applied sciences

MDPI

Article

Code Similarity and Location-Awareness Automatic Program Repair

Heling Cao [1,2,3], Dong Han [1,2,3], Fangzheng Liu [1,2,3], Tianli Liao [1,2,3,*], Chenyang Zhao [1] and Jianshu Shi [1]

[1] Key Laboratory of Grain Information Processing and Control, Henan University of Technology, Ministry of Education, Zhengzhou 450001, China; caohl@haut.edu.cn (H.C.); zhaochenyang@haut.edu.cn (C.Z.)
[2] Henan Key Laboratory of Grain Photoelectric Detection and Control, Henan University of Technology, Zhengzhou 450001, China
[3] College of Information Science and Engineering, Henan University of Technology, Zhengzhou 450001, China
* Correspondence: tianli.liao@haut.edu.cn

Abstract: Automatic program repair has drawn more and more attention since software quality is facing increasing challenges. In existing approaches, the unlimited search space is considered to be the main limitation in finding the correct patch. So how to reduce the search space to improve the efficiency of automatic program repair remains a problem to be solved. In this work, we represent a similarity-based and location-awareness-based automatic program repair (SLARepair). SLARepair takes the similarity between codes as important search information. The search space is further subdivided by the location-awareness strategy to improve search efficiency. In addition, to better guide the search process, a new fitness function is designed for genetic programming, which brings notable improvements. Moreover, the patch verification time is further reduced by utilizing the test case prioritization approach combined with test case filtering. Extensive experiments demonstrate that our SLARepair outperforms the state-of-the-art approaches on the Defects4J benchmark and achieves competitive performances.

Keywords: automatic program repair; code similarity; location awareness

Citation: Cao, H.; Han, D.; Liu, F.; Liao, T.; Zhao, C.; Shi, J. Code Similarity and Location-Awareness Automatic Program Repair. *Appl. Sci.* **2023**, *13*, 8519. https://doi.org/10.3390/app13148519

Academic Editor: Stefan Fischer

Received: 13 June 2023
Revised: 15 July 2023
Accepted: 21 July 2023
Published: 23 July 2023

1. Introduction

Automatic program repair is an important tool in software maintenance to improve software quality. Traditional debugging is typically done by developers, but it is a time-consuming and laborious process. Although several approaches have been proposed to support automatic program repair, the current automatic program repair techniques are still in the development stage in terms of available research results.

Automatic program repair based on heuristic search is the most widely studied. This technology finds the correct patch in the potential search space according to some heuristic information. Weimer et al. [1–3] first applied genetic programming to automatic program repair in 2009 and formally proposed GenProg [2] in 2012. This approach, combined with a genetic programming algorithm, restricted the influence of program semantic constraints, achieving good repair results. This groundbreaking work has prompted many researchers to explore the automatic program repair approach. Oliveira et al. [4] proposed a new code modification representation for further optimization of genetic algorithms. The code modification representation used in the original GenProg method treats the code and the modification as two separate parts that do not interfere with each other. The code representation proposed by Oliveira et al. encodes the mutation operator and the code element corresponding to the operator on the same chromosome. The advantage of this representation is that the mutation stage can reuse the previous code operations. It is more flexible than traditional genetic algorithms and can improve the patch generation capability

of repair tools. Yuan et al. [5–7] similarly optimized the representation in the genetic algorithm. They represent the code patches as triples to encode the modified location, modified operation type, and reused code elements on the code, respectively. In the patch verification phase, they remove the test cases that are not related to the current patch and the patches that are excluded by setting rules in advance to speed up the verification efficiency. The experimental results show that their method generates about four times more patches than GenProg. However, Le et al. [8] argued that software bugs recur in different applications and that previous bug fixing history can provide effective guidance for fixing patches, so they introduced a third-party data source (i.e., historical repairs) to optimize GenProg. The authors proposed a new bug fixing technique, HistoricalFix, which introduces third-party bug fixing history data to provide better guidance for patch generation than the previous genetic algorithm. For verification on 90 bugs in the Defects4J [9] dataset, GenProg can only fix 1 bug correctly, while HistoricalFix can fix 23 bugs correctly. Another stream of recent research focuses on optimizing the code elements used in the patch generation process. SimFix [10], an automatic repair tool to search for similar codes in projects through program structure and semantic features, achieved good results.

Although the above approaches have achieved progress, search-based automatic program repair still has two major problems: (1) The number of correct patches is small, but the search space involved is large. This usually results in low repair efficiency and repair failure. (2) Search information often cannot guide the search process correctly. It consumes a lot of time and produces many plausible patches that pass test cases but are incorrect.

To tackle these problems, we propose SLARepair, an automatic program repair approach based on location awareness and code similarity. In order to reduce the search space, SLARepair adopts a location-awareness strategy to further subdivide the search space. Specifically, we limit the repair ingredient used to generate the patch to the buggy program itself based on the redundancy assumption [11]. The search space for repair ingredients is divided into classes, packages, and applications, from small to large. A heuristic algorithm preferentially finds repair ingredients in the smallest search space. If no suitable repair ingredients are found, the search space is gradually expanded. This strategy limits the size of the search space to a certain extent so that the search algorithm can quickly find suitable repair ingredients. Compared with the approaches that have been proposed, SLARepair can effectively alleviate the problems caused by the large search space.

For the patch generation process, SLARepair uses code similarity as heuristic information to guide the search process. Recent research shows that the more similar a program is to a buggy program, the more likely it is to contain correct repair. To this end, we design a new fitness function for the genetic programming algorithm to select high-quality offspring populations. It combines the code similarity and the number of failed test cases for subvariant programs to calculate the fitness value. The population selected by SLARepair is more likely to contain the correct repair than other approaches. In addition, we utilized a test case prioritization method combined with test filtering to further reduce the time spent on patch verification.

The main contributions of this paper are as follows:

- An approach for automatic program repair based on code similarity and location-awareness is proposed. It improves repair efficiency by further subdividing the search space.
- A new fitness function is used to calculate the similarity between candidate patches and the original individuals to guide the search process.
- A new test case prioritization approach combined with test case filtering reduces patch verification time.
- Experimental results on Defects4J show the effectiveness of SLARepair, which outperforms the compared approaches.

2. Background

Automatic program repair involves automatically finding a solution to bugs without manual operation and using a certain program specification to automatically convert the buggy program into a program variant that satisfies the specification. Automatic program repair can be transformed into a search problem of finding the correct patch in a potential search space. Genetic programming [12] is a heuristic search method that can effectively represent the evolution process of programs. Automatic program repair typically uses genetic programming to search for the correct program patch. Many studies have shown that the more similar a program is to a buggy program, the more likely it is to contain the correct code. Therefore, code similarity is gradually used as an important heuristic to guide the search process for program repair.

2.1. Heuristic Search

Heuristic search is also called informative search. It uses the heuristic information of the problem itself to guide the search, thereby the scope of the search and reducing the complexity of the problem. Common heuristic search algorithms include genetic algorithms, particle swarm algorithms, ant colony algorithms, simulated annealing algorithms, genetic programming, gene expression programming, etc. In the automatic repair of bugs, the complexity of the program makes it difficult to be encoded, let alone expressed in the form of a fixed-length bit string, which makes the individual heuristic search algorithm with a non-tree structure unable to effectively represent the evolution process of the program. Due to its flexible individual representation, genetic programming can well guide the evolution process of such complex individuals.

2.2. Search Space

The search space consists of three parts. Wherein the suspicious space provides location information for the buggy program. The operation space consists of a series of operators, such as insertion, replacement, and deletion. The ingredient space consists of repair ingredients extracted from the buggy program [2,13–15] or other related programs [15,16]. When repairing a program bug, it is preferred to locate the buggy location information from the suspicious space, then select the appropriate operator from the operation space, and finally use the repair ingredient to generate a program patch. Furthermore, existing repair methods [2,13,14,17,18] show that the redundancy hypothesis, i.e., limiting the ingredient space to the buggy program itself, can achieve good repair results.

2.3. Genetic Programming

Genetic programming [12] is a search method inspired by biological evolution that is essentially an extension of the classical genetic algorithm for programming problems. Unlike genetic algorithms, individuals in genetic programming are computer programs rather than bit strings. In order to facilitate the variation and crossover of individual programs, individuals in genetic programming are generally represented as abstract syntax trees of computer programs. The program's statement or control flow structure corresponds to the nodes of the syntax tree, while crossover and mutation are based on the syntax tree. However, in the current genetic programming algorithm, the complexity of individual programs makes it difficult for the fitness function to accurately describe the strengths and weaknesses of individuals, so the computing resources consumed by each individual program are also difficult to control. In addition, the size of the search space for the correct candidate depends on the size of the program itself. If the target program is large and complex, the search space is almost unlimited. The application of genetic programming to the field of automatic program repair has made great progress in recent years.

2.4. Code Similarity Analysis

Program code similarity analysis is used to measure the degree of similarity between two program codes by certain detection means. Code similarity analysis is mainly applied to program replication detection. In recent years, it has also been applied to automatic program repair, which indicates that similarity analysis plays an important role in the repair process. In CapGen [15], the three similarities of the code context are analyzed and the candidate patches are ranked according to the similarity size. Yokoyama et al. [19] showed that the search space can be reduced by code similarity to find the correct patch. In real repair, the correct patch and the buggy program often have high code similarity. When the code similarity between the candidate patch and the original buggy program is higher, the patch is closer to the correct patch. Therefore, code similarity can be used as important information to guide the patch search process.

3. Our Approach

In this section, we make an introduction to the overall framework. For solving the large search space and improving low repair efficiency, we propose automatic program repair based on code similarity and location awareness.

Our approach contains three stages: fault localization, patch generation, and patch verification, as shown in Figure 1.

In the fault localization phase, we obtain a list of suspicious bug statements through the location approach. Then, we use the suspicious list to filter and prioritize the test cases. Finally, we reconstruct the processed test cases to form a new set of test cases.

In the patch generation phase, we use two sources of repair components: source code and historical code. The longest common subsequence (LCS) algorithm is used to select codes from two repair components to form an initial population, and candidate patches are generated through mutation and crossover.

In the patch verification phase, we verify each candidate patch through test cases and calculate its similarity to the original program. Then, we calculate the fit value of each individual patch and select a new generation group based on the fit value. Repeat the above steps until a patch that passes all test cases is generated. Finally, we performed overfitting tests on the patches that passed all test cases.

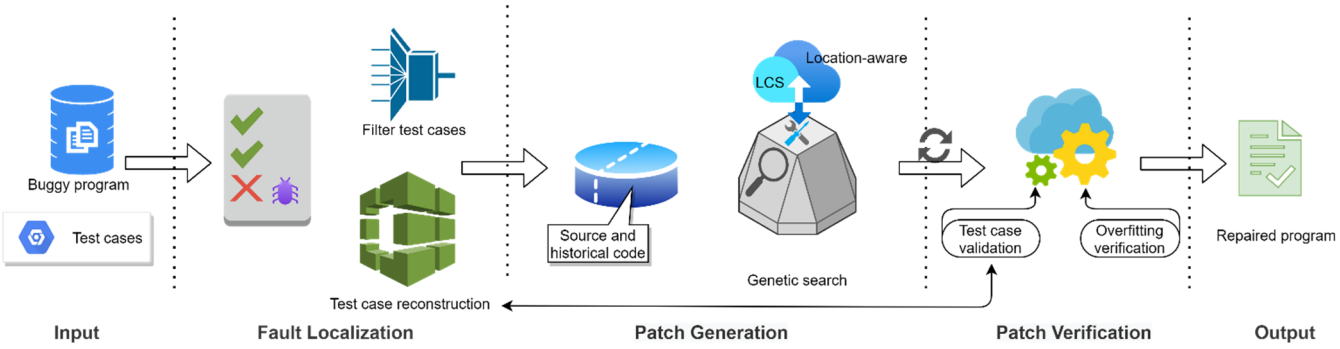


Figure 1. The overall framework of the SLARepair.

3.1. Fault Localization

3.1.1. Fault Localization Tool

For the fault location, SLARepair uses an available spectrum-based approach called Ochiai. It computes the suspicious value of a program entity (p) as follows:

$$susp(p) = \frac{n_{ef}(p)}{\sqrt{n_f \times (n_{ef}(p) + n_{ep}(p))}} \quad (1)$$

Among them, n_{ep} represents the number of successful test cases of covered program entity p, and n_{ef} represents the number of failed test cases of covered program entity p, and n_f represents the sum of the number of failed test cases of covered program entity p and non-covered program entity p. The larger the suspicious value, the more likely it is a buggy program.

Figure 2 is an example of fault localization, which comes from the JFreechart project (Chart12) of the Defects4J [9] benchmark. After the code is located, it is determined that the fault location is line 145. In the subsequent process, SLARepair will search for program patches around this line of code.

```
143  public MultiplePiePlot(CategoryDataset dataset) {
144  super();
145  this.dataset = dataset; // fault 1
146  PiePlot = new PiePlot(null);
147  this.pieChart = new JFreeChart(piePlot);
```

Figure 2. An example of fault localization.

3.1.2. Test Filtering

Figure 3 shows the test filtering process, which is used to speed up patch verification. During the process of fault localization, we record all the code line numbers covered by the test case during its execution. If a positive test case does not pass likely-buggy statements that may include bugs in fault localization, this positive test case is most probably not related to this bug statement. Therefore, this positive test case can be filtered out during the patch verification process. This approach can significantly accelerate the evaluation of patch candidates during the patch repair process.

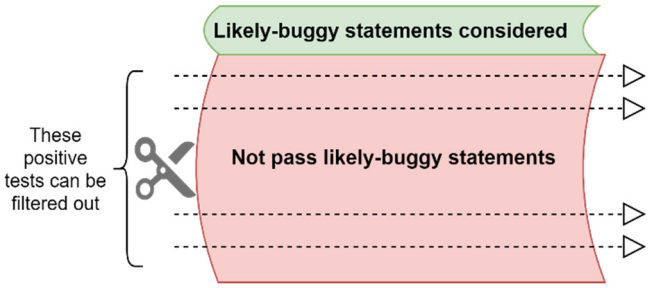


Figure 3. Test filtering overview.

3.1.3. Test Case Prioritization

Figure 4 shows the test case prioritization process. The test cases are divided into 10 groups, and the sorted test cases make it easier to identify invalid patches. The test cases are divided into 5 to 20 groups. Experimental tests showed that the most effective results could be obtained when the test cases were divided into 10 groups. If all the test cases pass, the variant is a trusted patch and enters patch overfitting verification. If the i group of test cases fails, the test verification is stopped, the corresponding indexvalue is added by one, and the 10 groups of test cases are reordered. Finally, we perform a new round of patch validation according to the updated test case prioritization.

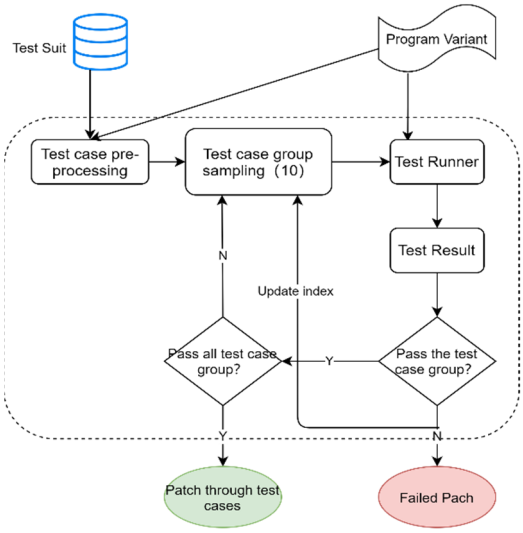


Figure 4. Test case prioritization overview.

3.2. Patch Generation

The task of the patch generation phase is to find potential candidate patches for the buggy code. At this stage, SLARepair uses code similarity information to guide the search process of genetic programming. The longest common subsequence algorithm is used to calculate the similarity between codes. At the same time, we combine the location-awareness strategy to more effectively improve search efficiency. Figure 5 is an example of a candidate patch generated by SLARepair for the fault in Figure 2. The green line is a candidate patch.

```
143   public MultiplePiePlot(CategoryDataset dataset) {
144   super();
145 −  this.dataset = dataset;
145 +  setDataset(dataset);
146   PiePlot = new PiePlot(null);
147   this.pieChart = new JFreeChart(piePlot);
```

Figure 5. An example of patch generation.

3.2.1. Longest Common Subsequence

Considering the efficiency of repairing algorithms and the complexity of buggy programs, we chose the longest common subsequence (LCS) algorithm to measure the similarity between the codes. LCS can take into account both the attribute characteristics and structural characteristics of the program in the process of repairing bugs. Program characteristics are represented by identifiers in the program. Program identifiers include common operators, keywords, standard method names, and user-defined identifiers. The attribute characteristics of the program are obtained by classifying and collecting the identifiers in the program. The structural characteristics of a program are captured by the order in which identifiers appear in the program. The LCS algorithm calculates the similarity by comparing the program characteristics. The sequential identifier sets of the two program segments are respectively deleted with zero or more identifiers, but the order of the remaining identifiers is not changed. The resulting longest strictly increasing sequence is called the longest common identifier subsequence.

The longest common subsequence (LCS) algorithm is used to measure the similarity between codes at the character level. For two sequences $X_i = (x_1, x_2, \ldots, x_i)$ and $Y_j = (y_1, y_2, \ldots, y_j)$, the following recurrence relation is used to calculate the LCS length:

(1) when $i = 0$ or $j = 0$, the LCS length of X_i and Y_j is zero.

(2) when $x_i = y_j$, the LCS of X_i and Y_j is the LCS of X_{i-1} and Y_{j-1} plus the LCS of x_i (or y_j);

(3) when $x_i \neq y_j$, the LCS of X_i and Y_j is the longer of the LCS of X_{i-1}, Y_j and the LCS of X_i, Y_{j-1}.

Assuming that $c[i][j]$ represents the length of the LCS of X_i and Y_j, the following calculations are performed:

$$c[i][j] = \begin{cases} 0 & i = 0 \text{ or } j = 0 \\ c[i-1][j-1] + 1 & i, j > 0 \text{ and } x_i = y_i \\ \max(c[i-1][j], c[i][j-1]) & i, j > 0 \text{ and } x_i \neq y_i \end{cases} \quad (2)$$

3.2.2. Genetic Programming Based on Code Similarity

In this section, we give an automatic program repair approach for genetic programming based on code similarity as follows:

(1) Localization process: First, use the fault localization tool to obtain a list of suspicious locations. Then, the suspicious locations are screened according to the preset minimum and maximum suspicious location numbers. Finally, combine these suspicious locations into a suspicious space.

(2) Population initialization: The initial population is created by means of code similarity, where each individual program has the same suspicious space.

(3) Mutation process: First, a parent variant is selected from the current population by fitness value. Then, different suspicious locations and operators are selected from the suspicious space and the operation space, respectively. Next, set the ingredient space and select the fixing ingredient. Finally, create a program variant and repeat the process until you exceed the specified number of mutations. Among them, operators and repair ingredients are selected by uniform random selection. The method of weighted random selection is adopted for the selection of suspicious locations. The probability calculation for each suspicious location is shown in Formula (3):

$$P_w(sl_i) = \frac{sv_i}{\sum_{j=1}^{n} sv_j} \quad (3)$$

where sl_i represents the ith suspicious location. sv_i represents the suspicious value of the ith suspicious location. n is the total number of suspicious positions. $P_w(sl_i)$ represents the probability of selecting the suspicious location sl_i.

(4) Crossover process: First, a child variant and a parent variant are randomly selected. Second, select a modified suspect location in the suspect space of the variant. Then, truncate the node corresponding to the suspect location on the two variants of the abstract syntax tree and swap the subtree. Finally, repeat the above steps until the maximum number of mutations is reached. This is a non-standard version of the crossover, and the reason for this crossover is that some mutations can cause irreparable damage to the program's functionality, which provides a way to preserve the original functionality of the program. Crossover of the nodes of the abstract syntax tree can effectively prevent the generation of code that does not conform to the compilation rules.

(5) Selection process: First, stop the fix and output the patch if a variant passes all test cases; otherwise, record the number of failed test cases. Then, the fitness values of the variants are calculated based on similarity and the number of failed test cases. Finally, repeat the procedure until all the variants have been verified and a new generation population is selected according to the fitness value. In this paper, automatic program repair is regarded

as a minimization problem. A program variant with a lower fitness value has a higher probability of being selected. The fitness function is shown in Formula (4):

$$Fitness\ (v_i) = \frac{|\ \{t \in T\ |\ (v_i)\ Faling\ (t)\}\ |}{Sim_{v_i}(v_i, v_0)} \tag{4}$$

where $Fitness(v_i)$ represents the fitness function of the variant v_i. v_i and v_0 respectively represent the ith program variant and the original program. T stands for test case set, t stands for test case, and $Failing(t)$ stands for failed test case; $Sim_{v_i}(v_i, v_0)$ represents the similarity between the program variant v_i and the original program v_0. $Sim_{v_i}(v_i, v_0)$ is shown in Formula (5):

$$Sim_{v_i}(v_i, v_0) = \frac{\sum_{i=1}^{n} Sim_{sl_i}}{n} \tag{5}$$

where sl_i represents the ith suspicious location. Sim_{sl_i} represents the similarity between the variant v_i and the code area where the original program v_0 is located in the suspect location sl_i.

3.2.3. Location-Awareness Repair

The location-awareness repair approach further subdivides the search space for repair components. Based on the concepts of package and class in a Java program, we divide the code in the buggy program into three types: Code that belongs to a class; code that belongs to the package; and code that belongs to the entire application. According to different code sources, three ingredient space types are constructed: Class (Cl), package (Pkg), and application (App). When the ingredient space is class, package, or application, the search space for the fixing ingredient is class, package, or the entire application where the suspect location code resides.

Further narrowing the ingredient space from Application to Class or Package can effectively improve the search efficiency, but the correct repair ingredient may not exist in the class or package. Therefore, location-awareness repair does not discard any kind of ingredient space, but gradually expands the search scope of the repair ingredient. Figure 6 shows the changing process of the ingredient space in location-awareness repair. We first try to search in the minimum ingredient space, and gradually expand the search scope if the correct patch is not generated within the limited conditions. It can be seen from the figure that after the ingredient space expands from Class to Package, we give up searching for repair ingredients in Class and search for other repair ingredients in Package. Therefore, location-awareness repair can improve repair efficiency and accuracy simultaneously.

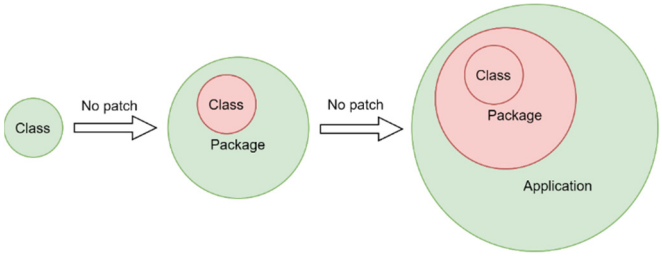


Figure 6. The changing process of ingredient space location-awareness repair.

3.3. Patch Verification

The patch verification phase is used to verify the validity of the patch. It usually requires two processes: test case testing and overfitting testing. A test case is passed when a candidate patch passes all test cases in the test suite. A patch is tested for overfitting by means of a patch overfitting test.

We will briefly describe the execution of our method. First, a reasonable patch and a buggy program are input to our method, and the method modified by the patch is marked.

Then, some of the input-output pairs can be caught by detecting the bytecode of these methods in the buggy program and running positive test cases. Assuming that we will end up with k such pairs, using Set PA = {(In_1, Out_1), (In_2, Out_2), ..., (In_k, Out_k)}. From the hypothesis, it seems that all input-output pairs reflect the correct program behavior. We put these inputs In_1, In_2, ..., In_k into the appropriate methods in the patch program. A set of outputs is obtained and recorded. If there is a difference between the two sets of inputs, then we can conclude that the plausible patches fail the overfitting test.

The candidate patches shown in Figure 5 pass the test case test and the overfitting test, which are called correct patches. After analysis, the patch is equivalent to the manual patch.

4. Empirical Study

In this section, we will explain the research questions, the datasets of bugs used, the evaluation standards, the experimental setup, and the experimental results.

4.1. Research Questions

To conduct a general evaluation of SLARepair, we set the following research questions in this study:

RQ1: Does SLARepair really work better than GenProg's automatic program repair?
RQ2: How does the SLARepair repair method compare with other existing advanced repair methods?

4.2. Datasets of Bugs

An effective evaluation of the effectiveness of an automatic program repair approach relies on high-quality benchmark datasets. To evaluate proposed techniques, researchers need to create datasets or rely on publicly available bug datasets. The latter is critical to advancing research in the field of automatic program repair because the publicly available datasets can better reproduce the experimental process and facilitate comparison between different methods. Therefore, Defects4J [9] was used as the experimental object to evaluate our approach. Defects4J is a database and extensible framework provided by JUST et al. at the University of Washington, USA. It provides real bugs to enable reproducible studies in software testing research. This benchmark has been widely used in software-engineering related fields, including automatic program repair [20].

This study selects Defects4J 1.5.0 and involves 4 projects: JFreeChart, Apache Commons Lang, Apache Commons Math, and Joda-Time. We exclude the Closure Compiler project because the test cases in the Closure Compiler are organized in an unconventional way, using a custom test format rather than the standard JUnit test format. Table 1 shows the statistical data for 224 real bugs in 4 projects.

Table 1. Defects4J contains statistics of bugs.

Subject	Subject Id	Bugs	KLOC	Test Cases
JFreechart	Chart	26	96	2205
Commons Lang	Lang	65	22	2245
Commons Math	Math	106	85	3602
Joda-Time	Time	27	28	4130
Total		224	231	12,182

4.3. Evaluation Standard

Based on existing research, the following three evaluation standards were used as repair capability measurements for different repair methods in this paper: (1) The number of repairable bugs; (2) the time consumption of successful repair; (3) the total number of candidate patches to be generated for successful repair. In the above evaluation standards, the number of repairable bugs and the time consumption of a successful repair can directly reflect the accuracy and efficiency of the repair approach. The total number of candidate

patches that need to be generated to successfully repair an item can indicate how difficult it is for the repair method to generate the correct patch. A study [21] pointed out that for an intelligent search algorithm with randomness, its search ability can be measured by the number of fitness evaluations in the search process. Specific to the repair method in this paper, the total number of candidate patches generated can effectively represent the number of fitness evaluation times. Therefore, the rest of this section focuses on discussing the repair capabilities of SLARepair using the above three evaluation standards.

4.4. Experimental Setup

The experimental setup for GenProg [2] and SLARepair in this paper was the same. The minimum suspicious value and maximum suspicious value of suspicious statements are set to 0.5 and 50, respectively, in the fault localization phase. In addition, this paper follows the existing practice of setting the maximum repair time budget for each bug at 90 min.

For location-awareness repair specific to SLARepair, the maximum search time for SLARepair in each ingredient space was set at 30 min.

4.5. Experimental Result

4.5.1. RQ1: Does SLARepair Really Work Better Than Genprog's Automatic Program Repair?

Table 2 summarizes the GenProg and SLARepair repair performance on Defects4J. GenProg has successfully repaired 25 bugs on Defects4J, with a success rate of 11.2%. SLARepair can repair 43 bugs with a success rate of 19.2%. SLARepair improves repair accuracy by 72% compared with GenProg, mainly because code similarity information in SLARepair can better guide the whole search process. Due to the lack of code similarity guidance information, GenProg failed to generate a valid patch for the remaining 18 bugs within a limited time or iteration.

Table 2. Overview of GenProg and SLARepair repair performance on Defects4J.

Subject Id	Bugs	Bug Id	
		GenProg	SLARepair
Chart	26	C1,C3,C5,C7, C13,C15,C25	C1,C3,C5,C7,C12,C13, C15,C19,C25,C26
Lang	65	-	L7,L10,L22,L24,L27,L39
Math	106	M2,M5,M8,M28,M40,M49,M50, M53,M70,M73, M80,M81,M82, M84,M85,M95	M2,M5,M8,M20,M28, M32,M39,M40,M49,M50, M53,M56,M60,M64,M70,M71, M73,M74,M78,M80,M81,M82, M84,M85,M95
Time	27	T4,T11	T4,T11
total	224	25	43

In order to further compare the repair efficiency of SLARepair and GenProg, Table 3 displays the specific running time and the number of candidate patches generated for the 25 bugs that can be repaired by both SLARepair and GenProg. It takes at least 3 min and at most 57.6 min for SLARepair to successfully repair a bug, with a total time of 491.1 min and an average time of 19.6 min. GenProg needed at least 7.8 min and at most 85.9 min to successfully repair a bug, which took a total of 1114.8 min and an average of 44.6 min. Compared with GenProg, SLARepair reduced the total repair time by 55.9% and the average repair time per bug by 56.1%. As above, from the point of view of patch generation, SLARepair reduces the total number of patch generations by 51%, and the average number of patch generations per bug can be reduced by 50.8%. Therefore, in the case that both repair methods can successfully repair, SLARepair improves the repair efficiency by at least 50% compared with GenProg. On the other hand, the time spans of SLARepair and GenProg on the 25 bugs are 54.6 min and 78.1 min, respectively, while the

time spans of the number of candidate patches are 92 and 131 respectively. This shows that the difficulty of repair can vary greatly for different bugs.

Table 3. Comparison of repair efficiency of SLARepair and GenProg on repairable bugs.

Subject	Bug Id	Approach	Time (min)	Candidate Patches	Reduced Repair Time	Reduced Candidate Patches
Chart	C1	SLARepair GenProg	6.1 11.3	11 29	46.0%	62.1%
	C3	SLARepair GenProg	10.0 34.0	23 71	70.6%	67.6%
	C5	SLARepair GenProg	8.7 17.0	21 40	48.8%	47.5%
	C7	SLARepair GenProg	9.6 16.4	21 34	41.5%	38.2%
	C13	SLARepair GenProg	12.1 24.7	19 34	51.0%	44.1%
	C15	SLARepair GenProg	16.7 34.7	42 82	51.9%	48.8%
	C25	SLARepair GenProg	15.8 33.3	21 40	52.6%	47.5%
Math	M2	SLARepair GenProg	29.5 74.5	18 40	60.4%	55.0%
	M5	SLARepair GenProg	18.2 67.3	11 64	73.0%	82.8%
	M8	SLARepair GenProg	21.9 36.5	12 15	40.0%	20.0%
	M28	SLARepair GenProg	28.0 85.9	18 48	67.4%	62.5%
	M40	SLARepair GenProg	57.6 84.7	30 44	32.0%	31.8%
	M49	SLARepair GenProg	49.0 66.4	34 45	26.2%	24.4%
	M50	SLARepair GenProg	8.0 21.6	15 20	63.0%	25.0%
	M53	SLARepair GenProg	18.1 85.9	14 65	78.9%	78.5%
	M70	SLARepair GenProg	51.4 83.8	24 37	38.7%	35.1%
	M73	SLARepair GenProg	19.4 82.1	16 98	76.4%	83.7%
	M80	SLARepair GenProg	3.0 13.2	19 80	77.3%	76.3%
	M81	SLARepair GenProg	4.8 23.6	26 35	79.7%	25.7%
	M82	SLARepair GenProg	21.6 59.0	68 140	63.4%	51.4%
	M84	SLARepair GenProg	22.5 40.7	67 110	44.7%	39.1%
	M85	SLARepair GenProg	20.1 42.5	103 145	52.7%	29.0%
	M95	SLARepair GenProg	23.6 51.0	45 90	53.7%	50.0%

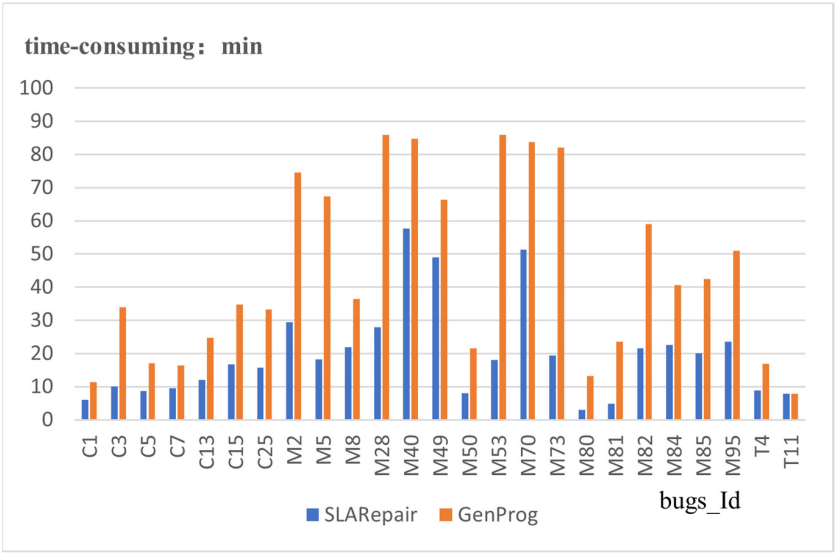
Table 3. *Cont.*

Subject	Bug Id	Approach	Time (min)	Candidate Patches	Reduced Repair Time	Reduced Candidate Patches
Time	T4	SLARepair GenProg	8.9 16.9	23 36	47.3%	36.1%
	T11	SLARepair GenProg	6.5 7.8	12 14	16.7%	14.3%
average		SLARepair GenProg	19.6 44.6	29 59	56.1%	50.8%
total		SLARepair GenProg	491.1 1114.8	713 1456	55.9%	51.0%

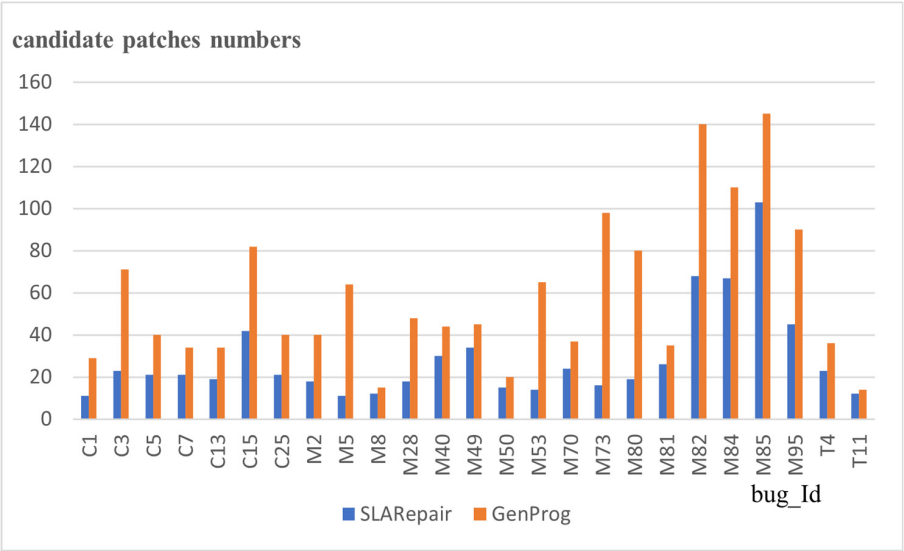
In order to compare the repair efficiency more intuitively and discuss the differences between different bugs, we drew a bar graph of the execution time of the two repair methods and the number of candidate patches generated based on the data in Table 3, as shown in Figure 7. Among them, the horizontal axis in Figure 7a,b represents the bug Id, and the vertical axis represents the running time (unit: minutes) and the total number of candidate patches. In Figure 7a, it can be clearly observed that the running time of SLARepair is significantly reduced compared with GenProg. In Figure 7b, the change in the number of candidate patches shows the same law as in Figure 7a. Figure 7 intuitively shows that SLARepair's repair ability is better than GenProg's. In Figure 7a,b, the running time and the number of candidate patches generated are quite different between different bugs, which shows that the difficulty of repairing each bug is very different. This phenomenon will exist even in different buggy versions of the same software project. For example, in the Math project, there are 16 different buggy versions from M2 to M95, and the time consumption and the number of candidate patch generations will also fluctuate greatly. One of the reasons for this phenomenon is that the faults in the Defects4J bug dataset come from actual development, and the Math project has been developed for 11 years, so there are certain differences in the software composition of different versions of Math faults. Another reason for this phenomenon is that the bug program in Defects4J is very complicated and the program scale is large, which leads to different costs for repairing different bugs even in the same software project. The above reasons show that in the actual industry, the automatic repair of software faults is a discrete problem with weak regularity.

It can be seen from the above that the repair ability of SLARepair is far better than that of GenProg, and the improvement of this repair ability comes from the guidance of code similarity information and the location-awareness repair. Code similarity information can enable SLARepair to search for the correct patch faster based on the original genetic programming, which is reflected in the fact that SLARepair can successfully repair with fewer candidate patches. Another reason why SLARepair has better repair capabilities is that the location-awareness repair approach greatly reduces the ingredient space. In order to verify the improvement of the location-awareness repair approach for repairing capabilities, we run SLARepair with the search scope of the repair ingredient limited to application, package, and category according to the description of location-awareness repair and calculate the specific repair time. For applications, the maximum time settings on packages and classes are 30 min, 60 min, and 90 min, respectively. Considering the cost of the experiment, we only experimented on the location-awareness repair capability of SLARepair on the Math project, and the experimental result is an average of 20 runs. Figure 8 shows the time consumption of each Math bug in different ingredient spaces in the form of a histogram, where the horizontal axis represents the ingredient space type and the vertical axis represents the repair time (unit: minutes). It can be seen from the figure that as the ingredient space shrinks from application to class, the repair time is declining, which shows that location-awareness repair can effectively improve the repair efficiency of SLARepair. In particular, in Figure 8d,j, the time consumption of the package is higher than

the application. This is because, in the actual repair process of the M20 bug and the M50 bug, too many complex invalid patches were generated in the package space during the actual repair process, resulting in excessive verification time. From Figure 8, we can also see that in some of the bugs, SLARepair cannot find the correct patch in a certain search space. For example, bugs M39, M40, M49, M70, and M74 cannot find effective patches in the class space. This is because the repair ingredients needed to repair these bugs are in a larger space or the time required to find the correct repair ingredients exceeds 30 min. The bugs M56, M71, and M78 cannot find effective patches in the application space. This is because the ingredient space is too large and the repair time exceeds the pre-set 90 min. The most special thing is that M60 can only be repaired correctly within the scope of application, because the correct repair ingredients only exist in another package. For the above special cases, the search space is automatically expanded when the location-awareness repair approach cannot complete the repair in a smaller search space. Therefore, SLARepair's location-awareness repair capabilities can improve repair efficiency while ensuring repair accuracy.



(**a**)



(**b**)

Figure 7. SLARepair and GenProg running time on repairable bugs and generating a bar graph of candidate patches. (**a**) SLARepair and GenProg running time spent on repairable bugs. (**b**) SLARepair and GenProg generate candidate patches for repairable bugs.

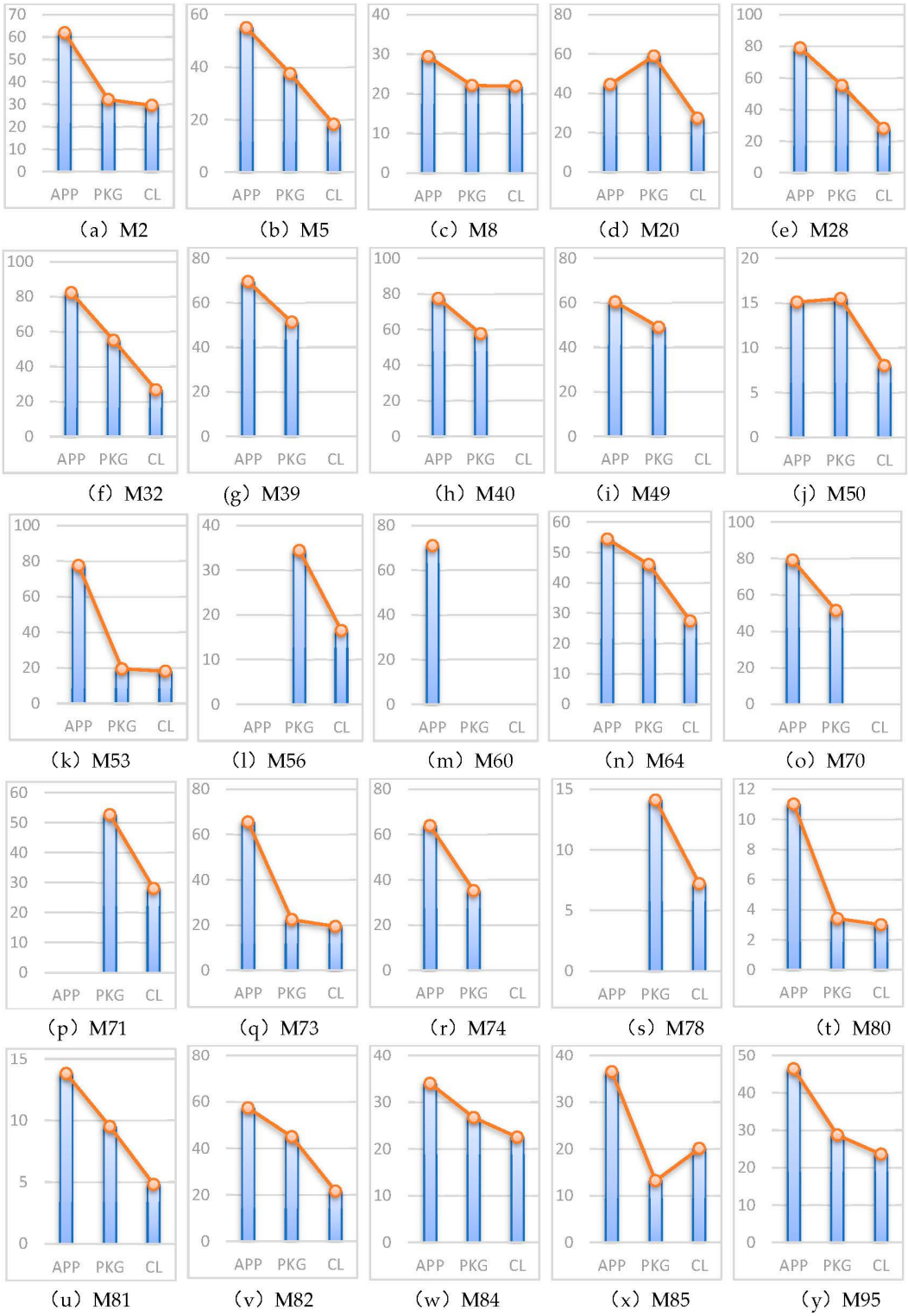


Figure 8. Time to repair math project bugs on SLARepair in three ingredient spaces of class, package, and application.

4.5.2. RQ2: How Does the SLARepair Repair Method Compare with Other Existing Advanced Repair Methods?

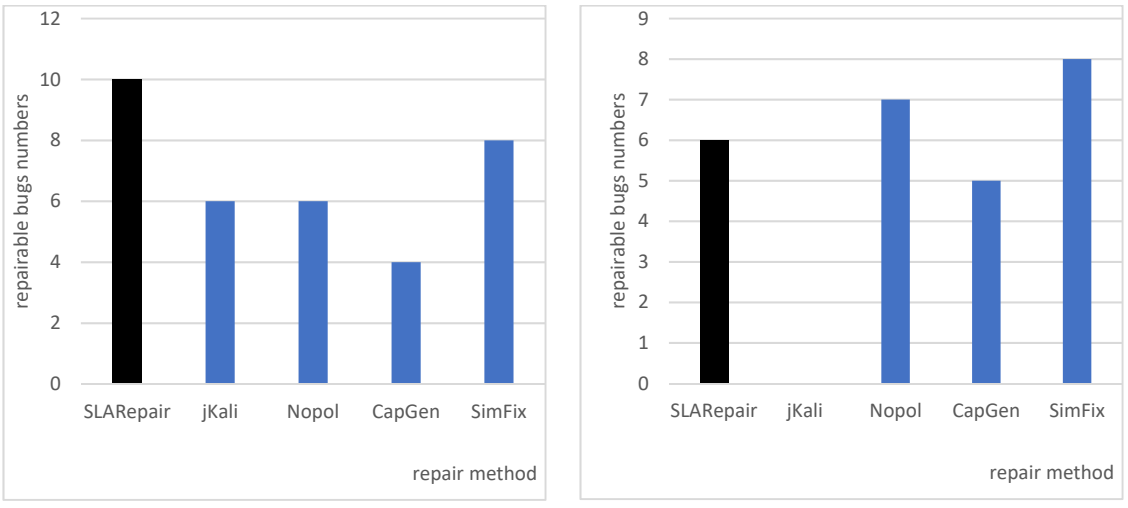
We chose the popular repair methods jKali [22], Nopol [23], CapGen [15], and SimFix [10] to compare with this study, where jKali is the Java implementation version of the Kali [24] repair method on Defects4J. The experimental data of jKali and Nopol are based

on the display of Martinez et al. in [22]. Martinez et al. verified the actual repair effect of jKali and Nopol on Defects4J under the same hardware and software conditions. The experiment uniformly uses the 180-min operating limit, so the experimental data in [22] is more authentic and credible. The experimental results of CapGen and SimFix come from their corresponding research papers, in which the upper limit of CapGen repair time is 90 min and the upper limit of SimFix repair time is 300 min. As shown in Table 4, SLARepair can successfully repair 43 bugs out of 224 bugs in Defects4J, with a success rate of 19.2%. In other repair methods, a maximum of 35 bugs can be repaired. Therefore, the repair ability of SLARepair is superior to these repair methods.

Table 4. Comparison of the number of bugs repaired by SLARepair and the existing four methods.

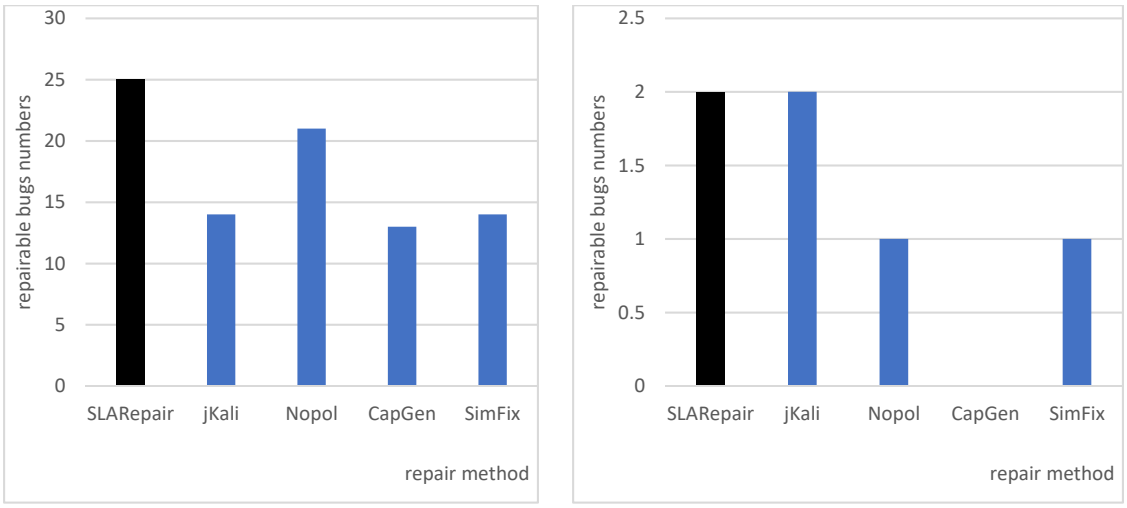
Project	SLARepair	jKali	Nopol	CapGen	SimFix
Chart (26)	10	6	6	4	8
Lang (65)	6	0	7	5	8
Math (106)	25	14	21	13	14
Time (27)	2	2	1	0	1
Total (224)	43	22	35	22	31
success rate	19.2%	9.8%	15.6%	9.8%	13.8%

Figure 9 more intuitively shows the repair quantity of SLARepair and the other 4 repair methods on the Chart, Lang, Math, and Time projects. We can see that in the Chart project shown in Figure 9a and the Math project shown in Figure 9c, the number of bugs that SLARepair can repair is significantly higher than the other four methods. It can be seen from Figure 9d that in the Time project, SLARepair and jKali can repair two bugs, while the remaining three repair methods can only repair one at most. This is because the Time project is too complex, resulting in a small number of repairable faults in the existing repair methods. As shown in Figure 9b, SLARepair has poor repair performance on the Lang project. The number of repairable bugs is one less than Nopol and two less than SimFix. This is because the correct repair ingredients for most of the bugs in the Lang project cannot be searched at the statement granularity level of the program itself, while Nopol can synthesize patches through constraint solving at a finer program granularity, and SimFix can also repair these bugs with the help of external repair templates.



(**a**) (**b**)

Figure 9. *Cont.*



(**c**) (**d**)

Figure 9. Compares the number of bug repairs on Defects4J between SLARepair and 4 repair methods. (**a**) Chart, (**b**) Lang, (**c**) Math, (**d**) Time.

Finally, we discuss the time-consumption problem of SLARepair's patch verification during the repair process. We found that patch verification consumes a huge amount of time when repairing bugs. Figure 10 shows the time-consumption situation of SLARepair's patch verification on 43 repairable bugs. The horizontal axis represents the bug Id, the main vertical axis represents time (unit: minutes), and the secondary vertical axis represents the percentage of patch verification in the total repair time. It can be seen from Figure 10 that for most bug repairs, patch verification takes much longer than other runtimes, such as positioning and searching. Patch verification accounts for the smallest proportion (37.4%) of the total repair time and the largest (98.7%). SLARepair repairing a bug requires an average of 79.4% time consumption for patch verification. The reason for this phenomenon is that SLARepair needs to rely on the fitness function to guide the patch search process, and the calculation of the fitness function is determined by the code similarity and the number of passed test cases. The calculation of code similarity includes collecting the program identifier of the buggy program and the candidate patch program at each suspicious location and calculating the length of the longest common subsequence of the identifier, which does not require excessive computing resources. Calculating the number of test cases passed by the patch needs to drive the entire buggy program to execute all test cases, which requires a lot of computing resources. On the other hand, from Table 3, SLARepair needs to generate 29 candidate patches on average to repair a bug, which means that every time a bug is repaired, the entire bug program needs to be driven to execute thousands of test cases dozens of times. Therefore, patch verification consumes huge computing resources in the repair process. This is also an important problem in all current search-based automatic program repair methods.

In summary, the repair ability of SLARepair under the three evaluation criteria is significantly better than the original GenProg repair method based on genetic programming. On the one hand, SLARepair's higher repair ability comes from code similarity, which effectively guides the patch search process. On the other hand, it comes from the reduction of the search space caused by location-awareness repair. In terms of the number of repairable faults, SLARepair is also superior to other repair methods.

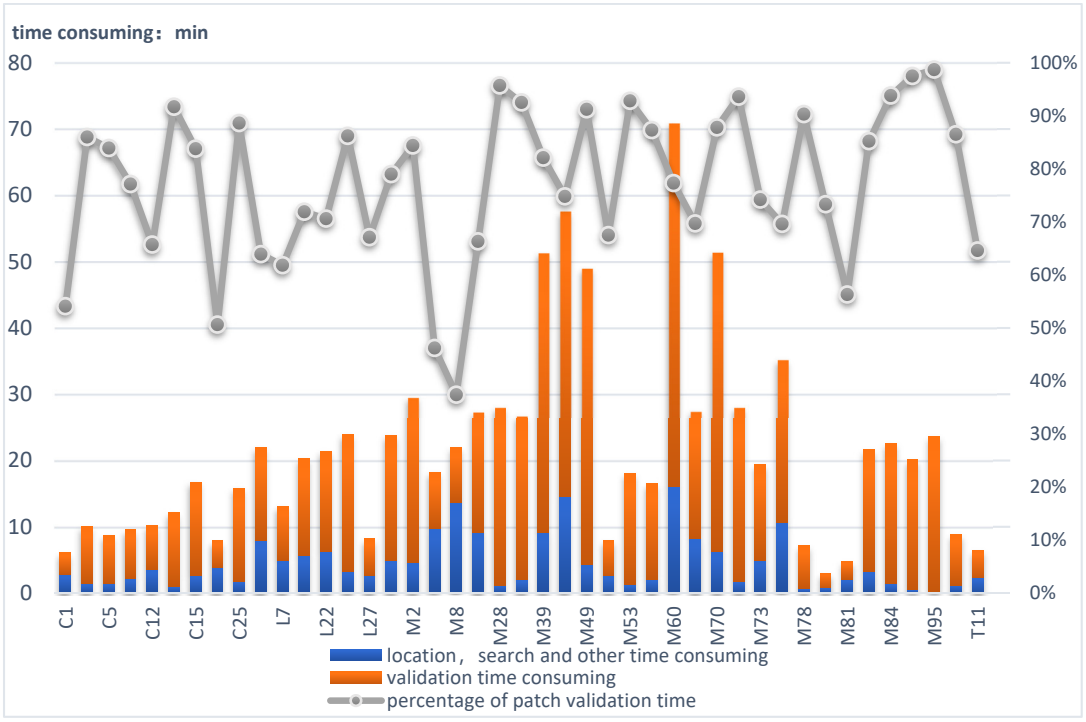


Figure 10. Time spent on patch verification during SLARepair.

5. Discussion

Genetic programming can find the correct patch in the potential search space. Like Genprog [2], its main problem is that the large search space only contains a small number of correct patches. This can lead to consuming a lot of time looking for candidate patches or failing to search. In order to reduce the search space, we combine the search information with code similarity to guide the repair process. Experiments show that code similarity guides the right direction for genetic programming and significantly improves the repair effect. Secondly, we further optimize the search space through our location-awareness strategy. Experiments have shown that the strategy significantly improves repair efficiency.

In research question 1, it can be seen that SLARepair outperforms Genprog in repair success rate and repair efficiency from Tables 2 and 3, Figure 7. The main reasons are three-fold. The first point is that SLARepair combines code similarity to guide the repair process, while Genprog uses only genetic programming. Genetic programming randomly selects repair ingredients, which, when combined with code similarity, allows for a more efficient selection of suitable repair ingredients. The second point is that the location-awareness strategy makes it easier for SLARepair to find the correct patch. The improvement of the repair ability is verified after the location-awareness strategy subdivides the search space from Figure 8. The third point is that SLARepair uses the test case prioritization method combined with test filtering to further reduce the time spent on patch verification.

In research question 2, we compare SLARepair with other methods. Although CapGen [15] and SimFix [10] also involve code similarity analysis, it can be seen that code similarity analysis combined with location-awareness strategies can lead to better repair results from Table 4 and Figure 9. At the same time, we can also see the complexity of fixing bugs in Figure 9. Furthermore, we should also note that despite the test case prioritization method and test filtering techniques, the patch verification process is still time-consuming.

6. Related Work

In this section, we review related work on automatic program repair. We focus on two aspects. One is genetic programming. Genetic programming is the most efficient way to search for the correct patch. The second is code similarity. Recent studies have shown that

code similarity plays an increasingly important role in automatic program repair. Finally, we compare SLARepair with other methods and analyze the advantages of our approach.

6.1. Genetic Programming-Based Repair Approaches

Le Goues et al. [2] proposed GenProg. The basic idea is that the code is redundant, so the patch in the reuse project has the probability of generating the correct patch. During the patch search process, mutation and crossover operations are used to increase the patch search space. The population is selected by the number of test cases passed by the patch to iterates until there is a patch that passes all test cases. This work has important implications for automatic program repair and has prompted more researchers to explore automatic program repair. However, GenProg is in the initial stage of the automatic repair approach, and the content has not yet matured. Qi et al. [14] believe that the fitness selection function in genetic programming does not play a significant role in candidate patch selection, so random search is used instead of genetic search. It was found that the random search consumed less time to generate the patches generated by GenProg. Oliveira et al. [4] proposed a new code modification representation to further optimize the genetic algorithm. They encoded the code representation method on the same chromosome with the mutation operator and the code element corresponding to the operator. This method can realize different mutation operations on the same code, which is more flexible than the traditional genetic algorithm and can improve the patch generation capability of repair tools. Mehne et al. [25] accelerated the repair process by filtering the test cases in the program, and Sun et al. [26] improved the original GenProg repair method by optimizing the population initialization and mutation processes of the genetic algorithm. Then, Li et al. [27] proposed an approach that combines search-based automatic program repair techniques with a neural machine translation-based approach, and Villanueva et al. [28] used novel search algorithms to optimize GenProg's search and avoid getting trapped in local optimal solutions. Because of the limited variety and simplicity of structures in bytecode, the patch space covered by the mutation operator is larger. In addition, bytecode modification can be run directly on the JVM without additional compilation, so it is more efficient. The actual verification results also show that this method is more efficient than the existing techniques.

The major difference between SLARepair and GenProg lies in the candidate patch stage, where we improve patch generation efficiency by filtering candidate patches using code similarity. In order to improve search efficiency, RSRepair [14] uses random search and abandons the use of fitness functions to guide patch generation. While SLARepair optimizes the above aspects, it also improves population initialization. The methods proposed by Sun et al. and Villanueva et al. are to improve patch quality by optimizing the way to guide patch generation. SLARepair not only optimizes bootstrap patch generation but also focuses on the efficiency of patch generation.

6.2. Code Similarity-Based Repair Approaches

The previous section mainly considered the optimization of search strategy and code modification, while this section focuses on the connection between bug code and reused code. Ji et al. [29] proposed to fix bugs by using program fragments similar to buggy programs in the project and implemented the bug repair tool SCRepair. SCRepair measures the syntax tree structure consistency between the buggy code and the reference code by comparing code similarity and measuring the previous difference in code using ChangeDistiller [30] to extract the bug code from the reference code modification operation. The reused reference code is filtered according to predefined similarity and difference thresholds. Lin et al. [31] found that existing techniques rarely consider the contextual information and program structure of the generated patches, which are crucial for the evaluation of patch correctness as revealed by existing research. Like the above approach, ssFix, proposed by Xin et al. [32], searches the code base for codes similar to buggy programs as repair ingredients. SsFix applies modification operations one by one to the differences between defective codes extracted by ChangeDistiller and similar codes, without directly

reusing similar codes. Experiments on the Defects4J dataset show that ssFix can correctly fix 20 bugs. Based on the ssFix theory, Jiang et al. [10] argue that code from the same project has better reference value. SimFix uses existing patches to generate a search space S1, while the program to be fixed forms a search space S2. A heuristic search is performed at the intersection of these two search spaces. Wen Ming et al. [15] used the contextual information of the AST nodes to repair the buggy programs. They used a fine-grained design to find more correct repair components. Not only the context-aware priority of the employed mutation operator can constrain the search space, but also the three context-aware models can prioritize the correct patches. Motwani et al. [33] empirically assess the quality of program fixes for real-world Java bugs. They developed JaRFly, outlined bugs, and built a methodology and dataset for evaluating the quality of new repair technology patches for real-world bugs. Phung et al. [34] proposed MIPI, a new method for reducing the number of overfitting patches generated in APR. They used the similarity between the names of patch methods and the semantic meanings of the method bodies to identify and remove the overfitting patches generated by APR tools.

CapGen [15] only considers the frequency of the code when reusing existing code in a project and does not even consider the characteristics of the code or the connection between the buggy code and the reused code. To address this problem, Ji et al. [29] proposed to fix bugs by reusing code ingredients similar to the defective code in this project. Different from them, SLARepair and CRSearcher, by reusing not only code in the project but also code in other projects, increase the patch search space. Compared with the SLARepair approach, ssFix has a finer granularity of code reuse, which results in a larger patch space and a lower accuracy rate. Phung et al. [34] exploit the similarity between the name of the patch method and the semantics of the method body from a new perspective to reduce patch overfitting. Unlike SLARepair, SimFix extracts code modifications that can be applied in combination and filters and optimize candidate patches using common modifications from historical fixes.

7. Conclusions

To resolve the problem of large search spaces and low search efficiency, we propose an automatic program repair based on code similarity and location awareness. First, in order to improve search efficiency, SLARepair combines code similarity with guided genetic programming. We designed a new fitness function for genetic programming to more efficiently select suitable repair ingredients. In addition, we use a location-awareness strategy to further reduce search space, which significantly improves repair efficiency. Finally, we utilize the test case prioritization approach combined with test case filtering to further reduce patch verification time. Experimental results show that SLARepair has higher repair efficiency and accuracy than existing automatic program repair approaches. SLARepair can repair 43 faults with a success rate of 19.2%, which is 3.6~9.4% higher than other repair tools. Compared with GenProg, SLARepair improves repair accuracy by 72%.

In the future, we plan to study the following issues: We want to consider the automatic repair of software faults in a multi-fault environment. The automatic program repair works at the sentence level of the abstract syntax tree, but for some faults, the correct repair ingredient may exist at a finer granularity (i.e., variable and expression) than the sentence. We will continue to refine the repair granularity and use fine-grained repair ingredients to generate patches.

Author Contributions: Conceptualization, H.C. and F.L.; methodology, H.C. and D.H.; software, F.L.; formal analysis, F.L.; investigation, D.H.; resources, T.L.; data curation, T.L., C.Z. and J.S.; writing—original draft, H.C.; writing—review and editing, D.H.; visualization, C.Z. and J.S.; supervision, T.L. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by the Natural Science Foundation of China (Nos. 62206087, 62276091, and 61602154), the Innovative Funds Plan of Henan University of Technology (No: 2022ZKCJ14), the Key Laboratory of Grain Information Processing and Control (Henan University of Technology), the Ministry of Education (No. KFJJ2022006), the Cultivation Programme for Young Backbone Teachers in Henan University of Technology, the Key scientific research projects of colleges and universities in Henan Province (No. 22A520024), the Major Public Welfare Project of Henan Province (No. 201300311200), and the Science and Technology Research Project of Henan Province (Nos. 232102210186, 222102210140).

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: All the data used in this paper can be traced back to the cited references.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Weimer, W.; Nguyen, T.V.; Goues, C.L.; Forrest, S. Automatically finding patches using genetic programming. In Proceedings of the 31st International Conference on Software Engineering, Vancouver, BC, Canada, 16–24 May 2009; IEEE: Piscataway, NJ, USA, 2009; pp. 364–374.
2. Goues, C.L.; Nguyen, T.V.; Forrest, S.; Weimer, W. GenProg: A Generic Method for Automatic Software Repair. *IEEE Trans. Softw. Eng.* **2012**, *38*, 54–72. [CrossRef]
3. Goues, C.L.; Dewey-Vogt, M.; Forrest, S.; Weimer, W. 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, Zurich, Switzerland, 2–9 June 2012; IEEE: Piscataway, NJ, USA, 2012; pp. 3–13.
4. Oliveira, V.P.L.; de Souza, E.F.; Goues, C.L.; Camilo-Junior, C.G. Improved representation and genetic operators for linear genetic programming for automated program repair. *Springer Empir. Softw. Eng.* **2018**, *23*, 2980–3006. [CrossRef]
5. Yuan, Y.; Banzhaf, W. ARJA: Automated repair of java programs via multi-objective genetic programming. *IEEE Trans. Softw. Eng.* **2018**, *46*, 1040–1067. [CrossRef]
6. Yuan, Y.; Banzhaf, W. A hybrid evolutionary system for automatic software repair. In Proceedings of the ACM Genetic and Evolutionary Computation Conference, Prague, Czech Republic, 13–17 July 2019; pp. 1417–1425.
7. Yuan, Y.; Banzhaf, W. Toward better evolutionary program repair: An integrated approach. *ACM Trans. Softw. Eng. Methodol.* **2020**, *29*, 1–53. [CrossRef]
8. Le, X.B.D.; Lo, D.; Goues, C.L. History driven program repair. In Proceedings of the 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, Suita, Japan, 14–18 March 2016; Volume 1, pp. 213–224.
9. Just, R.; Jalali, D.; Ernst, M.D. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In Proceedings of the ACM 2014 International Symposium on Software Testing and Analysis (ISSTA), San Jose, CA, USA, 21–25 July 2014; pp. 437–440.
10. Jiang, J.J.; Xiong, Y.F.; Zhang, H.Y.; Gao, Q.; Chen, X.Q. Shaping program repair space with existing patches and similar code. In Proceedings of the ACM 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, Amsterdam, The Netherlands, 16–21 July 2018; pp. 298–309.
11. Martinez, M.; Weimer, W.; Monperrus, M. Do the fix ingredients already exist? an empirical inquiry into the redundancy assumptions of program repair approaches. In Proceedings of the IEEE 36th International Conference on Software Engineering, Hyderabad, India, 31 May–7 June 2014; pp. 492–495.
12. Kim, J.; Kim, J.; Lee, E.; Kim, S. *The Effectiveness of Context-Based Change Application on Automatic Program Repair*; Springer Empirical Software Engineering: Berlin/Heidelberg, Germany, 2020; pp. 719–754.
13. Barr, E.T.; Brun, Y.; Devanbu, P.; Harman, M.; Sarro, F. The plastic surgery hypothesis. In Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, Hong Kong, China, 16–21 November 2014; pp. 306–317.
14. Qi, Y.; Mao, X.; Lei, Y.; Dai, Z.; Wang, C. The strength of random search on automated program repair. In Proceedings of the IEEE 36th International Conference on Software Engineering, Hyderabad, India, 31 May–7 June 2014; pp. 254–265.
15. Wen, M.; Chen, J.; Wu, R.; Hao, D.; Cheung, S.C. Context-aware patch generation for better automated program repair. In Proceedings of the 40th International Conference on Software Engineering, Gothenburg, Sweden, 27 May–3 June 2018; pp. 1–11.
16. Sidiroglou-Douskos, S.; Lahtinen, E.; Long, F.; Rinard, M. Automatic Error Elimination by Horizontal Code Transfer Across Multiple Applications. In Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, 13–17 June 2015; pp. 43–54.
17. Le Goues, C.; Forrest, S.; Weimer, W. Current challenges in automatic software repair. *Softw. Qual. J.* **2013**, *21*, 421–443. [CrossRef]
18. White, M.; Tufano, M.; Martinez, M.; Monperrus, M.; Poshyvanyk, D. Sorting and transforming program repair ingredients via deep learning code similarities. In Proceedings of the IEEE 26th International Conference on Software Analysis, Evolution and Reengineering, Hangzhou, China, 24–27 February 2019; pp. 479–490.

19. Yokoyama, H.; Higo, Y.; Hotta, K.; Ohta, T.; Okano, K.; Kusumoto, S. Toward improving ability to repair bugs automatically: A patch candidate location mechanism using code similarity. In Proceedings of the 31st Annual ACM Symposium on Applied Computing, Pisa, Italy, 4–8 April 2016; pp. 1364–1370.
20. Madeiral, F.; Urli, S.; Maia, M.; Monperrus, M. Bears: An extensible java bug benchmark for automatic program repair studies. In Proceedings of the IEEE 26th International Conference on Software Analysis, Evolution and Reengineering, Hangzhou, China, 24–27 February 2019; pp. 468–478.
21. Soto, M.; Le Goues, C. Using a probabilistic model to predict bug fixes. In Proceedings of the IEEE 25th International Conference on Software Analysis, Evolution and Reengineering, Campobasso, Italy, 20–23 March 2018; pp. 221–231.
22. Martinez, M.; Durieux, T.; Sommerard, R.; Xuan, J.; Monperrus, M. Automatic Repair of Real Bugs in Java: A Large-Scale Experiment on the Defects4J Dataset. *Empir. Softw. Eng.* **2017**, *22*, 1936–1964. [CrossRef]
23. Xuan, J.F.; Martinez, M.; Demarco, F.; Clement, M.; Marcote, S.L.; Durieux, T.; Le Berre, D.; Monperrus, M. Nopol: Automatic repair of conditional statement bugs in java programs. *Trans. Softw. Eng.* **2016**, *43*, 34–55. [CrossRef]
24. Qi, Z.; Long, F.; Achour, S.; Rinard, M. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In Proceedings of the 2015 International Symposium on Software Testing and Analysis, Baltimore, MD, USA, 13–17 July 2015; pp. 24–36.
25. Mehne, B.; Yoshida, H.; Prasad, M.R.; Sen, K.; Gopinath, D.; Khurshid, S. Accelerating search-based program repair. In Proceedings of the 2018 IEEE 11th International Conference on Software Testing, Verification and Validation, Västerås, Sweden, 9–13 April 2018; pp. 227–238.
26. Sun, S.Y.; Guo, J.X.; Zhao, R.L.; Li, Z. Search-based efficient automated program repair using mutation and fault localization. In Proceedings of the 2018 IEEE 42nd Annual Computer Software and Applications Conference, Tokyo, Japan, 23–27 July 2018; pp. 174–183.
27. Li, D.; Wong, W.E.; Jian, M.; Geng, Y.; Chau, M. Improving search-based automatic program repair with Neural Machine Translation. *IEEE Access* **2022**, *10*, 51167–51175. [CrossRef]
28. Villanueva, O.M.; Trujillo, L.; Hernandez, D.E. Novelty search for automatic bug repair. In Proceedings of the 2020 ACM Genetic and Evolutionary Computation Conference, Cancún, Mexico, 8–12 July 2020; pp. 1021–1028.
29. Ji, T.; Chen, L.Q.; Mao, X.G.; Yi, X. Automated program repair by using similar code containing fix ingredients. In Proceedings of the 2016 IEEE 40th Annual Computer Software and Applications Conference, Atlanta, GA, USA, 10–14 June 2016; pp. 197–202.
30. Fluri, B.; Wursch, M.; PInzger, M.; Gall, H. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Trans. Softw. Eng.* **2007**, *33*, 725–743. [CrossRef]
31. Lin, B.; Wang, S.; Wen, M.; Mao, X. Context-aware code change embedding for better patch correctness assessment. *ACM Trans. Softw. Eng. Methodol.* **2022**, *31*, 1–29. [CrossRef]
32. Xin, Q.; Reiss, S.P. Leveraging syntax-related code for automated program repair. In Proceedings of the 2017 32nd International Conference on Automated Software Engineering, Urbana-Champaign, IL, USA, 30 October–3 November 2017; pp. 660–670.
33. Motwani, M.; Soto, M.; Brun, Y.; Just, R.; Goues, C.L. Quality of Automated Program Repair on Real-World Defects. *IEEE Trans. Softw. Eng.* **2022**, *48*, 637–661. [CrossRef]
34. Phung, Q.N.; Kim, M.; Lee, E. Identifying Incorrect Patches in Program Repair Based on Meaning of Source Code. *IEEE Access* **2022**, *10*, 12012–12030. [CrossRef]