ITC 4/51

Information Technology and Control
Vol. 51 / No. 4 / 2022
pp. 738-756
DOI 10.5755/j01.itc.51.4.30515

Automatic Repair of Java Programs Weighted Fusion Similarity via Genetic Programming

Received 2022/01/16 | Accepted after revision 2022/05/26

http://dx.doi.org/10.5755/j01.itc.51.4.30515

HOW TO CITE: Cao, H., He, Z., Meng, Y., Chu, Y. (2022). Automatic Repair of Java Programs Weighted Fusion Similarity via Genetic Programming. *Information Technology and Control*, 51(4), 738-756. http://dx.doi.org/10.5755/j01.itc.51.4.30515

Automatic Repair of Java Programs Weighted Fusion Similarity via Genetic Programming

Heling Cao

College of Information Science and Engineering, Henan International Joint Laboratory of Grain Information Processing; Henan University of Technology; zhengzhou 450001; China; e-mails: caohl@haut.edu.cn

Zhenghao He, Yangxia Meng, Yonghe Chu

College of Information Science and Engineering; Henan University of Technology; zhengzhou 450001; China; e-mails: hezhenghao24@163.com, myx_ziqiboya@163.com, yonghechu@163.com

Corresponding author: yonghechu@163.com

Recently, automated program repair techniques have been proven to be useful in the process of software development. However, how to reduce the large search space and the random of ingredient selection is still a challenging problem. In this paper, we propose a repair approach for buggy program based on weighted fusion similarity and genetic programming. Firstly, the list of modification points is generated by selecting modification points from the suspicious statements. Secondly, the buggy repair ingredient is selected according to the value of the weighted fusion similarity, and the repair ingredient is applied to the corresponding modification points according to the selected operator. Finally, we use the test case execution information to prioritize the test cases to improve individual verification efficiency. We have implemented our approach as a tool called WSGRepair. We evaluate WSGRepair in Defects4J and compare with other program repair techniques. Experimental results show that our approach improve the success rate of buggy program repair by 28.6%, 64%, 29%, 64% and 112% compared with the GenProg, CapGen, SimFix, jKali and jMutRepair.

KEYWORDS: automated program repair, code similarity, genetic programming, test case prioritization.

1. Introduction

Automated program repair is designed to reduce the effort of repair bugs by automatically generating patches. One popular automatic patch generation aims to generate and verify candidate patches until a patch passed all the given test cases. The patch generation of automated program repair techniques is often described as an exploration of their patch search space. The problem is challenging because the space is usually huge, and contains a lot of plausible patches (incorrect patches still pass the test case). Test cases cannot distinguish between correct and plausible but incorrect patches. The automated program repair approach requires not only locating the correct patches from a huge space, but also avoiding patches that seem reasonable but are incorrect. Since 2009, automatic program repair had become a popular research area, attracted researchers from a wide range of communities including software engineering, artificial intelligence and formal verification. Le Goues et al. [11, 13, 26] was the first to apply genetic programming to automatic program repair. GenProg [13] uses genetic algorithms to realize the recombination of existing code fragments by defining the crossover and mutation operations of code fragments. Ji et al. [5] proposed to generate patches by reusing code fragments in the project that are similar to the buggy location. Jiang et al. [7] proposed an automatic repair method SimFix. SimFix extracts the abstract search space S1 from the existing patches, extracts the similar code from the bug source programs to form the search space S2, and obtains the patches from the intersection of these two search spaces. Wen et al. [27] proposed an automatic repair method CapGen. CapGen extracts the contextual environment information of the abstract syntax tree nodes at a fine-grained level (e.g. expressions), and selects repair operators and repair ingredients for the buggy program based on the contextual environment information.

The above automatic program repair approaches have proven useful. However, some obvious problems were found in the search space and the random of ingredient selection. Automatic patch generation is often described as a search problem for the patch candidate space, which has two main issues: one is the size of the search space, and the other is navigation. The existing automatic patch generation techniques search too large a space and the random nature of repair ingredient selection makes it possible for these techniques to produce meaningless patches. Test cases cannot distinguish between correct and plausible patches. Automatic program repair methods need not only to locate correct patches from a large space, but also to avoid plausible patches.

In order to reduce the search space, the candidate patch search efficiency is accelerated and the problem of too random ingredient selection is solved. In this paper, we propose a repair method combining weighted fusion similarity and genetic programming. Firstly, fault localization techniques are used to locate the buggy statements. Secondly, we use weighted fusion similarity and genetic programming to select the repair ingredient. Finally, test case prioritization techniques are used to speed up patch generation rate. We have implemented our approach as an automated program repair tool called WSGRepair, and evaluated it on the Defects4J benchmark [8].

The main contributions of this paper can be summarized as follows:

- An automated program repair approach is based on weighted fusion similarity and genetic programming.
- A new fitness function is designed to better guide the individual evolutionary process.
- We conduct experimental study on 224 real world bugs in Defects4J to show the effectiveness of our approach.

The remainder of this paper is organized as follows. Section 2 provides the background of based repair. Section 3 presents our approach and its detailed description. Section 4 illustrates the experimental study and its data analysis. Section 5 discusses the threats to validity. Section 6 presents related work and Section 7 draws conclusions and discusses some potential future work.

2. Background

2.1. Automatic Program Repair

Automatic program repair can successfully complete the automatic repair of some of these bugs, thus it effectively reducing debugging time for developer program. Therefore, automatic program repair has gradually become a hot research topic in the field of software maintenance and has made some progress in research. Different kinds of techniques can be used, such as genetic programming search in GenProg [13] and SMT based program synthesis in SemFix [19]. DirectFix [18] has optimized the patch generation process compared to SemFix. DirectFix does not strictly follow the locate results, but rather prioritizes the simpler candidate statements for repair. Weimer et al. [25] propose an adaptive repair method based on program equivalence. One of the mainstream approaches is the test case-based automatic program repair approach, which evaluates the quality of the generated patches through a set of test cases. This approach is subdivided into three phases: fault localization phase, patch generation phase and patch verification phase. The fault localization phase is the basis of automatic program repair, and its goal is to identify as accurately as possible the statements that may contain bugs. The patch generation phase generally modifies the buggy statements through predefined modification operations. The code modification operations can be considered when setting up the repair program own code, generation of open source projects etc. The patch evaluation phase evaluates the generated candidate patches until a patch is found that allows all test cases to pass, and then finalizes it with the help of manual analysis by the developers.

Prophet [16] learns a patch ranking model using machine learning algorithm based on existing patches. Saha et al. [23] used a Prophet-like approach to guide the repair process through predefined patch templates while applying a machine learning model to rank the candidate repair patches. Xiong et al. [30] proposed a high-precision conditional statement synthesis technique for conditional statement repair in Java language. Long et al. [15] designed the automatic repair technique Genesis for null pointers (NPE), array out-of-bounds (OOB), and strong type conversion (CC) defects in the Java language. Although the existing search-based APR techniques are effective, they still have limitations in terms of poor bug program repair success rates and long bug program repair times due to an overly simplistic approach to the selection of bug program repair ingredient. For example, incorrect patches are generated before the correct patch or budget timeout due to the generation of reasonable patches.

2.2. Genetic Programming

Genetic programming [10] is derived from genetic algorithms and is a different manifestation of genetic algorithms, which are essentially extensions of genetic algorithms to programming problems. The main process of genetic programming is basically the same as that of genetic. The main processes of genetic programming are basically the same as those of genetic algorithms, but there are significant differences in the individual manifestations. Individuals in genetic programming are generally represented as a program in the form of an abstract syntax tree, rather than a binary bit string as in a genetic algorithm. Each node in the abstract syntax tree each node in the abstract syntax tree corresponds to a statement or control flow structure in the program, and crossovers and mutations are performed on the corresponding nodes.

2.3. Code Similarity

Many existing techniques dedicate to the identification of the differences between two code snippets and the generation of the transformations from one to the other. Levenshtein distance is a typical representation of edit distance that can be used to calculate the similarity calculation between two strings and is commonly used in natural language processing to calculate the similarity between two texts. The algorithm represents the minimum number of operations required to transform two strings into each other by inserting, deleting and replacing a character in the string. The smaller the edit distance between the strings, the higher the similarity between them. The larger the edit distance, the lower the similarity value between them. Because when solving the minimum edit distance between two strings, the problem has the overlapping nature of sub-problems and the characteristics of optimal sub-structure. In line with the basic idea of dynamic programming, this method adopts the dynamic programming algorithm of edit distance to solve the problem.

The Dice coefficient is calculated as the ratio of 2 times the intersection of the two sets to the sum of the two sets. The Dice coefficient and its derivatives can also be used in the field of object detection and image segmentation. The range of the Dice coefficient is 0 to 1. The closer the value is to 1, the closer the two are, and the better the model effect is.

$$SimDice = \frac{2|X \cap Y|}{|X|+|Y|}. \quad (1)$$

In Equation (1), $\cap$ represents the intersection of set X and set Y, $|X \cap Y|$ represents the number of identical elements of sets X and Y, $|X|$ and $|Y|$ represents the size of sets X and Y.

3. Approach

We propose weighted fusion similarity and genetic programming. Firstly, the fault localization technique is used to locate the suspicious statements and select the eligible suspicious statements as the modification point. Secondly, repair ingredient is selected based on weighted fusion similarity values, and evolutionary repair of the buggy program is performed based on genetic programming. Finally, prioritize test cases according to test case execution information to improve individual verification efficiency. The above steps are repeated until either a program individual passes all test cases or the program iteration termination condition is met.

Figure 1 shows an overview of our approach. This method divides the whole process into three phases: fault localization, patch generation and patch verification.

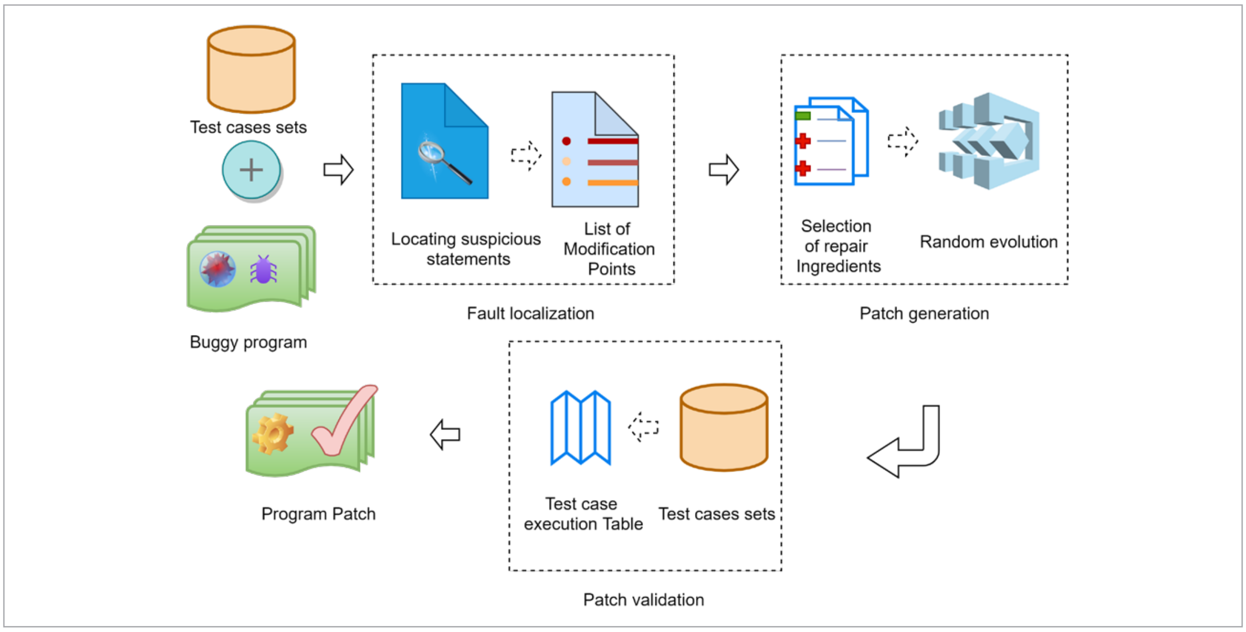
1 Fault Localization. This method uses the fault localization technique Ochiai to localization and analyze the bug source program, and its suspicious value is calculated as described in Equation (2).

$$Ochiai(s) = \frac{n_{ef}(s)}{\sqrt{n_f \times (n_{ef}(s) + n_{ep}(s))}}. \quad (2)$$

In Equation (2), s represents the program entity, $n_{ep}(s)$ and $n_{ef}(s)$ represent the number of successful test cases and the number of failed test cases covering the program entity s, respectively, and n_f represents the number of all failed test cases of the program entity s.

2 Patch generation. Firstly, the program evolution repair process is targeted to select repair ingredients, and the repair ingredients are selected according to the weighted fusion in the ingredient selection. Secondly, the operators required for the

Figure 1
The overall workflow of the proposed approach



buggy program evolution are further subdivided into four operators: insert before statement, insert after statement, modify and delete. Finally, the adaptation degree function equation is used to calculate the population in accordance with the fitness of an individual, which is an important condition to constrain the evolution of an individual. The calculation is shown in Equation (3). According to the mutation, crossover, selection and other operations in genetic programming for the evolution of bug programs, the lower the fitness value, the higher the probability that an individual is selected to enter the next generation of evolution. The above program evolution process is repeated until a patch can meet the program statute requirements or reach the program iteration termination condition.

$$fitness(i)=\frac{exeNum}{totalNum*tatalSim}. \tag{3}$$

In Equation (3), $fitness(i)$ represents the fitness value of the ith sub-individual, $exeNum$ represents the total number of test cases executed when the individual finishes verification, $totalNum$ represents the total number of test cases matched with the bug program, $totalSim$ represents the modification point in the individual and the size of the total similarity between materials.

3 Patch verification. After the candidate patches are generated by the program evolution, the test cases need to be run repeatedly to verify the validity of the candidate patches. In this method, in order to improve the efficiency of patch verification, a test case prioritization technique based on error recognition capability is adopted.

3.1. Fault Localization

This method uses the fault localization technique Ochiai to localization and analyze the bug source program, and the Ochiai suspicious value is calculated as described in Equation (2). A series of suspicious statements are obtained after fault localization, and according to the minimum suspicious value and maximum number of modification points set by the program. The modification statements are selected to generate a list of modification points, and the list of modification points is the suspicious space when the individual evolves. The modification points are selected according to the size of the suspicious value weight of the suspicion statements, and their weights are calculated as shown in Equation (4).

$$W(i)=\frac{S_i}{\sum_{j=1}^{n} S_j}, \tag{4}$$

where *i* denotes the location information of the suspicious statements, *Si* and S_j denote the suspicious value of the suspicious statements, and *W(i)* denotes the weight value of the suspicious statements. The magnitude of the weight value of the suspicious utterance is used as a constraint for selecting the modification point when selecting the modification point.

3.2. Patch Generation

Weighted fusion similarity focuses on ingredient selection using code similarity, and repair of buggy program using code similarity is mainly based on two theories: redundancy theory and plastic surgery conjecture. These two theories coincide in stating that there may be correct patches in the buggy program that can repair the buggy program. Based on these two theories, we adopt a weighted fusion similarity calculation to select program statements that are similar to the buggy statements as the ingredient needed for repair in order to make the ingredient selection method more appropriate and the program repair more efficient.

However, in this paper, the weighted fusion similarity mainly refers to combination of the similarity of the textual features and that of the contextual environment features at the modification points. The contextual environment features include the characteristics of the variables, the characteristics of the names, return value types, and parameters of the methods containing in the ingredient and modification points. After obtaining the text feature similarity and the context feature of the modification point, a weighted fusion is obtained, which is an important constraint for ingredient selection to guide the ingredient selection process.

When calculating the similarity of text features between program segments, a strict distinction is made between system identifier similarity and user-defined identifier similarity. The system identifier similarity and user-defined similarity are calculated separately from the Levenshtein distance. The text feature similarity between program segments is the weighted sum of the system identifier similarity and the user-defined identifier similarity, as shown in Equation (5).

$$simEdit = s_1 * simSystem + s_2 * simUser, \quad (5)$$

where *simEdit* denotes the text feature similarity, s_1 denotes the weight of the system identifier similarity, s_2 denotes the user-defined identifier similarity weight. The *simSystem* denotes the calculated system identifier similarity, and *simUser* denotes the calculated user-defined identifier similarity. In this Equation, the weight coefficient satisfies $s_1 + s_2 = 1$.

To obtain the contextual environment features of the modification point and the repair ingredient, it is necessary to extract the method features of the method in which the repair ingredient and the modification point are located, as well as the variable features contained in the modification point statement and the repair ingredient. The method features contain the return value type of the method, the name of the method, and the parameters passed in the method. The variable features contain the type of the variable and the name of the variable. After obtaining the contextual environment features, the method features and variable features are put into the set, and then the similarity of method features and the similarity of variable features between the ingredient and the modification point are calculated separately using the Dice similarity coefficient.

Assuming that the similarity feature of a material and the program text of the modification point is $\boldsymbol{sim_1}$, the feature attribute is $\boldsymbol{sim_2}$, and the variable attribute feature is $\boldsymbol{sim_3}$, the similarity between the material and the modification point is shown in Equation (6).

$$totalSim = w_1 * sim_1 + w_2 * sim_2 + w_3 * sim_3, \quad (6)$$

where *totalSim* is the total similarity value, sim_1 represents the similarity of program text features between ingredients and modification points, sim_2 represents the similarity of method features between ingredients and modification points, sim_3 represents the similarity of variable features between ingredients and modification points, and w_1, w_2 and w_3 are the weight coefficients of text features, method features, and variable features respectively. Among them, w_1, w_2 and w_3 satisfy $w_1+w_2+w_3=1$.

After obtaining the similarity values of ingredients and modification points, the total similarity value is set to the ingredients and the ingredient space is reconstructed according to the total similarity value size. When the repair ingredient is selected for the modification point, it is preferred according to the similarity weight size of the repair ingredient.

3.3. Patch Generation

When repair a buggy program, the large number of candidate patches may have to be verified and the test case set executed multiple times before the correct patch for the buggy program is found. In the previous program automatic repair, the execution of test cases was random, so a situation arose where some test cases in the test case set might be executed multiple times over. Random execution of test cases is not only inefficient in validation, but also causes waste of computing resources and prolongs test case execution time.

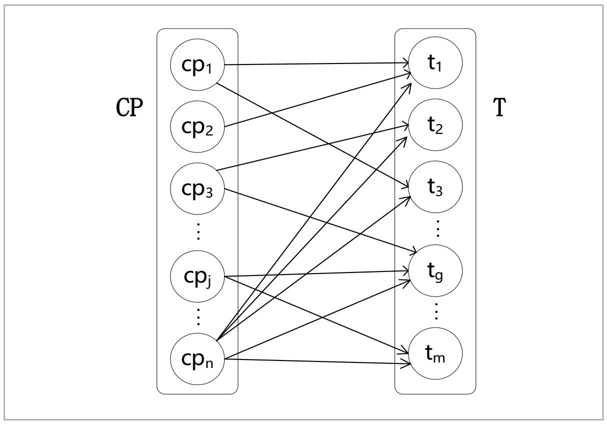
To address the inefficiency of the long execution time of test cases in existing repair techniques. We use a test case prioritization technique to prioritize test cases by recording the execution information of test cases at the time of individual validation. The candidate patch and test case mapping relationship is shown in Figure 2, where CP is the set of candidate patches generated during individual validation, T is the set of test cases, and cp_n is the correct patch that passes all test cases. During the validation process, the following two phenomena were observed.

1. There is a partial overlap of test cases executed in the n-1 candidate patches before the correct patch cp_n is found.
2. Since individual validation ends the validation process for that individual when a test case validation failure is encountered, there is a case of partial execution of test cases in the test case set at this time.

Given the above two phenomena, in Figure 2, the directed line segment indicates the test cases that the patch has been executed, and patch cp_n is the correct patch that passed all test cases. From Figure 2, it can be seen that the number of times each test case is executed before the correct patch is found is dif-

Figure 2

Mapping between patch candidates and test cases



ferent, reflecting the different bug recognition capabilities of the test cases. Our uses test case prioritization techniques based on bug recognition capability to rank test cases in descending order according to their number of executions and priority the test cases with the highest number of executions during verification. For each execution of a test case, the marker recording the number of executions of that test case is added by one, and test cases with the same number of executions are randomly selected for verification. After each candidate patch is verified, the test cases are sorted in descending order according to the number of times they were executed. If a candidate patch passes the validation of all test cases, the candidate patch is considered to be a valid patch.

3.4. WSGRepair

Algorithm 1 depicts the overall flow of the system. According to Algorithm 1, the buggy program repair tool WSGRepair is implemented in Java programming language by combining weighted fusion similarity and genetic programming. Given a bug source program P and its corresponding test case set T. In (lines 1-2) of this algorithm, a sequence of suspicious statements is located using the fault localization tool to derive a sequence of suspicious statements, and according to the minimum suspicious value *minSus* set by the program and the maximum modification. Lines 3 to 4 indicate the initialization operation of the population and the setting of the fitness function of the initial population, in which each individual has the same value of the fitness function. Lines 6 to 19 of the algorithm indicate the specific process of individual

Algorithm 1: WSGRepair repair algorithm

```
Input: P              // buggy program
T                     // Test case sets
OperatorSpace         // Operator space
popSize               // Population size
IngredientSpace       // Ingredient Space
Output: cp  // Valid patches through all test cases
begin
1  SusList ← FaultLocalization(T,P)
2  ModList ← GetSusSpace(SusList,minSus,maxMod)
3  Pop ← InitPopulation(popSize)
4  InitFitness(Pop)
5  for i ← 1 to maxMut do
6    parent ← URSelect(Pop,Fitness)
7    modPoint ← WRSelect(ModList)
8    Op ← URSelect(OperatorSpace)
9  if canApplyOp(Op,modPoint) then
10     if opNeedIngredients(Op) then
11         newIngredientSpace←resetIngSpace
           (IngredientSpace,modPoint)
12         fixIngredient←WRSelect(newIngredientSpace)
13         child←generateNewVariant(parent,modPoint,
           Op, fixIngredient)
14     else
15         Child ← generateNewVariant(parent,mod-
           Point,Op)
16         Offsprings add child
17 end for
18 for i ← 1 to maxCrossover do
19     parent ← URSelect(Pop)
20     child ← URSelect(Offsprings)
21     modPoint ← selectModLine(child)
22     NewChild ← Crossover(parent,child,modPoint)
23     Offsprings add Newchild
24 end for
25 exeNum ← Validation(T,Offsprings)
26 T' ← testCasePariority(T)
27 Similarity ← totalSim(child)
28 Fitness ← fitFunction(exeNum, Similarity, Off-
springs)
29 Pop←WRSelect(PopSize,Offsprings,parents,Fitness)
30 until    ∈ Offsprings passed all the testcases
31 return cp
end
```

variation in the population. Lines 20 to 26 of the algorithm describe the specific implementation of the crossover of individuals in the population. Lines 27 to 31 represent the specific process of individual validation and selection. Lines 6 to 31 of the algorithm are repeated until an individual meets the statute requirement of passing the entire test case set or reaches the maximum loop termination condition set by the repair program.

4 Experimental Study

4.1. Experimental Setup and Design

We implement the tool WSGRepair in Java with the code parsing tool Spoon [20] and the fault localization tool GZoltar [22], running on Ubuntu 18.04 LTS, 2.40 GHz Intel(R) CPU, and 8G of running memory.

Referring to the experimental setup of existing repair techniques, the maximum running time of the program set by this method is 120 minutes. In the fault localization phase, the minimum suspicious threshold set is 0.5, the maximum number of modification points set is 50, the maximum number of iterations of the program and the number of individuals per generation of the population is 10. The maximum number of variants is 1.5 times of the number of individuals per generation of the population, and the crossover probability is 0.2 times of the number of individuals per generation of the population. Therefore, at most 10×1.5 + 10×0.2×2 = 19 individuals are generated per generation, and the whole repair process generates at most 19×10 = 190 candidate patches. For the similarity weight coefficients in WSGRepair, s_1 and s_2 are 1/3 and 2/3 in Equation (5), respectively. The similarity weights w_1, w_2 and w_3 are taken as 1/3 each in Equation (6).

To apply to the real world, automatic repair methods need execute fast. This runtime is acceptable to us. When automated repair methods can synthesize a patch suitable for testing, it is often found within a few minutes. During the 20.6 days of our experiment, this meant that we spent a lot of time on unfixed bugs which were not fixed due to timeouts. We found that increasing the runtime of each program not only can't improve the repair efficiency but also wastes a plenty of time. Therefore, we choose the most running time of 120 minutes.

When the number of iterations is too large, a large number of candidate patches will be generated, and it will take a long time to find the correct patch in the search space. A patch that passes the test case can be found in a short time in the search space. Too many iterations will lead to too long search time, and expanding the number of candidate patches often does not increase the number of correct patches. Too many iterations will lead to too long search time, and expanding the number of candidate patches often does not increase the number of correct patches. Therefore, 10 generations are the most suitable for our repeated experiments, because it is acceptable for us to do a complete experiment in 20.6 days.

4.2. Research Questions

Our evaluation aims to answer the following research questions:

RQ1: How performance does WSGRepair fix real world bugs?

In order to evaluate the performance of WSGRepair on Defects4J, we mainly verify the repair effect of the defect repair method from the following three aspects: (1) the number of successful repairs of the bug program; (2) the number of candidate patches generated when the bug program is successfully repaired Number of NCP (Number of Candidate Patches); (3) The times for the buggy program to be successfully repaired.

RQ2: Can WSGRepair outperform the compared approaches?

To investigate the difference in repair performance between WSGRepair and other types of repair methods, we selected popular methods for comparison, WSGRepair achieves better performance than the compared approaches.

RQ3: What repair actions does WSGRepair use to fix bugs?

The repair actions of WSGRepair are deeply studied in order to generate program patches, and the specific defect types repaired by WSGRepair can be judged through the repair actions.

4.3. Defects4J Dataset

Researchers have integrated bug database of different sizes for different programming languages. In order to test the effectiveness of the proposed repair technique, the Defects4J, a mature set of bug database for large real projects, was selected as the experimental dataset after comparing the different features among the bug database. Real program buggy are provided in the Defects4J test set, which is a scalable bug database. Defects4J by Just et al. [8] is a bug database that consists of 357 real world bugs from five widely-used open-source Java projects. In addition, each program has a comprehensive test suite, and each bug can be reproduced with accompanying test cases. At least one of these provided test cases is capable of triggering a bug. Currently, Defects4J is respected by a wide range of researchers in software engineering, and is widely used in the fields of fault localization and bug repair.

The version of Defects4J selected for this experiment is 1.4.0, which collects one more Mockito project than the initial version of the dataset, which consists of 38 bug programs. Since the test cases of two projects, Closure Compiler and Mockito, do not conform to the standard JUnit organization, they are excluded from this experiment. This, the experimental projects in this experiment are JFreeChart, Commons Math, Joda-Time, and Commons Lang, with a total of 224 bugs. Table 1 gives the details of the four bug items in the Defects4J test set used.

Table 1
Details of the experiment benchmark

Subject	Project	Bugs	KLOC	Test Cases
JFreeChart	Chart	26	96	2,205
Commons Math	Math	106	85	3,602
Joda-Time	Time	27	28	4,130
Commons Lang	Lang	65	22	2,245
Total	-	224	231	12182

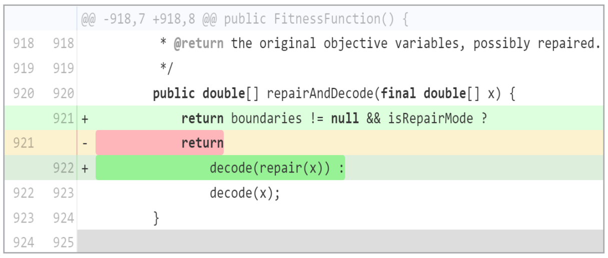
4.4. Experimental Evaluation

RQ1: What performance does WSGRepair do to fix real world bugs?

Referring to existing repair techniques [7, 27], the repair effectiveness of the repair method was verified in three main aspects: (1) The number of successful repairs of the buggy program. (2) The Number of Candidate Patches (NCP) generated when the buggy program was successfully repaired. (3) The times for the buggy program to be successfully repaired. In order to further demonstrate the repair effectiveness of this repair method at multiple levels, the classic repair tool GenProg was selected as the comparison experiment for this experiment to reproduce the repair process of GenProg and record the repair information during the repair process for comparison.

This section uses a real-world example from our experiments. Figure 3 shows a bug in the Defects4J benchmark, Math20. Note that in this article, a line of code that begins with "+" indicates a newly added correct code, and a line that begins with "-" indicates a buggy code to be deleted.

Figure 3
Fragment program with the buggy and the corresponding correct code in WSGRepair



```
          @@ -918,7 +918,8 @@ public FitnessFunction() {
918  918              * @return the original objective variables, possibly repaired.
919  919              */
920  920             public double[] repairAndDecode(final double[] x) {
     921  +              return boundaries != null && isRepairMode ?
921       -              return
     922  +                  decode(repair(x)) :
922  923                      decode(x);
923  924             }
924  925
```

Figure 3 shows an example of a bug caused by the return statement on line 921. To fix it, the developer modified the original return expression and added a return statement. This motivates us to search for similar code through code structure features as well as code context features, extract repair components, and use them for mutation operators, such as expression level. At the same time, GenProg does not fix this bug because the GenProg search space is too large. GenProg mutation operators are all random, and randomness will cause the search space of the program to be too large. However, enlargement of the search space does not necessarily improve the probability of correct patches. Therefore, we use the code similarity to greatly reduce the search space and improve the repair success rate.

Table 2 shows the comparison between WSGRepair and GenProg on the four major items of Chart, Math, Lang, and Time. GenProg can successfully repair 28 bugs in the Defects4J dataset, with a success rate of

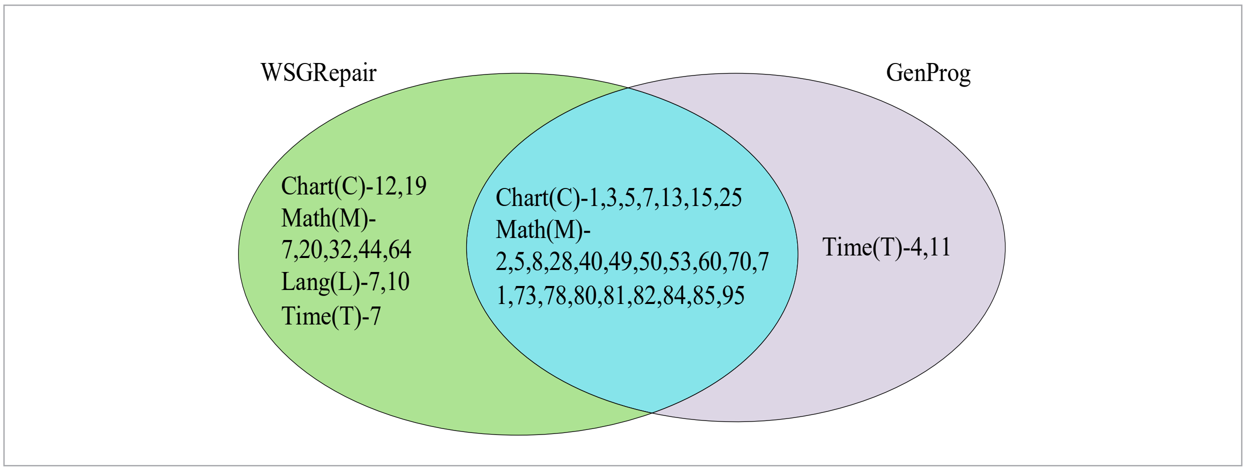
12.50%, while WSGRepair can successfully repair 36 bugs, with a success rate of 16.07%. Compared with GenProg, the repair accuracy of WSGRepair is improved by 28.6%, which is mainly because the weighted fusion similarity measure developed in WSGRepair can select repair ingredients more accurately. Thereby speeding up the program patch search process, improve the program repair efficiency. In contrast, there is no corresponding ingredient selection strategy in GenProg, which fails to better select bug repair ingredients, resulting in its failure to generate effective patches for bug programs within the specified repair time or number of program evolutions.

Figure 4 shows the comparison Venn diagram of bug program repair of WSGRepair and GenProg. From Figure 4, we can visually see that on the Chart project, WSGRepair can repair two bug programs C12 and C19 that GenProg failed to repair. On the Math project, WSGRepair can repair M7, M20, M32, M44, and M64 that GenProg failed to repair, totaling five

Table 2
Comparison with GenProg on Defects4J

Project	bugs	WSGRepair	GenProg
Chart	26	C1,C3,C5,C7,C12,C13,C15,C19,C25	C1,C3,C5,C7,C13,C15,C25
		å=9	å=7
Math	106	M2,M5,M7,M8,M20,M28,M32,M40,M44,M49, M50,M53,M60,M64,M70,M71,M73,M78,M80, M81,M82,M84,M85,M95	M2,M5, M8,M28,M40,M49,M50,M53, M60,M70,M71,M73,M78,M80,M81,M82, M84,M85,M95
		å=24	å=19
Lang	65	L7,L10	-
		å=2	å=0
Time	27	T7	T4,T11
		å=1	å=2
Total	224	36	28
Precision	-	16.07%	12.50%

Figure 4
Venn diagram comparing WSGRepair and GenProg restoration



bug programs. On the Lang project, WSGRepair can repair two bug programs L7 and L10 that GenProg failed to repair. On the Time project, WSGRepair can repair the bug program T7, while GenProg can repair two bug programs T4 and T11.

Table 3 collects specific verification times, run times, and NCP details for the methods WSGRepair and GenProg on the 26 bug programs that can be repaired by both. For the specific runs, the data collected for each buggy program are averaged over 10 runs in order to reduce the effect of chance.

As seen in Table 3, the minimum time taken by GenProg to successfully repair a buggy program was 15.8 minutes and the maximum time was 118.4 minutes, while the minimum time taken by WSGRepair to successfully repair a buggy program was 5.9 minutes and the maximum time was 63.4 minutes. The total time taken by GenProg to repair these 26 buggy programs

Table 3
Detailed performance comparison with GenProg

Project	Bug ID	Approach	NCP	Validation time (min)	Total time (min)
Chart	C1	GenProg	20	15.6	23.2
		WSGRepair	11	7.3	14.3
	C3	GenProg	58	17.3	32.3
		WSGRepair	50	11.3	20.7
	C5	GenProg	40	10.4	19.2
		WSGRepair	9	2.2	10.7
	C7	GenProg	38	10.7	15.8
		WSGRepair	21	2.1	5.9
	C13	GenProg	34	23.8	37.8
		WSGRepair	15	9.1	15.8
	C15	GenProg	46	11.3	25.3
		WSGRepair	28	1.25	9.4
	C25	GenProg	35	18.5	36.7
		WSGRepair	16	8.2	23.5
Math	M2	GenProg	42	45.6	63.4
		WSGRepair	12	11.4	23.4
	M5	GenProg	63	58.2	68.3
		WSGRepair	19	12.7	23.7
	M8	GenProg	25	44.8	80.6
		WSGRepair	14	11.2	22.8
	M28	GenProg	53	66.2	89.1
		WSGRepair	24	12.3	20.7
	M40	GenProg	54	78.3	89.4
		WSGRepair	40	31.5	63.4
	M49	GenProg	43	90.2	100.4
		WSGRepair	22	37.1	50.8
	M50	GenProg	20	21.5	43.6
		WSGRepair	19	13.7	21.3
	M53	GenProg	64	77.2	117.3
		WSGRepair	41	11.9	31.8
	M60	GenProg	42	37.4	50.9
		WSGRepair	20	11.7	22.3
	M70	GenProg	38	18.3	23.8
		WSGRepair	17	4	11.5
	M71	GenProg	64	28.9	42.7
		WSGRepair	26	10.4	21.5
	M73	GenProg	54	24.2	40.2
		WSGRepair	17	9.2	25.8
	M78	GenProg	120	82.7	118.4
		WSGRepair	57	17.3	30.8
	M80	GenProg	27	17.8	23.8
		WSGRepair	16	3.6	11.8
	M81	GenProg	19	20.2	34.1
		WSGRepair	11	2.1	12.2
	M82	GenProg	128	27.3	33.5
		WSGRepair	45	4.1	6.8
	M84	GenProg	125	90.1	106.5
		WSGRepair	66	16.8	32.5
	M85	GenProg	30	13.8	26.3
		WSGRepair	27	11.5	20.1
	M95	GenProg	87	21.8	27.1
		WSGRepair	31	3.5	13.5

was 1369.7 minutes, with an average. The total time spent by GenProg to repair these 26 buggy programs was 1369.7 minutes, with an average time of 52.7 minutes, while the total time spent by WSGRepair was 567 minutes, with an average time of 21.8 minutes. Compared with GenProg, the total time required for repair by WSGRepair is reduced by 58.6% and the average time taken is reduced by 58.6%, indicating that WSGRepair has a more obvious superiority in terms of time efficiency of buggy program repair.

In terms of the number of candidate patches generated NCP, the smaller the NCP value, the fewer attempts the buggy repair method has to make to find the correct repair patch. The 53 candidate patches are generated on average for each buggy program repaired by GenProg, while only 26 candidate patches are generated on average for one buggy program repaired by WSGRepair, which is 50.9% less than GenProg. This indicates that WSGRepair can repair buggy program after fewer program evolution times. Analyzing the data in Table 3, we can see that the total time span of GenProg to repair the buggy program is 106.2 minutes, and the maximum difference in NCP is 109. The total time span of WsGRepair to repair the buggy program is 57.5 minutes, and the maximum gap of NCP is 57. The repair time and NCP values of each buggy program in Table 3 show that the difficulty of repairing each buggy program is different and the time efficiency is also very different.

In order to visually compare the bug repair efficiency as well as to further explore the variability between different buggy program, a line graph of the total time spent on bug repair for both repair methods was drawn, as shown in Figure 5, with the x-axis representing the buggy program that can be repaired and the y-axis representing the total time spent on bug repair (in minutes). According to Figure 5, it can be visualized that the total time taken by WSGRepair to repair buggy program is much less than the total time required by GenProg, which indicates that WSGRepair has a great efficiency gain in terms of time efficiency of repairing buggy program.

Figure 6 is a histogram of the NCP comparison of the number of candidate patches, with the x-axis representing the buggy program and the y-axis representing the number of candidate patches generated by repairing that buggy program. As can be seen in Figure 6, for each different buggy program, the NCP values

Figure 5

Run time comparison with GenProg

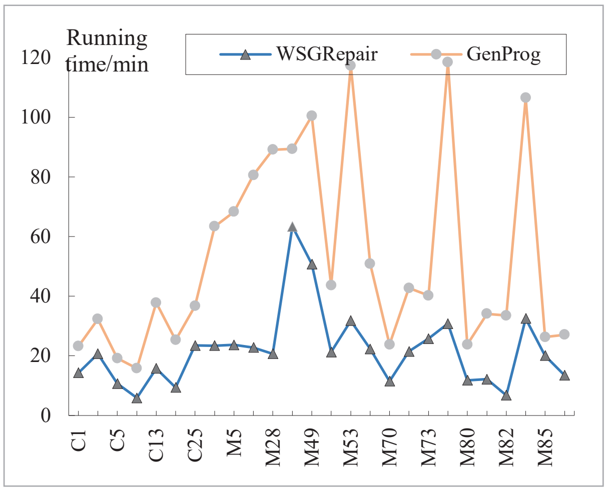
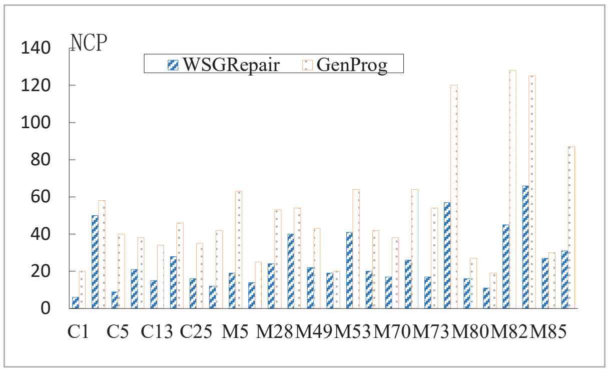


Figure 6

Comparison with GenProg NCP numbers

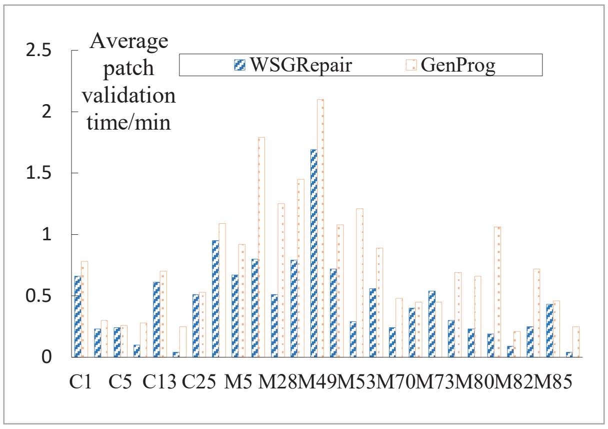


generated by the repair are different, and the repair efficiency improved by WSGRepair on each buggy program relative to GenProg is also different. Even for the same program, such as C1 to C25 and M2 to M95, the time taken to repair these bug programs and the number of program evolution repair are different, all with different degrees of data fluctuation. This is because each buggy program contains different types of bugs, the difficulty of repair, especially the bug items in the dataset Defects4J originated from the actual development, the size of the project and the project development time are different. For example, Math project development time up to 11 years. These factors lead to the actual repair of bug items when the difficulty of repair is also very different, repair different buggy program required the cost of repair different buggy program also varies. On the other hand, this also shows that in the actual industrial program bug repair, the repair law of automatic repair of buggy program is not obvious, and the difficulty of program repair cannot be clearly judged.

In order to improve the candidate patch verification efficiency, a test case prioritization technique based on the buggy identification capability is adopted in the verification process. To further show the efficiency improvement effect, Figure 7 shows the comparison of the average verification time for each individual program, with the x-axis representing the buggy program and the y-axis representing the average verification time of the buggy program (in minutes). According to Figure 7, it is obvious that the average verification time of WSGRepair when verifying individuals is smaller than that of GenProg, which indicates that the adopted test case sequencing technique can improve the efficiency of individual verification. Combined with the detailed data of individual verification in Table 3. The total verification time per buggy program repair by GenProg is 37.4 minutes on average, and the average verification time per candidate patch is 0.78 minutes. While the total verification time per buggy program repair by WSGRepair is 10.7 minutes on average, and the average verification time per candidate patch is 0.46 minutes. Compared with GenProg, the total verification time of WSGRepair can save 71.4% and the average verification efficiency can be improved by 41%, which shows that the test case prioritization technique adopted by WSGRepair can indeed speed up patch verification and improve individual verification efficiency.

Figure 7

Comparison with GenProg candidate patch validation time



RQ2: Can WSGRepair outperform the compared approaches?

We compare WSGRepair with four automatic program repair approaches. SimFix [7], CapGen [27], jKali [17] and jMutRepair [17], which have been evaluated on the Defects4J benchmark within our knowledge.

Table 4 shows the comparison results. The baselines results are directly extracted from existing literature [7, 17, 27]. Compared with these techniques, WSGRepair outperforms all of them in terms of the number of correctly repaired bugs and precision.

The number of repairs on the four major projects of Lang and Time is compared. From the table, we can see that WSGRepair can successfully repair 36 bug programs, CapGen can successfully repair 25 bug

Table 4

Comparisons with existing tools on Defects4J

project	bugs	WSGRepair	CapGen	SimFix	jKali	jMutRepair
Chart	26	9	4	4	6	4
Math	106	24	16	14	14	11
Lang	65	2	5	9	0	1
Time	27	1	0	1	2	1
Total	224	36	25	28	22	17
Precision	-	16.07%	11.16%	12.50%	9.80%	7.59%

programs, SimFix can successfully repair 28 bug programs, jKali can successfully repair 22 bug programs, and jMutRepair can successfully repair 17 bug programs. WSGRepair repair 11, 8, 14, and 19 more than CapGen, SimFix, jKali, and jMutRepair. Our approach is improving the success rate of buggy program repair by 28.6%, 64%, 29%, 64% and 112% compared with the GenProg, CapGen, SimFix, jKali and jMutRepair. The main reason why WSGRepair repair outperforms these four repair methods is that not only the similarity between the material and the modified point program fragment is taken into account, but also the similarity measure between the contextual environment information is introduced. It is able to find the repair more precisely materials and improve the repair efficiency.

RQ3: What repair actions does WSGRepair use to fix bugs?

In this research problem, the repair actions of WSGRepair are deeply studied. The type of repair operation is selected by WSGRepair to generate program patches. The specific defect types repaired by WSGRepair can be judged through the repair actions. We manually analyzed the 36 correct patches generated by WSGRepair on Defects4J and counted the different repair actions to generate the correct patches.

It can be seen from Table 5 that WSGRepair uses 17 types of repair actions. The use of multiple repair actions has proven that WSGRepair can repair many different types of bugs, therefore WSGReapir can be applied to the industrial scenarios in the future.

The above results illustrate that WSGRepair can repair many types of bugs. It can be seen from Table 6 that WSGRepair repairs more bug types than CapGen, SimFix, jKali and jMutRepair.

The experiment results show that WSGReapir can repair different types of bugs. Our method will yield decent performance if it applies to C, C++ and python languages, because Java language has many similarities with the language types of the three languages C, C++ and python.

We take the program with the buggy program and its current test suites as input. There is at least one failed test case that makes the program fail. The output is zero for the automatic program repair method, or the output is one patch or multiple patches. The automatic program repair consists of two parts: how to find the location of the bug and how to generate the code segment for the bug. To address the repair problem, we treat the buggy program and its code changes at a potential location as an individual, and treat all such individuals as a huge search space. When we look for individuals which can fix bugs in the search space, we make changes to the code fragments (such as additions, deletions, or modifications to the code),

Table 5
Description of Repair actions

Repair Actions		Number of times repair operations
Assignment	modification	6
	addition	10
Conditional Expression	modification	5
	removal	1
	expansion	2
Conditional Branch	addition	4
Method Call	modification	2
	removal	5
	addition	3
Loop	addition	1
Variable	addition	2
	modification	5
	removal	1
Return Expression	addition	1
	modification	6
Object Instantiation	modification	2
	addition	2

Table 6
Number of repair actions compared to other repair methods

Repair Actions	Total number of repair types
WSGRepair	17
CapGen	12
SimFix	14
jKali	10
jMutRepair	9

generate patches after the changes, and finally verify whether the generated patches pass the test suites. Passing the test suites is an important indicator to measure the executable after we repair the source code, because the number of candidate patches generated is large, and other bugs may be introduced.

5. Threats to Validity

In this work, we on Defects4J evaluated our approach in repair 36 real world bugs, a threat to our work being that the number of bugs was not large enough nor representative. In addition to Defects4J we also use IntroClassJava for validation to mitigate the threat of WSGRepair generality. The reason we choose the Java language. 1.Using the same data set is convenient for comparison and verification of different methods. 2.The ease of use of the dataset. However, a single dataset may cause the automatic repair method to face the problem of overfitting and affect the objective evaluation of its effect. Therefore, in future research, different datasets should be utilized. We will generalize to datasets in other languages in our follow-up work. Le Goues et al. [12] propose a C language IntroClass dataset. IntroClassJava is the Java language version of the IntroClass dataset, by Durieux and Monperrus [2] using script automation to convert C/C++ language programs to Java programs by manually defining conversion rules. On the other hand, our job is to evaluate actual bugs in large Java programs, and we note that it is possible to collect more bugs in the real world, but it requires more human effort.

WSGRepair performs a weighted fusion of feature similarity using program fragment features and contextual environment features between ingredient and modification points. The technique searches for similar code at the statement level, combines the relevant variables in the buggy statement and the contextual environment features of the method it is in to select the repair ingredient. Evolution of buggy program using genetic programming to find valid patches that pass all test cases. To accelerate patch verification and improve patch verification efficiency, test cases are prioritize using a test case prioritization technique based on bug recognition capabilities.

6. Related Work

In order to reduce the time and labor cost of the repair process, the automatic program repair method has been developed. This method automatically generates patches to repair buggy in the program based on a given program problem. Le Goues et al. [11,13,26] was the first to apply genetic programming to the repair of buggy program and proposed GenProg, a repair tool based on redundancy theory and plastic surgery conjectures. GenProg applies genetic programming to mutate existing source code for patch generation. However, GenProg does not select the repair ingredient based on the attribute characteristics of the current modified location. We use a weighted fusion similarity metric to select repair ingredients for the point to improve the repair success rate. In 2016, a study by Yokoyamaet et al. [31] and others indicated that using code similarity can effectively reduce the size of the search space and accelerate the repair process. SimFix [7], CapGen [27], LSRepair [14] and DeepRepair [28] have implemented corresponding repair tools based on the above theory and demonstrated the effectiveness of the proposed method on the dataset Defects4J [8].

For the genetic programming class of bug repair methods, a fine-grained fitness function was proposed by de Souza et al. [1], who suggested the use of intermediate program states. However, this technique involves inspecting the source code and the collection and use of checkpoints introduces additional overhead. The fitness function proposed by Jang et al. [4], which records the number of modification points touched by a test case in addition to the failed test cases, is a more lightweight approach than de Souze et al. [1] approach, but it does not take into account the test case output results. In the study by Petke et al. [21], a two-stage adaptation function was proposed. First, the usual Boolean fitness assessment is used to compare program mutants, preferentially selecting the program individual that passes more test cases. However, when two program individuals have the same Boolean fitness, a more fine-grained calculation of the fitness function is used based on the distance between the individuals expected and actual output. WSGRepair use the test case execution information to determine the program repair status, if there is a child individual can pass all the test cases, then the child individual will be

Table 6
Comparison between automatic repair methods

Automatic repair method	Language	Characteristic
CapGen	Java	CapGen not only increases the number of correct patches in the search space, but also solves the problem of further expanding the search space.
DeepRepair	Java	White et al. proposed an automatic program repair technology (DeepRepair) to infer code similarity by deep learning. DeepRepair can sort code fragments according to the similarity of suspicious elements, and convert statements by mapping identifiers outside the range to similar identifiers within the range.
Genpat	Java	Jiang proposed a program conversion method, called GENPAT. GENPAT infers program transformation through the statistical information of code context and large code corpus, it can be used in system editing and program repair.
SimFix	Java	SimFix extracts an abstract search space from existing patches. Then extracts similar codes from buggy source programs to form a specific candidate patch search space. Finally, SimFix obtains the program patch by the intersection of the above two search spaces.
LSRepair	Java	LSRepair utilizes three similar code search strategies on the code repository in method-level granularity to search for repair ingredients.
PraPR	JVM bytecode	PraPR repaired buggy code on the basis of bytecode, implementing the first bytecode-level repair tool. Experimental results show that bytecode-mutation-based program repair techniques have good prospects for repair applications.

considered as the repair patch of the bug program, complete the repair task, stop the repair and output the program patch.

Many existing techniques are dedicated to identifying the differences between two codes and considering the characteristics between the codes. White et al. [28] proposed DeepRepair, an automatic repair technique that uses deep learning to reason about code similarity. The technique can sort code fragments based on similarity to suspicious elements and can transform statements to in-range similar identifiers by mapping out-of-range identifiers. The repair method ssFix proposed by Xin et al. [29] searches for code similar to the buggy code from the code base as the ingredient for repairing the patch. Wang et al. [24] also proposed the repair tool CRSearcher based on the basic idea of reusing similar code. CRSearcher extends the code search to other projects, applies token-based code similarity measures, and does not require similar code to be necessarily different from buggy code. Jiang et al. [7] proposed SimFix which extracts an abstract search space from existing patches and similar code from bug source programs to form a concrete candidate patch search space, and obtains program patches from the intersection of these two search spaces. Subsequently, Jiang et al. [6] proposed another program transformation method GENPAT, which inferred program transformations based on code context and statistical information from a large corpus of code, and could be used for both system editing and application to program repair. WSGRepair determines the similarity between program fragments with the help of the program's attribute characteristics in order to select the repair ingredients needed for the repair.

Wen et al. [27] proposed CapGen, a context-aware technique for automatic repair of software defects, which increases the number of correct patches in the search space and also solves the problem of further expansion of the search space due to the use of finer granularity, making it more difficult to search. The problem of further expanding the search space and increasing the search difficulty due to the use of finer granularity. Kim et al. [9] extracted abstract syntax tree changes from manually written patches and contextual environments, and only locations with the same contextual environment changes were made to filter out invalid change locations. In selecting the Ingredients, WSGRepair mainly consider the method characteristics of the location of the bug statement and the property characteristics of the variables contained in the bug statement, while taking into account the semantic characteristics contained in the program statement itself. Unlike the approach of bug repair using contextual environment, Ghanbari et al. [3] repaired buggy program on the basis of bytecode, implementing the first bytecode-level repair tool PraPR. PraPR experimental results show that bytecode-mutation-based program repair techniques have good prospects for repair applications. Table 7 lists comparison between the repair in the automatic repair method and the types of languages supported.

7. Conclusion

In this paper, we propose a novel automatic program repair approach, which is based on weighted fusion similarity with genetic programming. WSGRepair can find the correct repair ingredient faster and improve the efficiency of buggy program repair by combining the ingredient with the environmental characteristics of the modification point. At the same time, in order to speed up the efficiency of patch verification, when verifying a single program, test cases are prioritized using the test case prioritization technique. We conducted a large-scale experimental study of 224 real world bugs on the Defects4J benchmark. We find out that the systems under consideration can synthesize patch for 36 out of 224 bugs.

There is still a lot of work to be done to fully utilize the advantages and potential of automatic program repair technology in the software maintenance process. For example, our future work includes refine the repair granularity and expand ingredient space to enhance its performance on real world bugs. Therefore, in future work, we will collect AST changes and their AST contexts from human-written patches to provide a rich resource for patch generation.

Acknowledgements

This work was partially supported by Cultivation Programme for Young Backbone Teachers in Henan University of Technology, Key scientific research project of colleges and universities in Henan Province (No.22A520024), Major Public Welfare Project of Henan Province (No.201300311200) and National Natural Science Foundation of China (Nos. 61602154, 61340037, 62206087).

References

1. de Souza, E. F., Goues, C. L., Camilo-Junior, C. G. A Novel Fitness Function for Automated Program Repair Based on Source Code Checkpoints. Proceedings of Proceedings of the Genetic and Evolutionary Computation Conference, 2018, 1443-1450. https://doi.org/10.1145/3205455.3205566
2. Durieux, T., Monperrus, M., IntroClassJava: A Benchmark of 297 Small and Buggy Java Programs. [Research Report] hal-01272126, Universite Lille 1. 2016, Universite Lille 1, 2016.
3. Ghanbari, A., Zhang, L., PraPR: Practical Program Repair Via Bytecode Mutation. Proceedings of 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, 2019, 1118-1121. https://doi.org/10.1109/ASE.2019.00116
4. Jang, Y., Phung, Q N., Lee, E., Improving the Efficiency of Search-Based Auto Program Repair by Adequate Modification Point. Proceedings of International Conference on Ubiquitous Information Management and Communication. Springer, Cham, 2019, 694-710. https://doi.org/10.1007/978-3-030-19063-7_56
5. Ji, T., Chen, L., Mao, X., et al. Automated Program Repair by Using Similar Code Containing Fix Ingredients. Proceedings of 2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC). IEEE, 2016, 1: 197-202. https://doi.org/10.1109/COMPSAC.2016.69
6. Jiang, J., Ren, L., Xiong, Y., et al. Inferring Program Transformations from Singular Examples Via Big Code. Proceedings of 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, 2019: 255-266. https://doi.org/10.1109/ASE.2019.00033
7. Jiang, J., Xiong, Y., Zhang, H., et al. Shaping Program Repair Space with Existing Patches and Similar Code. Proceedings of 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, 2018, 298-309. https://doi.org/10.1145/3213846.3213871

8. Just, R., Jalali, D., Ernst, M D., Defects4J: A Database Of Existing Faults to Enable Controlled Testing Studies for Java Programs. Proceedings of 2014 International Symposium on Software Testing and Analysis. 2014: 437-440. https://doi.org/10.1145/2610384.2628055

9. Kim, J., Kim, S., Automatic Patch Generation with Context-Based Change Application. Empirical Software Engineering, 2019, 24(6), 4071-4106. https://doi.org/10.1007/s10664-019-09742-5

10. Koza, J. R. Genetic Programming as a Means for Programming Computers by Natural Selection. Statistics and computing, 1994, 4(2), 87-112. https://doi.org/10.1007/BF00175355

11. Le Goues, C., Dewey-Vogt, M., Forrest, S., et al. A Systematic Study of Automated Program Repair: Fixing 55 out of 105 Bugs for $8 Each. Proceedings of 2012 34th International Conference on Software Engineering (ICSE). IEEE, 2012, 3-13. https://doi.org/10.1109/ICSE.2012.6227211

12. Le Goues, C., Holtschulte, N., Smith, E. K., et al. The ManyBugs and IntroClass Benchmarks for Automated Repair of C Programs. IEEE Transactions on Software Engineering, 2015, 41(12): 1236-1256. https://doi.org/10.1109/TSE.2015.2454513

13. Le Goues, C., Nguyen, T V., Forrest, S., et al. Genprog: A Generic Method for Automatic Software Repair. IEEE Transactions on Software Engineering, 2011, 38(1), 54-72. https://doi.org/10.1109/TSE.2011.104

14. Liu, K., Koyuncu, A., Kim, K., et al. LSRepair: Live Search of Fix Ingredients for Automated Program Repair Proceedings of 2018 25th Asia-Pacific Software Engineering Conference (APSEC), IEEE, 2018, 658-662. https://doi.org/10.1109/APSEC.2018.00085

15. Long, F., Amidon, P., Rinard, M. Automatic Inference of Code Transforms for Patch Generation. Proceedings of 2017 11th Joint Meeting on Foundations of Software Engineering, 2017, 727-739. https://doi.org/10.1145/3106237.3106253

16. Long, F., Rinard, M. Automatic Patch Generation by Learning Correct Code. Proceedings of 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 2016, 298-312. https://doi.org/10.1145/2837614.2837617

17. Martinez, M., Monperrus, M. Astor: A Program Repair Library for Java. Proceedings of International Symposium on Software Testing and Analysis, Saarbrücken Germany. CM, 2016, 441-444. https://doi.org/10.1145/2931037.2948705

18. Mechtaev, S., Yi, J., Roychoudhury, A., Directfix: Looking for Simple Program Repairs. Proceedings of 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering. IEEE, 2015, 1, 448-458. https://doi.org/10.1109/ICSE.2015.63

19. Nguyen, H D T., Qi, D., Roychoudhury A, et al. Semfix: Program Repair Via Semantic Analysis. Proceedings of 2013 35th International Conference on Software Engineering (ICSE). IEEE, 2013, 772-781. https://doi.org/10.1109/ICSE.2013.6606623

20. Pawlak, R., Monperrus, M., Petitprez, N., et al. Spoon: A Library for Implementing Analyses and Transformations of Java Source Code. Software: Practice and Experience, 2016, 46(9): 1155-1179. https://doi.org/10.1002/spe.2346

21. Petke, J., Blot, A. Refining Fitness Functions in Test-based Program Repair. Proceedings of IEEE/ACM 42nd International Conference on Software Engineering Workshops, 2020, 13-14. https://doi.org/10.1145/3387940.3392180

22. Riboira, A., Abreu, R. The GZoltar Project: A Graphical Debugger Interface. Proceedings of International Academic and Industrial Conference on Practice and Research Techniques. Springer, Berlin, Heidelberg, 2010, 215-218. https://doi.org/10.1007/978-3-642-15585-7_25

23. Saha, R. K., Lyu, Y., Yoshida, H., et al. Elixir: Effective Object-oriented Program Repair. Proceedings of 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, 2017, 648-659. https://doi.org/10.1109/ASE.2017.8115675

24. Wang, Y., Chen, Y., Shen, B., et al. CRSearcher: Searching Code Database for Repairing Bugs. Proceedings of 9th Asia-Pacific Symposium on Internetware, 2017, 1-6. https://doi.org/10.1145/3131704.3131720

25. Weimer, W., Fry, Z P., Forrest, S., Leveraging Program Equivalence for Adaptive Program Repair: Models and First Results. Proceedings of 2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, 2013, 356-366. https://doi.org/10.1109/ASE.2013.6693094

26. Weimer, W., Nguyen, T V., Le Goues, C., et al. Automatically Finding Patches Using Genetic Programming. Proceedings of 2009 IEEE 31st International Conference on Software Engineering. IEEE, 2009, 364-374. https://doi.org/10.1109/ICSE.2009.5070536

27. Wen, M., Chen, J., Wum R., et al. Context-aware Patch Generation for Better Automated Program Repair. Pro-

ceedings of 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE). IEEE, 2018, 1-11. https://doi.org/10.1145/3180155.3180233

28. White, M., Tufano, M., Martinez, M., et al. Sorting and Transforming Program Repair Ingredients Via Deep Learning Code Similarities. Proceedings of 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER). IEEE, 2019, 479-490. https://doi.org/10.1109/SANER.2019.8668043

29. Xin, Q., Reiss, S. P., Leveraging Syntax-related Code for Automated Program Repair. Proceedings of 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, 2017: 660-670.27. https://doi.org/10.1109/ASE.2017.8115676

30. Xiong, Y., Wang, J., Yan, R., et al. Precise Condition Synthesis for Program Repair. Proceedings of 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE). IEEE, 2017: 416-426. https://doi.org/10.1109/ICSE.2017.45

31. Yokoyama, H., Higo, Y., Hotta, K., et al. Toward Improving Ability To Repair Bugs Automatically: A Patch Candidate Location Mechanism Using Code Similarity. Proceedings of 31st Annual ACM Symposium on Applied Computing, 2016, 1364-1370. https://doi.org/10.1145/2851613.2851770