AN EMPIRICAL STUDY OF THE PERFORMANCE OF CODE SIMILARITY IN AUTOMATIC PROGRAM REPAIR TOOL

Xingyu Zheng, Zhiqiu Huang, Yongchao Wang and Yaoshen Yu

College of Computer Science and Technology,
Nanjing University of Aeronautics and Astronautics, Nanjing, China

ABSTRACT

Recently, code similarity has been used in several automated program repair (APR) tools. It suggests that similarity has a great contribution to APR. However, code similarity is not fully utilized in some APR tools. For example, SimFix only uses structure similarity (Deckard) and name (variable and method) similarity to rank candidate code blocks that are used to extract patches and do not use similarity in patch filtering. In this paper, we combine the tool with longest common sequence (LCS) and term frequency-inverse document frequency (TFIDF) to rank candidate code blocks and filter incorrect patches. Then we design and set up a series of experiments based on the approach and collect the rank of the correct patch and time cost for each selected buggy program. In the candidate ranking phase, LCS and TFIDF improve the rank of the block, which contains the correct patch for several bugs. In the patch validation phase, LCS filters out 68% of incorrect patches on average. It shows that code similarity can greatly improve the performance of APR tools.

KEYWORDS

APR, empirical study, LCS, TFIDF.

1. INTRODUCTION

In recent years, automated program repair has become a hot research direction. APR consists of automatically finding a solution to software bugs without human intervention [1] due to the fact that debugging software failures is still a painful, time consuming, and expensive process [2]. Generally, the process of APR can be divided into three stages: fault localization, patch generation and patch validation [3]. When given a buggy program, the buggy code block is first determined in fault localization stage, and then the APR tool tries to create patches in the patch generation stage, the program is finally executed to see whether the patch is correct in patch validation stage.

Automated program repair can be simply divided into four categories: manual template (PAR [4], SketchFix [5]), semantic constraint (Angelix [6], SOSRepair [7]), statistical analysis (GenPat [8], SequenceR [9]), heuristic search (SimFix [10], CapGen [11]). Redundancy-based program repair is fundamental to APR, and it is based on the hypothesis that code may evolve from existing code that comes from somewhere else, for instance, from the program under repair [12]. When given the fault localization result, APR tools based on redundancy search code blocks that are similar to the buggy code block in the current program, the patches generated from these code blocks are then validated to find the first correct patch that passes all test cases. Recently, there have been several redundancy-based APR techniques (such as SimFix [10], CapGen [11], ssFix [13],

David C. Wyld et al. (Eds): ARIA, SIPR, SOFEA, CSEN, DSML, NLP, EDTECH, NCWC - 2022
pp. 25-36, 2022. CS & IT - CSCP 2022 DOI: 10.5121/csit.2022.121503

CRSearcher [14]). These techniques leverage different similar metrics like LCS, TFIDF, name similarity, Deckard and so on to compute the similarity between the buggy code block and the other blocks. The experimental results indicate that they all perform well in repair effectiveness. However, the techniques do not take full advantage of code similarity. On the one hand, for candidate code block ranking in patch generation stage, similar metrics used in different techniques can be combined to see if the priority operation is improved in this way. On the other hand, in patch validation stage, each of the techniques needs to validate a large number of patches generated in the last stage. We can utilize similar metrics to dynamically reduce the search space of patches before validation.

Our study of how code similarity works on the performance of APR tool takes SimFix, a state-of-the-art approach to design the experiment and analyze the results. First, we only select the buggy programs that have been successfully repaired by the tool for the following experiments. We exclude the buggy programs that the tool incorrectly repaired (the generated patch passes all test cases but is not correct) and failed to repair (fail to generate a patch that can pass all test cases or time out) for convenience. Second, we utilize two similarity metrics that capture syntactic messages —LCS and TFIDF in both patch generation stage and patch validation stage of the tool, then we design and perform a series of experiments. The main goal of the experiments is to study how good the new similarity metrics are at improving the rank of the first correct patch in the search space. For evaluating the experimental performance, we collect the rank of the first correct patch and the time cost of each buggy program before and after applying new metrics and compare the experimental data. Our experimental results are clear-cut. Firstly, LCS and TFIDF effectively rank candidate similar code blocks, improving the mean reciprocal rank (MRR) by 26%. Secondly, utilizing LCS in patch filtering successfully excludes 68% incorrect patches on average.

To sum up, the main contributions of our work are as follows:

- An empirical study of how code similarity performs in a concrete APR tool.
- When we add new code similarity metrics to improve the rank of the first correct patch for a buggy program in APR tool, the results perform well. LCS and TFIDF rank the true candidate code block higher in patch generation stage and patch validation stage, LCS filter out a large number of incorrect patches, let the correct patch be validated earlier.
- We collect the rank distributions and time cost of the buggy programs after applying new similar metrics. The results show what contributes to the huge distinction of the result of ranking and time cost and why new metrics do not perform better in some cases.

The remainder of this paper is structured as follows: The related work is given in Section II. Section III describes the research methodology. The numerical results and analysis are presented in Section IV. Finally, conclusions are drawn in Section V.

2. RELATED WORK

In this section, we discuss the redundancy-based APR tools and several empirical studies.

2.1. Redundancy-based APR

In this subsection, we will present an overview of redundancy-based automated program repair. As mentioned before, the repair process can be divided into three stages: fault localization, patch generation and patch validation. In fault localization stage, given a buggy program, APR tool identifies an ordered list of suspicious buggy blocks using approaches like Ochiai [15]. In patch

generation stage, the tool first searches for donor code blocks that are similar to the buggy code for each buggy location, then compares the buggy code and donor code to extract patches for each donor code block and get the search space of patches. In patch validation stage, the patches are tried one by one from the search space to find the first correct patch that can pass all test cases.

In particular, SimFix [10] leverages Deckard [16] and name similarity to calculate the similarity between the buggy code block and candidate code blocks in patch generation stage and exclude patches that use less frequent modifications based on offline mining of the frequency of different modifications. CapGen [11] prioritizes candidate code blocks that are contextually similar to buggy code block and rank the generated patches by integrating the fault space (suspicious value of buggy code), the modification space (replacement, insertion or deletion) and the ingredient space (context similarity) together. ssFix [13] extracts tokens from buggy code block and leverages TFIDF to search for candidate code blocks and rank them. CRSearcher [14] treats similar code search as a clone detection problem and utilizes a variant of Running Karp Rabin Greedy-String-Tiling Algorithm (RKR-GST), a token based approach to search for candidate code blocks.

2.2. Empirical Studies on APR

In this subsection, we will present recent empirical studies on automatic program repair. Moumita Asad et al. [17] analyze the impact of syntactic and semantic similarity on patch prioritization. They rank patches by integrating genealogical, variable similarity (semantic similarity) and LCS(syntactic similarity), and the results are better than using one similarity metric alone. Xiong et al. [18] perform an empirical study that identify patch correctness in test-based APR. For each test case, they leverage LCS to calculate the similarity of complete-path spectrum (the sequence of executed statement IDs) between the two executions before and after applying a patch and judge whether a patch is correct. The approach successfully filters out 56.3% of incorrect patches. Liu et al. [19] systematically investigates the repair effectiveness and efficiency of 16 APR tools in recent years. They find that fault localization plays an extremely important role in a repair process, and accurate localization can greatly reduce the number of generated patches. Chen et al. [12] define and set up a large-scale experiment based on four code similarity metrics that capture different similarities—LCS, TFIDF, Decksrd and Doc2vec. The paper considers two cases—context-less and context-aware (i.e., one-line buggy code and multi-line buggy code) and performs the experiments. According to their rank statistic, all of the 4 similar metrics reduce over 90% of the search space in both cases; at the same time, LCS and TFIDF can rank the correct patch higher than Deckard and Doc2vec in context-less cases, and the performance of the 4 metrics is close in context-aware case. Furthermore, the paper studies the feasibility of combining different metrics and points out that LCS and TFIDF can be used together in context-aware cases. Consider the great performance of LCS and TFIDF and the fact that most bugs selected in our experiment have multiple lines. We decide to utilize them in SimFix to investigate how code similarity can improve the performance of APR tool.

3. RESEARCH METHODOLOGY

In this section, we will describe our research methodology.

The overall workflow of our work is illustrated in Fig. 1. For each buggy program, the fault localization stage returns a list of buggy code block list based on their suspicious score, then the APR tool searches similar code blocks in the program for the buggy blocks, the similarity between the candidate code block and the buggy block is requested to be more than a threshold

(dynamically changed based on the number of lines of the buggy block). The searched similar code blocks will be ranked based on the similarity metrics of the APR tool. Here we apply LCS and TFIDF, and this step returns the list of ranked candidate code blocks. Patches generated from the candidates compose the search space. Different from the original process that directly validated the patches one by one, when a patch is to be validated, we first calculate whether it is highly similar to the patches that have been validated to be incorrect. If so, we exclude this patch and verify the next one; otherwise, we validate this patch and add it to the set of incorrect patches if it can not repair the bug. Finally, a correct patch is returned.

3.1. Overview of the Research Methodology

Our research methodology is as follows. In the patch generation stage, when given the list of candidate code blocks, The original tool will rank them based on structure similarity and name similarity. We first directly add LCS and TFIDF, respectively, to the original metric and collect the experiment data. Next, we combine LCS and TFIDF, execute the tool with and without name similarly and collect the data. We investigate the statistical data of the rank of the first correct patch and the time cost for each bug based on different settings and compare which condition the tool performs best.

In the patch validation phase, the tool ranks the generated patches based on three rules and excludes delete operation, and the tool includes 16 most frequent modifications (like inserting an If statement in the buggy point) to reduce the search space of patches. However, there still exists a great number of incorrect patches. Moumita Asad et al. [17] confirms that similar metrics based on syntactic message like LCS can greatly prioritize patches. We leverage LCS to filter out those incorrect patches. Since unvalidated incorrect patches are similar to validated incorrect patches and are not similar to the correct unvalidated patch, we calculate the similarity scores between the next patch to be validated and the patches which have been validated and filter out those patches that are very similar to the incorrect patch (for example, the similarity score > 0.9). We investigate the statistical data of the rank of the first correct patch and time cost and then analyze the performance of the filtering metric.

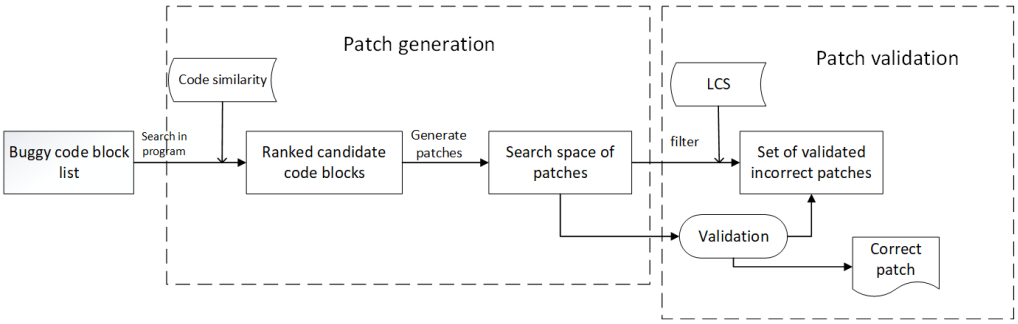

Figure 1. Illustration of our work in APR tool.

3.2. Similarity Metrics

Our core idea is to analyze how code similarity contributes to APR. The metrics used by the original tool and the metrics we use are:

1) Structure similarity between AST (Deckard)
2) Variable similarity and method similarity (Name similarity)
3) Longest common sequence (LCS)

4) Term frequency–inverse document frequency (TFIDF)

The four metrics are explained in detail in the following:

1) Deckard: Deckard is a metric that calculates similarity at AST level. It first generates feature vectors from the buggy code and candidate code blocks. Each dimension of the feature vector captures different information of the code block, such as the number of operations. Deckard then calculates cosine similarity of the feature vectors of buggy block and candidate block. SimFix uses Deckard for measuring AST similarity and selects the top 1000 at most candidates to further measure name similarity.
2) Name similarity: Name similarity consists of variable similarity and method similarity. Variable similarity calculates how similar the variables in two code blocks are. The tool first obtains two sets of variables from two code blocks and then calculates the similarity of the two sets using Dice's coefficient. Like variable similarity, method similarity calculates how similar the names of methods in two code blocks are, and method similarity is calculated in the same way as the variable similarity.
3) LCS: LCS treats the source code as a sequence of characters to calculate the longest common sequence of the code blocks. So LCS is the most syntactic metric. In the current scene, similar blocks or patches are highly consistent in the sequence of words and are, of course, highly consistent in the sequence of characters. So in our methodology, LCS is used to calculate the similarity between candidate code blocks and the buggy code block and the similarity between the next verified patch and the validated patches.
4) TFIDF: TFIDF, i.e., term frequency-inverse document frequency, is used to evaluate the importance of a word to a document set or one of the documents in a corpus. This means the importance of a word increases in proportion to the number of times it appears in the document but decreases in inverse proportion to the frequency of its appearance in the corpus. The similarity calculation of TFIDF is based on a token level, so TFIDF is less syntactic than LCS. TFIDF can more effectively utilize unique tokens. If we have a candidate code block that contains the same variable that only occurred once in the buggy block, it is likely that the code block is a good candidate. In our methodology, TFIDF is used to calculate the similarity between candidate code blocks and the buggy code block. We do not use TFIDF to filter incorrect patches because the set of validated patches is dynamically changing. Every time the next patch is to be validated, IDF needs to be calculated again for all the patches in the set.

4. EVALUATION

To evaluate how code similarity can improve the performance of APR tool, we design a series of experiments. Our experiment was conducted on a 64-bit Linux system with Intel(R) Core(TM) i5-6300HQ CPU and 12GB RAM, which is close to the environment of the original experiment of the tool that we study.

4.1. Research Questions

RQ1: How do LCS and TFIDF perform in candidate ranking? When given the buggy code block after fault localization, the original tool searches the current subject and gets a set of candidate code blocks based on a similarity threshold. Then the tool uses Deckard and name similarity to rank the candidates. The rank of candidate code blocks directly affects the rank of generated patches in the following validation step. The higher the true candidate ranking, the earlier the correct patch will be validated. Therefore the main purpose of RQ1 is to see how LCS and TFIDF can rank the true candidate code block which contains the correct patch over the other.

RQ2: How does LCS perform in incorrect patch filtering? After the candidate code blocks are ranked, the original tool matches the candidate code block with the buggy code block at AST level and extracts patches for validation. Due to a large amount of ranked candidate code blocks, the corresponding patch search space is enormous. For those bugs whose true candidate ranks low, a great number of incorrect patches will be validated, which has a bad effect on the tool's performance. We use LCS to filter out patches that are similar to validated incorrect ones and see to what extent the approach will improve the rank of correct patch.

RQ3: What affect the performance of LCS and TFIDF? We repeat the original experiment of the tool in our machine and collect the result of repair time and rank of the first correct patch. Then we integrate LCS and TFIDF with the tool and get the experimental data. The result of the original experiment shows that in different bugs from different subjects, the rank of the first correct patch and the time cost is greatly distinct. the rank of the first correct patch varies from 1 to 1500, and the time cost ranges from 1 to 54 minutes. The purpose of RQ3 is to analyze the factors that influence the performance of code similarity metrics.

4.2. Data set

To evaluate the effectiveness of LCS and TFIDF, we select the 26 bugs from 4 subjects in Defects4j [20] benchmark that has been fixed by the tool and perform our experiment. We exclude 4 bugs from JFreechart (Chart) because they can not be successfully fixed on our machine. Table 1 shows statistics about the projects.

Table 1. Subjects For Evaluation

Subjects	**Bugs**
Closure compiler (Closure)	4
Apache commons-math (Math)	13
Apache commons-lang (Lang)	8
Joda-Time (Time)	1
Total	26

4.3. Experimental Results

We now present the result of our experiments on the performance of code similarity in APR tool.

4.3.1. Research Question 1: How do LCS and TFIDF Perform in Candidate Ranking?

We investigate the research question on the 26 bugs, and the rank of the first correct patch and time cost (in minutes) for the bugs are shown in TABLE II. In the table, column "Origin" denotes the ranking result of the original tool, column "L+T" denotes the ranking result of the combination of LCS, TFIDF and the tool, column "L+T−NS" denotes the ranking result of LCS, TFIDF and Deckard. It is nice that integrating LCS and TFIDF with the original tool in the candidate ranking phase does not change the time cost a lot whether the rank of the first correct patch is changed. So we do not need to analyze the time cost performance (in minutes) in this research question.

From TABLE II, we can see that both LCS and TFIDF successfully improve the rank of the first correct patch in many cases, and for those bugs whose correct patch has been ranked first in the tool, LCS and TFIDF do not make the result worse. We calculate the mean reciprocal rank for each setting and see that the performance of LCS and TFIDF is very close. They both improve the MRR by 10%. When combining the two metrics and integrating with the tool, the

performance is almost unchanged from the results of using the two tools alone. This may be because LCS and TFIDF both calculate similarity scores based on the syntactic message of code blocks. As shown in the L+T−NS column, we replace the name similarity metric with the combination of LCS and TFIDF and execute the program. The results show that in this setting, MRR has an improvement of 26%, which is higher than integrating all the metrics. This means LCS and TFIDF rank candidate code blocks better than name similarity. It is because name similarity only collects the variable name and method name in a code block, but TFIDF collects all tokens, i.e., all words in the block. LCS similarly calculates the scores according to all the characters in the block.

Table 2. Rank and Time Statistic in Different Settings

Bugs	**Origin**	**LCS**	**TFIDF**	**L+T**	**L+T-NS**	**Time**
Math5	1	1	1	1	1	2
Math50	8	10	9	9	4	4
Math53	2	2	2	2	2	1
Math63	16	16	14	16	9	5
Math70	1	1	1	1	1	1
Math71	14	9	9	9	9	5
Math75	8	4	8	4	4	1
Lang27	7	6	4	4	1	2
Lang41	2	1	1	1	1	10
Lang58	1	1	1	1	1	1
Closure73	1	1	1	1	1	7
Math33	55	62	62	62	61	4
Math35	5	5	5	5	5	6
Math57	112	119	113	110	111	5
Math59	9	9	9	9	9	10
Math79	562	559	559	559	559	18
Math98	212	211	212	211	212	10
Lang16	1491	1497	1494	1489	1436	51
Lang33	42	36	39	35	32	1
Lang39	1296	1316	1314	1310	1302	54
Lang43	6	6	6	6	6	7
Lang60	130	117	116	118	115	10
Time7	431	428	438	430	440	23
Closure14	344	344	344	344	347	53
Closure57	21	25	21	24	22	17
Closure115	3	3	3	3	3	6
MRR	0.248	0.274	0.273	0.277	0.313	

4.3.2. Research Question 2: How does LCS Perform in Incor Rect Patch Filtering?

We investigate the research question on the 26 bugs, and the rank of the first correct patch and time cost (in minutes) are shown in TABLE III. In the table, column ”Rank(o)” denotes the ranking result of the original tool, column ”Rank(f)” denotes the ranking result with incorrect patch filtering, column ”Time(o)” denotes the time cost of the tool, column ” Rank(f)” denotes the ranking result with incorrect patch filtering. As analyzed in Section III and RQ1, The ranking performance of LCS and TFIDF are highly similar, and calculating TFIDF in a dynamically changing set of patches is time-consuming. So we only focus on LCS. In this research question, we can see that LCS makes significant progress in filtering incorrect patches and reducing time cost.

From TABLE III, we can see that the rank of the first correct patch for some bugs is extremely low, especially for lang16 and lang39. The rank of the correct patch is over 1000. This means the vast majority of patches in the search space are incorrect. We utilize LCS to calculate the similarity scores between the next patch to be validated and the set of validated incorrect patches and exclude the related patch if one of the similarity scores exceeds a threshold. Here we set the threshold as 0.9 in common.

Table 3. Rank and Time Statistic After Filtering.

Bugs	**Rank(o)**	**Rank(f)**	**Time(o)**	**Time(f)**
Math5	1	1	2	2
Math50	8	6	4	2
Math53	2	2	1	1
Math63	16	13	5	2
Math70	1	1	1	1
Math71	14	12	5	5
Math75	8	5	1	1
Lang27	7	6	2	2
Lang41	2	1	10	4
Lang58	1	1	1	1
Closure73	1	1	7	7
Math33	55	26	4	4
Math35	5	5	6	6
Math57	112	39	5	3
Math59	9	9	10	10
Math79	562	110	18	5
Math98	212	58	10	4
Lang16	1491	354	51	15
Lang33	42	15	1	2
Lang39	1296	173	54	9
Lang43	6	6	7	7
Lang60	130	59	10	3
Time7	431	244	23	15
Closure14	344	42	53	12
Closure57	21	21	17	17
Closure115	3	2	5	5
AVERAGE	189	47	12	5.4

We can see that the ranks of the first correct patch of the bugs in TABLE III are extremely improved, and the time cost is greatly reduced. 75% improves the average rank of the first correct patch and the average time cost is improved by 55%, and the average reduction of search space is 68% for the bugs whose ranking results have changed. This means in the patch validation stage, filtering unvalidated incorrect patches greatly improves the rank of the first correct patch. For math33 and lang33, the time cost does not reduce and even increases a little. It is because the number of exclusions is in the dozens, which is inadequate to offset the time consumption of calculating similarity scores. For other bugs, the time cost reduces greatly because the time consumed in executing the tests is significantly cut down, achieving an average reduction of 56%.

4.3.3. Research Question 3: What Affect the Performance of LCS and TFIDF?

Due to the distinct experimental result of the 26 bugs from 4 subjects, we further analyze what contributes to the difference in this research question. To do this, we print out the buggy code

block list (the rank of the true buggy block of each bug is shown in TABLE IV), the ordered list of candidate code blocks and the list of generated patches in relative repair stages and investigate the collected message together with the log message of each bug printed by the original tool.

Table 4. Rank of the Buggy Code.

Bugs	Rank	Bugs	Rank
Math33	6	Math5	1
Math35	3	Math50	1
Math57	12	Math53	1
Math59	2	Math63	1
Math79	18	Math70	1
Math98	3	Math71	1
Lang16	20	Math75	1
Lang33	3	Lang27	1
Lang39	19	Lang41	1
Lang43	4	Lang58	1
Lang60	7	Closure73	1
Time7	11		
Closure14	2		
Closure57	2		
Closure115	5		

As shown in TABLE IV, the 26 bugs can be divided into two groups according to the rank of the true buggy block. On the one hand, for those bugs that the true buggy block is ranked first, the rank of the first correct patch improved when combining LCS and TFIDF with the original tool, but filtering incorrect patches did not contribute to a higher ranking. This is because these bugs' search space is too small and does not contain many similar incorrect patches. On the other hand, for those bugs where the true buggy block is not ranked first, the search space was extremely reduced by leveraging LCS to filter out incorrect patches, but leveraging LCS and TFIDF in ranking candidate blocks did not work on these bugs. This is because ranking candidate blocks for the current true buggy block can not exclude the patches generated before for the false buggy blocks. All in all, the result of fault localization and the size of search space greatly influence the performance of LCS and TFIDF.

4.3.4. Case Analysis

We now present case analyses of great rankings. We investigate the buggy code block list, the ordered list of candidate code blocks, the list of generated patches and the log message of the bugs to explain the excellent performance in these cases.

```
buggy code block:
if(expPos > -1){
   mant = str.substring(0, expPos);
}else {
   mant = str;
}

false candidate:
if(expPos > -1){
   if(expPos < decPos){
     throw new NumberFormatException(str + " is not a valid number.");
   }
   dec = str.substring(decPos+1, expPos);
} else{
   dec = str.substring(decPos+1);
}
mant = str.substring(0, decPos);

true candidate:
if(expPos > -1 && expPos < str.length()-1){
   exp = str.substring(expPos+1, str.length());
} else{
   exp = null;
}
```

Listing 1. Lang27 from Apache commons-lang

```
buggy code block;
int sum = 0;
for(int i = 0; i < pointSet.size(); i++){
   final T p = pointSet.get(i);
   final Cluster<T> nearest = getNearestCluster(resultSet, p);
   final double d = p.distanceFrom(nearest.getCenter());
   sum += d * d;
   dx2[i] = sum;
}

incorrect patch;
int sum = 0;
- for(int i = 0; i < pointSet.size(); i++){
+ for(int i = sum; i < pointSet.size(); i++){
   final T p = pointSet.get(i);
   final Cluster<T> nearest = getNearestCluster(resultSet, p);
   final double d = p.distanceFrom(nearest.getCenter());
   sum += d * d;
   dx2[i] = sum;
}

an incorrect patch similar to the incorrect patch;
int sum = 0;
- for(int i = 0; i < pointSet.size(); i++){
+ for(int i = 0; i <= pointSet.size(); i++){
   final T p = pointSet.get(i);
   final Cluster<T> nearest = getNearestCluster(resultSet, p);
   final double d = p.distanceFrom(nearest.getCenter());
   sum += d * d;
   dx2[i] = sum;
}
```

Listing 2. Math57 from Apache commons-math

1) Case analyse 1: For the bug Lang27, the correct patch is a replacement of an if statement. Combining LCS and TFIDF and excluding name similarity (setting 5, i.e., column 6 in table 2) improve the rank of the correct patch from 7 to 1 for lang27. If we do not exclude name similarity (setting 4), the rank is 4. According to the buggy block list, the true buggy code is ranked first.

This means all patches in search space are generated for this block. The original approach ranks the candidate block that contains the correct patch at 6th, but the best setting ranks it at 2nd. The tool does not extract any patch from the most similar block because it is equal to the buggy code. So the best setting successfully ranks the correct patch generated from the true candidate block first. As shown in Listing 1, the false candidate is ranked first in setting 4, and the true candidate is ranked first in setting 5. This is because the false candidate block shares more variable and method names with the buggy code block than the true candidate. This leads to a low rank of the correct block. However, the correct patch is most similar to the buggy code when treating code block as a sequence of characters or tokens, which LCS and TFIDF do. So in this case, including all similarity metrics contributes to a worse result.

2) Case analyse 2: For the bug math57, the correct patch is to replace ”int” with ”double”. This modification is too small. Utilize LCS successfully improve the rank of the correct patch from 112 to 39, and the time cost is reduced from 5 to 3 minutes. We present the buggy code block and two incorrect patches in Listing 2. The two incorrect patches are extremely similar in character level. The approach generates a large amount number of patches like this. Filtering out these patches can greatly reduce the search space and accelerate the repairing process. And as mentioned before, we initially set the filtering threshold to 0.9. However, the APR tool failed to repair this bug because the correct patch was identified as incorrect and filtered out before validating. This is because the patches in search space are highly similar to each other due to the small size of the repair. We finally fixed the buggy when the threshold was adjusted to 0.97. The good news is that although the threshold is really high, LCS still filters out 65% incorrect patches.

5. CONCLUSION

In this paper, we design and set up a series of experiments to investigate how code similarity can improve the performance of APR tool. We study a state-of-the-art approach called SimFix to see where we can utilize code similarity and a recent empirical study to select suitable similar metrics. To rank the first correct patch higher in the search space and reduce time cost, we apply two similarity metrics—LCS and TFIDF in candidate code block ranking and incorrect patch filtering. We analyze the experimental result and get the following conclusion: First, in candidate code block ranking phase, combining LCS and TFIDF and excluding name similarity used in SimFix perform best and improve the MRR by 26%. Second, in patch filtering stage, applying LCS to calculate the similarity between the next verified patch and the set of validated patches can greatly prevent incorrect patches from being verified. The method reduces search space of patches by an average of 68% and time cost by 56%.

REFERENCES

[1] M. Monperrus, “Automatic software repair: a bibliography,” ACM Computing Surveys (CSUR), vol. 51, no. 1, pp. 1–24, 2018.

[2] L. Gazzola, D. Micucci, and L. Mariani, “Automatic software repair: A survey,” IEEE Transactions on Software Engineering, vol. 45, no. 1, pp. 34–67, 2017.

[3] K. Liu, L. Li, A. Koyuncu, D. Kim, Z. Liu, J. Klein, and T. F. Bissyande, “A critical review on the evaluation of automated program ´ repair systems,” Journal of Systems and Software, vol. 171, p. 110817, 2021.

[4] D. Kim, J. Nam, J. Song, and S. Kim, “Automatic patch generation learned from human-written patches,” in 2013 35th International Conference on Software Engineering (ICSE). IEEE, 2013, pp. 802–811

[5] J. Hua, M. Zhang, K. Wang, and S. Khurshid, “Towards practical program repair with on-demand candidate generation,” in Proceedings of the 40th international conference on software engineering, 2018, pp. 12–23.

[6] S. Mechtaev, J. Yi, and A. Roychoudhury, “Angelix: Scalable multiline program patch synthesis via symbolic analysis,” in Proceedings of the 38th international conference on software engineering, 2016, pp. 691– 701.
[7] A. Afzal, M. Motwani, K. T. Stolee, Y. Brun, and C. Le Goues, “Sosrepair: Expressive semantic search for real-world program repair,” IEEE Transactions on Software Engineering, vol. 47, no. 10, pp. 2162– 2181, 2019.
[8] J. Jiang, L. Ren, Y. Xiong, and L. Zhang, “Inferring program transformations from singular examples via big code,” in 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, 2019, pp. 255–266.
[9] Z. Chen, S. Kommrusch, M. Tufano, L.-N. Pouchet, D. Poshyvanyk, and M. Monperrus, “Sequencer: Sequence-to-sequence learning for end-to-end program repair,” IEEE Transactions on Software Engineering, vol. 47, no. 9, pp. 1943–1959, 2019.
[10] 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, 2018, pp. 298–309.
[11] M. Wen, J. Chen, R. Wu, D. Hao, and S.-C. Cheung, “Contextaware patch generation for better automated program repair,” in 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE). IEEE, 2018, pp. 1–11.
[12] Z. Chen and M. Monperrus, “The remarkable role of similarity in redundancy-based program repair,” arXiv preprint arXiv:1811.05703, 2018.
[13] Q. Xin and S. P. Reiss, “Leveraging syntax-related code for automated program repair,” in 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, 2017, pp. 660–670.
[14] Y. Wang, Y. Chen, B. Shen, and H. Zhong, “Crsearcher: Searching code database for repairing bugs,” in Proceedings of the 9th Asia-Pacific Symposium on Internetware, 2017, pp. 1–6.
[15] X. Xie and B. Xu, Essential Spectrum-based Fault Localization. Springer, 2021.
[16] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, “Deckard: Scalable and accurate tree-based detection of code clones,” in 29th International Conference on Software Engineering (ICSE’07). IEEE, 2007, pp. 96– 105.
[17] M. Asad, K. K. Ganguly, and K. Sakib, “Impact analysis of syntactic and semantic similarities on patch prioritization in automated program repair,” in 2019 IEEE International Conference on Software Maintenance and Evolution (ICSME). IEEE, 2019, pp. 328–332.
[18] Y. Xiong, X. Liu, M. Zeng, L. Zhang, and G. Huang, “Identifying patch correctness in test-based program repair,” in Proceedings of the 40th international conference on software engineering, 2018, pp. 789–799.
[19] K. Liu, S. Wang, A. Koyuncu, K. Kim, T. F. Bissyande, D. Kim, P. Wu, ´ J. Klein, X. Mao, and Y. L. Traon, “On the efficiency of test suite based program repair: A systematic assessment of 16 automated repair systems for java programs,” in Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, 2020, pp. 615–627.
[20] R. Just, D. Jalali, and M. D. Ernst, “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, 2014, pp. 437–440.