2018). Next, faulty nodes are replaced with fixing ingredients for patch generation. In Figure 2, a sample patch is replacing *Character.isDigit(next)* with *Character.isDigit(next)* || *next* == *'.'* from line 93.

```
// buggy statement
68: -   if (Character.isDigit(next)) {
// fixed statement
68: +   if (Character.isDigit(next) || next == '.') {
// fixing ingredient
93:     if (Character.isDigit(next) || next == '.') {
```

Figure 2: Buggy Statement, Fixed Statement and Fixing Ingredient of Bug *fasseg_exp4j_4*.

3. **Patch Prioritization:** The patch generation step produces numerous patches due to having large solution space. To find potentially correct patches earlier, the generated patches are prioritized. Both syntactic and semantic similarities between faulty code and fixing ingredient are used to prioritize patches (shown in Figure 1). To measure semantic similarity, genealogical and variable similarity are used since these are effective in differentiating between correct and incorrect patches (Wen et al., 2018). For capturing syntactic similarity, two widely-used metrics namely normalized Longest Common Subsequence (LCS) and token similarity are considered individually (Ragkhitwetsagul et al., 2018). Thus, two patch prioritization approaches namely *Com-L* and *Com-T* are proposed. *Com-L* combines genealogical and variable similarity with normalized LCS to rank patches. *Com-T* uses combination of genealogical, variable and token similarity for patch prioritization.

- **Genealogical Similarity:** Genealogical structure indicates the types of code elements, with which a node is often used collaboratively (Wen et al., 2018). For example, node *Character.isDigit(next)* is used inside *if* statement. To extract the genealogy contexts of a node residing in a method body, it's ancestor, as well as, sibling nodes are inspected. The ancestors of a node are traversed until a method declaration is found. For sibling nodes, nodes having a type *Expressions* or *Statements* within the same block of the specified node are extracted. Next, the type of each node is checked and the frequency of different types of nodes (e.g., number of *for* statements) are stored. Nodes of type *Block* are not considered since these provide insignificant context information (Wen et al., 2018). On the other hand, for nodes appearing outside method body, only its respective type is stored. The same process is repeated for the faulty node (fn) and the fixing ingredient (fi). Lastly, the genealogical similarity (gs) is measured using (1).

$$gs(fn, fi) = \frac{\sum_{t \in K} min(\phi_{fn}(t), \phi_{fi}(t))}{\sum_{t \in K} \phi_{fn}(t)} \quad (1)$$

where, ϕ_{fn} and ϕ_{fi} denote the frequencies of different node types for faulty node and fixing ingredient respectively. *K* represents a set of all distinct AST node types captured by ϕ_{fn}.

- **Variable Similarity:** Variables (local variables, method parameters and class attributes) accessed by a node provide useful information as these are the primary components of a code element (Wen et al., 2018). In Figure 2, both the faulty node *Character.isDigit(next)* and the fixing ingredient *Character.isDigit(next) || next == '.'* access the same variable *next*. To measure variable similarity, two lists containing names and types of variables used by the faulty node (θ_{fn}) and the fixing ingredient (θ_{fi}) are generated. Next, variable similarity (vs) is calculated using (2).

$$vs(fn, fi) = \frac{|\theta_{fn} \cap \theta_{fi}|}{|\theta_{fn} \cup \theta_{fi}|} \quad (2)$$

  Two variables are considered same if their names and types are exact match. To measure variable similarity of nodes that do not contain any variable, e.g., *Boolean Literal* type nodes, only their respective data types are matched (Wen et al., 2018).

- **Normalized LCS:** LCS finds the common subsequence of maximum length by working at character-level (Chen and Monperrus, 2018). This study computes normalized LCS (nl) between faulty code (fn) and fixing ingredient (fi) at AST node level using (3).

$$nl(fn, fi) = \frac{LCS(fn, fi)}{max(|fn|, |fi|)} \quad (3)$$

  where, $max(|fn|, |fi|)$ represents the maximum length between fn and fi.

- **Token Similarity:** Unlike normalized LCS, token similarity ignores the order of text (Chen and Monperrus, 2018). It only checks whether a token (e.g., *identifiers*) exists regardless of its position. For example, both faulty node *Character.isDigit(next)* and fixing ingredient *Character.isDigit(next) || next == '.'* have *isDigit* token in common. To calculate token similarity, at first, the faulty node and fixing ingredient are tokenized. Similar to (Saha et al., 2017), camel case identifiers are further split and converted into lower-case format. For example, *isDigit* is converted into *is* and *digit*. Next, token similarity (ts) is computed using (4).

$$ts(fn, fi) = \frac{|\theta_{fn} \cap \theta_{fi}|}{|\theta_{fn} \cup \theta_{fi}|} \quad (4)$$

  where, θ_{fn} and θ_{fi} represent the token list of faulty node and fixing ingredient respectively.

Each of the above mentioned metrics outputs a score between 0 and 1. The final similarity score is calculated by adding these scores (sum of *gs*, *vs* and *nl* or *ts*), as followed in (Jiang et al., 2018). For example, if *gs*, *vs* and *ts* are 0.92, 0.66 and 0.57 respectively, the final score will be 2.15. Next, all the patches are sorted in descending order based on this final score (shown in Figure 1).

4 EXPERIMENT AND RESULT ANALYSIS

This section presents the implementation details, evaluation criteria and result analysis of the study. At first, the language and tools used for implementing the proposed approaches are discussed. Next, evaluation metrics are described. Finally, results of the proposed approaches based on the evaluation metrics are reported.

4.1 Implementation

This study proposes two approaches to examine the impact of combining syntactic and semantic similarities on patch prioritization. The devised approaches are implemented in Java since it is one of the most popular programming languages (Saha et al., 2018). It uses Eclipse JDT parser[1] for manipulating AST. It uses javalang tool[2] for tokenizing code. Javalang takes Java source code as input and provides a list of tokens as output.

To understand combined impact of similarities, the proposed approaches need to be compared with

[1] https://github.com/eclipse/eclipse.jdt.core/blob/master/org.eclipse.jdt.core/dom/org/eclipse/jdt/core/dom/ASTParser.java

[2] https://github.com/c2nes/javalang

patch prioritization techniques that use semantic and syntantic similarity individually. Therefore, this study further implements semantic or syntantic similarity based patch prioritization approaches using metrics discussed in Section 3.2 (genealogical similarity, variable similarity, normalized LCS and token similarity). The techniques are described below:

1. **Semantic Similarity based Approach (*SSBA*):** It uses only semantic similarity metrics namely genealogical and variable similarity to prioritize patches.
2. **LCS based Approach (*LBA*):** It is a syntactic similarity based approach that prioritizes patches using only normalized LCS score.
3. **Token based Approach (*TBA*):** It is another syntactic similarity based approach that uses only token similarity to prioritize patches.

All of these three 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. The implementations of these approaches are publicly available at GitHub[3].

4.2 Evaluation

In this study, following evaluation metrics are inspected:

1. **Median Rank of the Correct Patch:** The lower the median rank, the better the approach is (Le et al., 2016).
2. **Average Space Reduction:** It indicates how much search space can be avoided for finding the correct patch (Chen and Monperrus, 2018). It is calculated using (5).

$$avg\ space\ reduction = \left(1 - \frac{mean\ correct}{mean\ total}\right) * 100 \tag{5}$$

   where, *mean correct* and *mean total* denote mean of the correct patch rank and total patches generated respectively.
3. **Perfect Repair:** It denotes the percentage of bug fixes for which the correct solution is ranked at first position (Chen and Monperrus, 2018).

Similar to (Chen and Monperrus, 2018), this study considers a patch identical to human patch as correct, which is provided with the dataset.

Figure 3 shows the rank distributions of correct patches for *SSBA*, *Com-L*, *Com-T*, *LBA* and *TBA*.

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

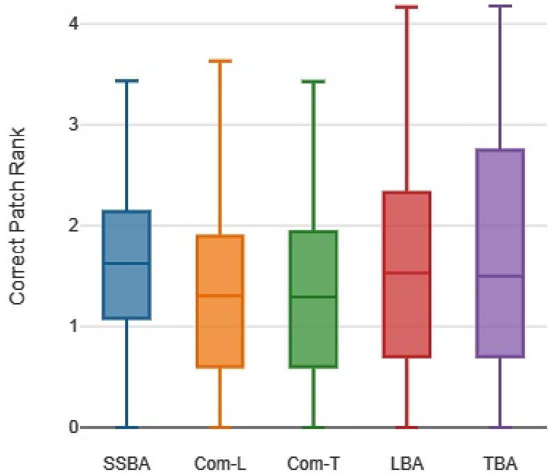

Figure 3: Comparison of Correct Patch Rank among *SSBA*, *Com-L*, *Com-T*, *LBA* and *TBA*.

Since the data range is high (1-15005), log transformation is used in this figure. It can be seen that *Com-L* and *Com-T* outperform *SSBA*, *LBA* and *TBA* in terms of median rank of the correct patch. The ranks are 20, 19.5, 42, 34 and 31.5 respectively. The reason is *Com-L* and *Com-T* incorporate information from multiple domains (both textual similarity and code meaning). A sample patch is shown in Figure 4. Here, the lines of code started with "+" and "-" indicate the added and deleted lines respectively. For this bug, *SSBA*, *LBA* and *TBA* rank the correct solution at 3, 4 and 9 respectively. On the other hand, both *Com-L* and *Com-T* rank the correct solution at 1.

```
protected void encodeBody(ConsumerFlowTokenMessage message,
        RemotingBuffer out) throws Exception {
-       out.putFloat(message.getTokens());
+       out.putInt(message.getTokens());
}
```

Figure 4: Sample Patch for Bug *hornetq_hornetq_70*.

Figure 5 demonstrates that *Com-L* and *Com-T* are effective in reducing the search space compared to *SSBA*, *LBA* and *TBA*. When random search is used, on average 50% of the search space needs to be covered before finding the correct solution (Chen and Monperrus, 2018). Using *Com-L* and *Com-T*, 96.52% and 96.62% of the total search space can be ignored to find the correct patch, whereas it is 95.38%, 84.07% and 79.49% for *SSBA*, *LBA* and *TBA* correspondingly. By using *Com-L* or *Com-T*, future automated program repair tools can consider a larger search space to fix more bugs (Wen et al., 2018).

In terms of perfect repair, *Com-L* and *Com-T* outperform *SSBA*, as shown in Figure 6. The values are

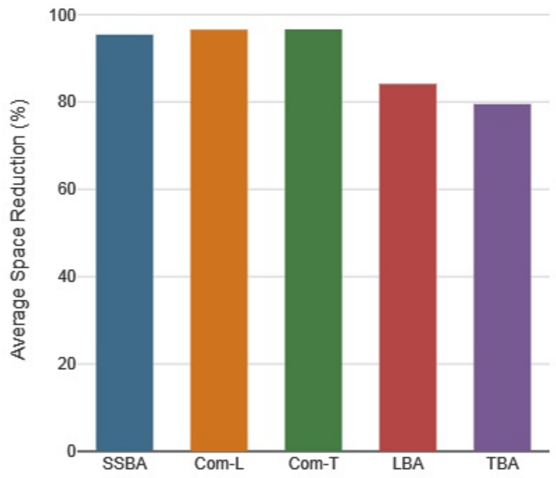

Figure 5: Comparison of Average Space Reduction among *SSBA*, *Com-L*, *Com-T*, *LBA* and *TBA*.

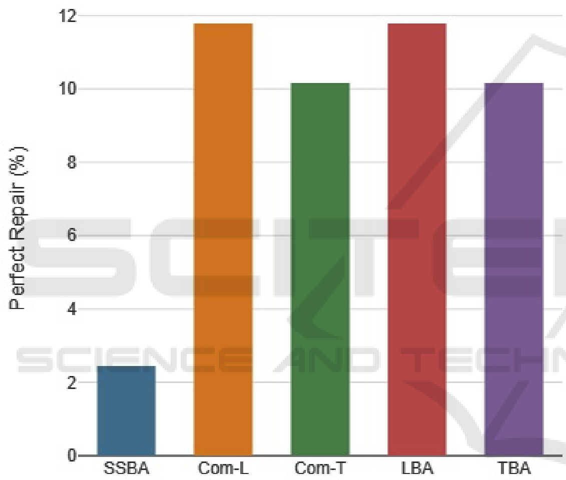

Figure 6: Comparison of Perfect Repair among *SSBA*, *Com-L*, *Com-T*, *LBA* and *TBA*.

11.79%, 10.16% and 2.44% respectively. Regarding syntactic similarity based approach, *Com-L* performs better than *TBA* and as good as *LBA*. However, *Com-T* obtains lower result than *Com-L* and *LBA*. For example, for the bug in Figure 7, *Com-T* ranks the correct solution at 8. On the other hand, both *Com-L* and *LBA* rank the correct patch at 1 due to high character level similarity. Nevertheless, the values obtained by *Com-L* and *Com-T* indicate that these approaches contribute to solving the overfitting problem. Since the first patch is the correct one in 11.79% and 10.16% cases, there is no chance of generating plausible patch before the correct one.

Table I reports the statistical significance of the obtained result using significance level = 0.05. Wilcoxon Signed-Rank test is used for this purpose

```
public Path getChunkPath(Vector3i chunkPos) {
-        return worldsPath.resolve(getChunkFilename(chunkPos));
+        return worldPath.resolve(getChunkFilename(chunkPos));
}
```

Figure 7: Sample Patch for Bug *Cervator_Terasology_2*.

since no assumption regarding the distribution of samples has been made (Walpole et al., 1993). Results show that the mean rank of *Com-L* and *Com-T* are significantly better than *SSBA*, *LBA* and *TBA*. The p-value is 0.00 in all of these cases. Although the mean rank of *SSBA* is significantly better than *TBA*, it is not significantly different from *LBA*. For some bugs such as Figure 8, the fixing ingredients are very different from the faulty code. To fix the bug, *null* is replaced with *paramType*. In this case, syntactic similarity based approaches *LBA* and *TBA* cannot rank the correct patch higher since there is no textual similarity between faulty code and fixing ingredient.

```
public int complete(String buffer, int cursor, List<String> candidates) {
-        candidate.convertFromText("*", null, include.optionContext());
+        candidate.convertFromText("*", paramType, include.optionContext());
}
```

Figure 8: Sample Patch for Bug *spring-projects_spring-roo_10*.

Results further reveal that the mean rank of *Com-T* is significantly better than *Com-L* (p-value = 0.00). The reason is when the character level difference between faulty code and fixing ingredient is high, *Com-L* can not perform well (Asad et al., 2019). However, *Com-T* has no such drawback. An example is shown in Figure 9. Here, a larger expression is replaced with a smaller one that has low character level similarity. Therefore, *Com-L* ranks the correct patch at 256, whereas *Com-T* ranks it at 87.

```
public static void scatterContigIntervals(SAMFileHeader fileHeader,
        List<GenomeLoc> locs, List<File> scatterParts) {
-        totalBases += loc.getStop() - loc.getStart();
+        totalBases += loc.size();
}
```

Figure 9: Sample Patch for Bug *broadgsa_gatk_2*.

5 THREATS TO VALIDITY

This section presents potential aspects which may threat the validity of the study:

Table 1: Difference between Mean Ranking of Correct Patch.

Compared Groups	Mean		P-value	Decision
Com-L and *SSBA*	*Com-L*	*SSBA*	0.00	Significant
	125.98	166.99		
Com-T and *SSBA*	*Com-T*	*SSBA*	0.00	Significant
	122.31	166.99		
Com-L and *LBA*	*Com-L*	*LBA*	0.00	Significant
	125.98	576.46		
Com-T and *LBA*	*Com-T*	*LBA*	0.00	Significant
	122.31	576.46		
Com-L and *TBA*	*Com-L*	*TBA*	0.00	Significant
	125.98	742.17		
Com-T and *TBA*	*Com-T*	*TBA*	0.00	Significant
	122.31	742.17		
Com-L and *Com-T*	*Com-L*	*Com-T*	0.00	Significant
	125.98	122.31		
SSBA and *LBA*	*SSBA*	*LBA*	0.13	Insignificant
	166.99	576.46		
SSBA and *TBA*	*SSBA*	*TBA*	0.00	Significant
	166.99	742.17		
LBA and *TBA*	*LBA*	*TBA*	0.29	Insignificant
	576.46	742.17		

- **Threats to External Validity:** External threat deals with the generalizability of the obtained result (Le et al., 2016). The analysis is conducted on 246 out of 3302 bugs from historical bug fix dataset (Le et al., 2016), which are selected through preprocessing (details are mentioned in Section 3.1). To mitigate the threat of generalizability, bugs belonging to popular, large and diverse projects are used. This dataset has been adopted by existing approaches (Le et al., 2017), (Wen et al., 2018) as well.
- **Threats to Internal Validity:** Threats to internal validity include errors in the implementation and experimentation (Le et al., 2016). This study assumes that fault localization outputs the correct faulty line, as followed in (Chen and Monperrus, 2018). This assumption may not be always true (Liu et al., 2019a). However, the focus of this work is patch prioritization and thereby studying fault localization is out of the scope. Similarly, analyzing the patch correctness is itself a research, which is explored by (Xin and Reiss, 2017a), (Yu et al., 2019). Following the research presented in (Chen and Monperrus, 2018), this study considers a patch identical to the patch developed by human as correct.
  To generate patches, fixing ingredients are collected from the corresponding buggy file. This process is widely followed by existing approaches (Le Goues et al., 2012), (Wen et al., 2018), (Liu et al., 2019b). For manipulating AST and tokenizing code, this study relies on Eclipse JDT parser and javalang tool respectively. These tools are widely used in automated program repair (Le et al., 2016), (Chen et al., 2017), (Chen and Monperrus, 2018), (Jiang et al., 2018).

6 CONCLUSION AND FUTURE WORK

This paper proposes two patch prioritization algorithms combining syntactic and semantic similarity metrics. Genealogical and variable similarity are used to measure semantic similarity. For capturing syntactic similarity, normalized longest common subsequence and token similarity are used individually. The approaches take source code and faulty line as input and outputs a sorted list of patches. The patches are sorted using similarity score, obtained by integrating genealogical, variable similarity with normalized longest common subsequence or token similarity.

To understand the combined impact of similarities, proposed approaches are compared with techniques that use either semantic or syntactic similarity. For comparison, 246 replacement mutation bugs out of 3302 bugs from historical bug fixes dataset are used (Le et al., 2016). The median ranks of the correct patch are 20 and 19.5 for these approaches, which outperform both semantic or syntactic similarity based techniques. Using combined methods, 96.52% and 96.62% of the total search space can be eliminated to find the correct patch. Results further show that these approaches are significantly better in ranking the correct patch earlier than semantic or syntactic based approaches. Moreover, these two techniques rank the correct solution at the top in 11.79% and 10.16% cases. It indicates that combined approaches have the potential to rank correct patch before incorrect plausible ones.

The combined methods obtain promising result in terms of median rank of the correct patch, average space reduction and perfect repair. Therefore, these approaches can be further explored using other benchmark datasets such as Defects4J (Just et al., 2014), QuixBugs (Lin et al., 2017). In addition, existing approaches such as (Le Goues et al., 2012), (Qi et al., 2014), (Le et al., 2016) can be modified to incorporate the combination of syntactic and semantic similarities for complementing their techniques.

ACKNOWLEDGEMENTS

This research is supported by the fellowship from ICT Division, a department of Ministry of Posts, Telecommunications and Information Technology, Bangladesh. No-56.00.0000.028.33.093.19-427; Dated 20.11.2019. The virtual machine facilities used in this research is provided by the Bangladesh Research and Education Network (BdREN).

REFERENCES

Asad, M., Ganguly, K. K., and Sakib, K. (2019). Impact analysis of syntactic and semantic similarities on patch prioritization in automated program repair. In *Proceedings of the 2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 328–332. IEEE.

Barr, E. T., Brun, Y., Devanbu, P., Harman, M., and Sarro, F. (2014). The plastic surgery hypothesis. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, pages 306–317. ACM.

Chen, L., Pei, Y., and Furia, C. A. (2017). Contract-based program repair without the contracts. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 637–647. IEEE.

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

Gabel, M., Jiang, L., and Su, Z. (2008). Scalable detection of semantic clones. In *Proceedings of the 30th International Conference on Software Engineering (ICSE)*, pages 321–330. ACM.

Gazzola, L., Micucci, D., and Mariani, L. (2017). Automatic software repair: A survey. *IEEE Transactions on Software Engineering*.

Jiang, J., Xiong, Y., Zhang, H., Gao, Q., and Chen, X. (2018). 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)*, pages 298–309. ACM.

Just, R., Jalali, D., and Ernst, M. D. (2014). Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis (ISSTA)*, pages 437–440. ACM.

Kim, D., Nam, J., Song, J., and Kim, S. (2013). Automatic patch generation learned from human-written patches. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE)*, pages 802–811. IEEE Press.

Le, X.-B. D., Chu, D.-H., Lo, D., Le Goues, C., and Visser, W. (2017). S3: syntax-and semantic-guided repair synthesis via programming by examples. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (FSE)*, pages 593–604. ACM.

Le, X. B. D., Lo, D., and Le Goues, C. (2016). History driven program repair. In *Proceedings of the 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 213–224. IEEE.

Le Goues, C., Dewey-Vogt, M., Forrest, S., and Weimer, W. (2012). A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *Proceedings of the 34th International Conference on Software Engineering (ICSE)*, pages 3–13. IEEE.

Le Goues, C., Nguyen, T., Forrest, S., and Weimer, W. (2011). Genprog: A generic method for automatic software repair. *IEEE Transactions on Software Engineering*, 38(1):54–72.

Lin, D., Koppel, J., Chen, A., and Solar-Lezama, A. (2017). Quixbugs: a multi-lingual program repair benchmark set based on the quixey challenge. In *Proceedings Companion of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity (SPLASH)*, pages 55–56. ACM.

Liu, K., Koyuncu, A., Bissyandé, T. F., Kim, D., Klein, J., and Le Traon, Y. (2019a). You cannot fix what you cannot find! an investigation of fault localization bias in benchmarking automated program repair systems. In *Proceedings of the 2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, pages 102–113. IEEE.

Liu, K., Koyuncu, A., Kim, D., and Bissyandé, T. F. (2019b). Tbar: revisiting template-based automated program repair. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 31–42. ACM.

Martinez, M., Weimer, W., and Monperrus, M. (2014). Do the fix ingredients already exist? an empirical inquiry into the redundancy assumptions of program repair approaches. In *Companion Proceedings of the 36th International Conference on Software Engineering (ICSE)*, pages 492–495. ACM.

Monperrus, M. (2018). Automatic software repair: a bibliography. *ACM Computing Surveys (CSUR)*, 51(1):17.

Nguyen, T. T., Nguyen, A. T., Nguyen, H. A., and Nguyen, T. N. (2013). A statistical semantic language model for source code. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (FSE)*, pages 532–542. ACM.

Qi, Y., Mao, X., Lei, Y., Dai, Z., and Wang, C. (2014). The strength of random search on automated program repair. In *Proceedings of the 36th International Conference on Software Engineering (ICSE)*, pages 254–265. ACM.

Ragkhitwetsagul, C., Krinke, J., and Clark, D. (2018). A comparison of code similarity analysers. *Empirical Software Engineering*, 23(4):2464–2519.

Saha, R., Lyu, Y., Lam, W., Yoshida, H., and Prasad, M. (2018). Bugs. jar: A large-scale, diverse dataset of real-world java bugs. In *Proceedings of the 2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*, pages 10–13. IEEE.

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

Tan, S. H. and Roychoudhury, A. (2015). relifix: Automated repair of software regressions. In *Proceedings of the 37th International Conference on Software Engineering (ICSE)-Volume 1*, pages 471–482. IEEE Press.

Thada, V. and Jaglan, V. (2013). Comparison of jaccard, dice, cosine similarity coefficient to find best fitness value for web retrieved documents using genetic algorithm. *International Journal of Innovations in Engineering and Technology*, 2(4):202–205.

Walpole, R. E., Myers, R. H., Myers, S. L., and Ye, K. (1993). *Probability and statistics for engineers and scientists*, volume 5. Macmillan New York.

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

White, M., Tufano, M., Martinez, M., Monperrus, M., and Poshyvanyk, D. (2019). Sorting and transforming program repair ingredients via deep learning code similarities. In *Proceedings of the 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 479–490. IEEE.

Xin, Q. and Reiss, S. P. (2017a). Identifying test-suite-overfitted patches through test case generation. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 226–236. ACM.

Xin, Q. and Reiss, S. P. (2017b). Leveraging syntax-related code for automated program repair. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 660–670. IEEE.

Xiong, Y., Liu, X., Zeng, M., Zhang, L., and Huang, G. (2018). Identifying patch correctness in test-based program repair. In *Proceedings of the 40th International Conference on Software Engineering (ICSE)*, pages 789–799. ACM.

Xiong, Y., Wang, J., Yan, R., Zhang, J., Han, S., Huang, G., and Zhang, L. (2017). Precise condition synthesis for program repair. In *Proceedings of the 39th International Conference on Software Engineering (ICSE)*, pages 416–426. IEEE.

Yu, Z., Martinez, M., Danglot, B., Durieux, T., and Monperrus, M. (2019). Alleviating patch overfitting with automatic test generation: a study of feasibility and effectiveness for the nopol repair system. *Empirical Software Engineering*, 24(1):33–67.