symmetry

MDPI

Article

Applying Genetic Programming with Similar Bug Fix Information to Automatic Fault Repair

Geunseok Yang [1], Youngjun Jeong [1], Kyeongsic Min [1], Jung-won Lee [2] and Byungjeong Lee [1,*]

[1] Department of Computer Science, University of Seoul, Seoul 02504, Korea; ypats87@uos.ac.kr (G.Y.); realjuni@uos.ac.kr (Y.J.); ksmin1710@uos.ac.kr (K.M.)
[2] Department of Electrical and Computer Engineering, Ajou University, Suwon 16499, Korea; jungwony@ajou.ac.kr
* Correspondence: bjlee@uos.ac.kr; Tel.: +82-2-6490-2451

Received: 19 February 2018; Accepted: 28 March 2018; Published: 2 April 2018

Abstract: Owing to the high complexity of recent software products, developers cannot avoid major/minor mistakes, and software bugs are generated during the software development process. When developers manually modify a program source code using bug descriptions to fix bugs, their daily workloads and costs increase. Therefore, we need a way to reduce their workloads and costs. In this paper, we propose a novel automatic fault repair method by using similar bug fix information based on genetic programming (GP). First, we searched for similar buggy source codes related to the new given buggy code, and then we searched for a fixed the buggy code related to the most similar source code. Next, we transformed the fixed code into abstract syntax trees for applying GP and generated the candidate program patches. In this step, we verified the candidate patches by using a fitness function based on given test cases to determine whether the patch was valid or not. Finally, we produced program patches to fix the new given buggy code.

Keywords: automatic fault repair; genetic programming; bug fix information; software maintenance

1. Introduction

Owing to the high complexity of recent software products, developers cannot avoid major/minor mistakes, and software bugs are generated during the software development process. In open source projects, a large number of bug reports containing 350 bugs a day are submitted to the bug repository for Eclipse and Mozilla [1]. Because of the huge number of bug reports, developers spend more time fixing bugs, thus increasing their workloads and costs. To reduce their bug fixing efforts, automatic fault repair is necessary.

In the general bug-fixing process, developers try to fix software bugs according to the descriptions in bug reports by creating a patch solution. After that, quality assurance engineers may check the patch and then update the program with the patch solution. Because of the large number of daily bugs, developers spend more time tracing the bugs; hence, they may generate incorrect patches.

Our motivations are the following:

- As developers' workloads increase owing to many daily bugs, they may spend more time debugging to fix these. If an automatic fault repair technique is provided to fix bugs, program debugging time and cost can be reduced.
- Developers may make mistakes in debugging the program source code. As a result, they may generate incorrect program patches. Thus, if an automatic fault repair technique that generates the correct patch is provided, software quality will improve significantly.

- If bug reporters give descriptions with helpful information to developers such as stack trace and scenario reproduction, the developers can trace and fix the bugs easily. However, if the bug reports contain insufficient information [2], the developers may have difficulty debugging. Thus, it is expected that the automatic fault repair with similar bug fix information can effectively fix the bugs despite lacking descriptions.

To address these problems, many researchers have proposed automatic fault repair approaches. To our best knowledge, GenProg [3] is well-known for automatic fault repair. They utilize Genetic Programming (GP) to generate the program patches. However, as GenProg conducts patch validation by passing given test cases, it might require too much time and generation cycles. PAR [4] captures fault patterns by analyzing human-written patches. However, if it encounters patterns that it did not capture, it cannot fix the bugs. RSRepair [5] utilizes a random search algorithm instead of the GP algorithm on GenProg. AE [6] uses a deterministic patch search algorithm and program equivalence relations to prune equivalent patches during testing. However, even if the patches pass all the test cases to verify whether the patches can be adopted or not, they may not be correct. To produce the correct patches, we utilized bug fix information in this paper. In web applications, PHPQuickFix [7] and PHPRepair [7] use a string constraint solving technique to automatically fixed HTML generation errors. FixMeUp [8] automatically fixes the missing access-control statement using program analysis. Prophet [9] is based on learning the correct code. The work extracts code interaction information from the fixed code and changes the nearby code. By applying a machine learning algorithm, it generates a new patch. SPR [10] generates the correct patches in large-scale applications. To generate the candidate program patches, the work utilizes a set of transformation schemas. Although it generates program patches, the performance should be improved.

To resolve these problems, we propose a novel approach for automatic fault repair by using similar bug fix information based on GP. First, we searched for similar buggy codes related to a new given buggy code and the fixed code of the similar buggy codes. Then, we converted the source code into abstract syntax trees (ASTs). We applied GP with the ASTs of the fixed codes. Finally, we generated the program patches for the new given buggy code from the ASTs.

Our contributions are the following:

- The bug fixing time and effort can be reduced as we support automatic fault repair.
- The quality of bug fixing can be improved as we utilize similar bug fix information with GP to generate program patches.
- We perform a small case of study using our model in IntroClass [11]. The result will likely generate a correct patch.

This paper is organized as follows. We describe the background information on fault repair in Section 2. In Section 3, we discuss related studies. We describe our approach in Section 4 and present a case study in Section 5. Then, we provide the discussion in Section 6. Finally, we conclude our study in Section 7.

2. Background Knowledge

Genetic programming (GP) is a programmatic approach based on evolutionary algorithm. In order to apply GP, the input buggy code should first be represented in AST. Next, we adopted GA [12] operators such as selection, mutation, and crossover to generate a new population. Then, we utilized a fitness function to evaluate the validity of each patch in the population. Finally, we transformed the valid patch into a program source code to patch the given buggy code.

Owing to the random generation characteristic of GP, the quality of program patches can decline. To ensure patch quality, we carried out program analysis [13] using a test suite. In this case, we are likely to improve the quality of candidate program patches, because the GP process and the analysis are independent. If we adopt white-box and black-box test cases, the overall quality of the patches is expected to improve.

3. Related Work

Many researchers have proposed automatic fault repair approaches. We provide a qualitative comparison of related works on automatic fault repair, as shown in Table 1.

Table 1. A Qualitative Comparison.

Study	Evolutionary Algorithm	Metric
GenProg [3]	O	Basis of GP for Repairing
PAR [4]	X	Bug Fix Pattern Templates
AE [6]	X	Deterministic Algorithm
Prophet [9]	X	Machine Learning Algorithm
SemFix [14]	X	Component-based Synthesis
Yokoyama [15]	O	Code Similarity (Line based)
Our Approach	O	Bug Fix Information

Le Goues et al. proposed GenProg [3], which is a popular approach in this fault repair domain. They utilized AST-based GP. First, they transformed an input buggy source code into AST (e.g., original AST). Then, they adopted GP operations like selection, crossover, and mutation to generate a new AST. Next, the new AST transform program source code was used to verify whether the buggy code was fixed using a test case. Finally, they generated a new program patch. However, during the generation, it occurred too many times and in too many generation cycles. Similar to GenProg, Qi et al. utilized GP operations, including selection and crossover as well as test case prioritization [16].

Kim et al., proposed PAR [4], an automatic fault repair method using a fixing template from human-written patches. They verified a buggy pattern and classified several pattern templates. In detail, they classified the target program code by fault localization into the related code in the pattern template. Then, they computed the fitness function from the target source code using a test case. Next, they selected the related code using tournament selection. Finally, they compared the new generated patch to the fixed source code from developers. However, they could fix a new bug if it does not exist in the proposed pattern.

Long et al. introduced SPR [10] and improved the performance of fault repair by more than five times compared with previous work using parameterized transformation schemas. However, they could not fix the bug correctly if they needed a new condition that did not exist in the program.

Qi et al., presented an automated fault repair technique called RSRepair [5] that uses random search. Then, they adopted test case prioritization to speed up the patch validation process. Finally, they showed results that outperformed the GenProg results. Weimer et al. proposed an approach called AE [6]. They used a deterministic patch search algorithm and program equivalence relations to prune equivalent patches during testing. However, the results of RSRepair and AE did not generate the correct patches.

Long et al. proposed Prophet [9] based on learning the correct code. They extracted code interaction information from the fixed code and changed nearby code. Then, they adopted a machine learning algorithm to generate a new patch by learning.

Nguyen et al. proposed SemFix [14] to fix the buggy program code. They utilized semantic information via source code and the SemFix outperformed the GenProg.

Yokohama et al. proposed an automated repair approach [15] based on source code analysis. In detail, they analyzed changed source lines in before/after patches.

In web applications, PHPQuickFix [7] and PHPRepair [7] utilized string constraint solving techniques to automatically fix HTML generation errors.

In the data mining area, Guo et al. proposed [17] an approach for bug severity prediction. They analyzed descriptions of bug reports to predict bug severity in an Android project. Singh et al. proposed an approach [18] for sentiment analysis using machine learning. Souri et al. surveyed [19] malware detection approaches using data mining.

In terms of testing, the GA technique was proposed by Sabharwal [20] to generate test cases for pairwise testing.

The main differences are the following:

- If we do not consider bug fix information, the buggy code cannot be fixed correctly by GP. In this paper, we first find the most similar buggy codes related to the new given buggy code in order to find the related fixed code. Thus, we can generate a program patch for the buggy program.
- If a bug pattern and template are adopted, the various bugs that do not exist in the proposed pattern will not be fixed. Thus, we utilized a buggy code and a related fixed code in order to fix various buggy codes.

4. Applying Genetic Programming with Similar Bug Fix Information

In this section, we describe our approach for automatic fault repair by applying GP with similar bug fix information. First, we identified suspicious buggy code lines (a) using a fault localization technique. Then, we applied our approach to generate patches (b-1) by applying GP with similar bug fix information. Next, we verified the candidate patches to determine whether the patches could be adopted using test cases (b-2). If the patches could not be adopted (e.g., test case failure), we repeated our patch generation (b-1). Finally, we applied the patches to the buggy program for fixing (c). The overview of our approach is shown in Figure 1.

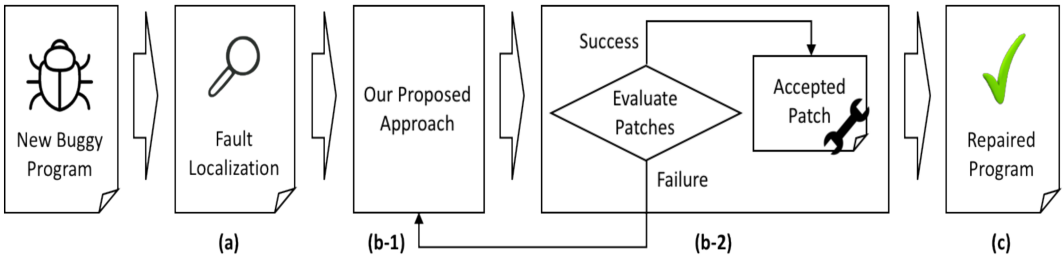

Figure 1. An overview of our approach. (**a**) Fault Localization; (**b**) Applying GP (**b**-1), and Patch Evaluation (**b**-2); (**c**) Fixed Program Generation.

4.1. Fault Localization

In general, we first identified the program source code lines that are buggy. Identifying the buggy code lines is a prerequisite for repair. For example, to fix the buggy program, e.g., multiple buggy lines, the fault localization technique should be performed. Many researchers have proposed techniques to find buggy code lines using the information retrieval model [21] and Latent Dirichlet Allocation [22]. However, the aims of these fault localization studies are different from those of fault localization for automatic fault repair, in the sense that all the correctly identified lines are used together for repair. First, we verified whether a program can pass all the black box and white box test cases. If the program cannot pass all the test cases, we modified the source code and ran the test cases again to ensure that the modified program was correct. After that, we identified the modified line in the program. This information was used in the application of our approach and makes the repair better.

4.2. Converting Code to AST

To apply the GP technique, we transformed the program source codes into AST. The AST conversion can be expressed by a relation between AST nodes and the source code as follows:

$$\mathrm{AST}_{Target} = \mathrm{ASTConversion}\left(CodeLines_{Target}\right), \quad (1)$$

- *Target* indicates the original buggy code and similar fixed buggy code.

- *CodeLines* are source code lines.

After that, we changed the source codes by modifying AST nodes. Then, we generated source codes from the changed AST nodes.

4.3. Applying GP

In this section, we describe the application of GP with similar bug fix information. Creating the initial population, GP operation, fitness computation, etc., are shown in Figure 2. The GP operations consist of selection, crossover, and mutation. The initial patches were created from random generation and each patch was evaluated using a fitness function. While applying the GP operations, we generated the candidate patches to fix the buggy code. They were also evaluated using a fitness function to determine whether they were adoptable. For the fitness function, we utilized test cases in a test suite for each program. If the fitness value of a patch is 1.0, it means that the patch passes all test cases and is the correct patch for the buggy program. If not, our approach repeats the GP generations.

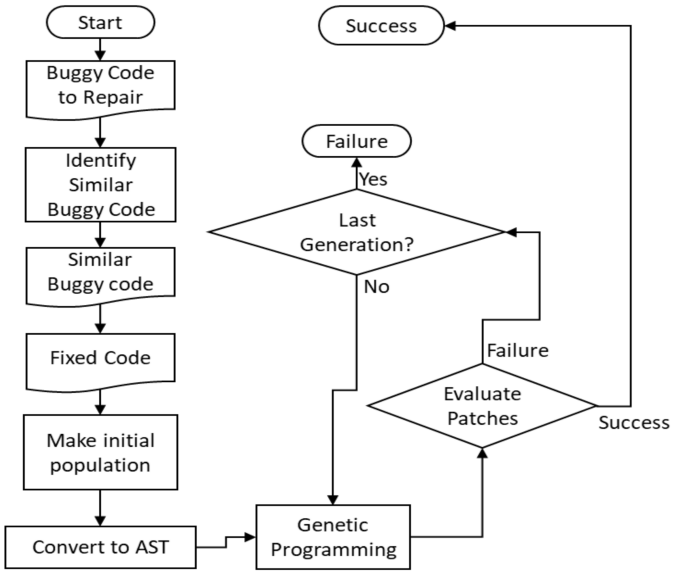

Figure 2. A detailed view of b-1 and b-2 in Figure 1.

4.3.1. Making Initial Population

First, we searched for buggy codes similar to the given buggy code by utilizing a clone detection technique [23] and obtained the fixed code of the most similar code. We produced an initial population by utilizing the similar lines in the fixed code. To produce an initial population, we found lines in the fixed code that were similar to buggy lines in the buggy code identified by fault localization technique. To find similar lines, we transformed each line into token sequences and compared token sequences between those lines using the longest common subsequence (LCS) [24]. We took the token sequence with the highest LCS value from the lines. We compared the token sequences between a buggy line and the similar fixed buggy code using LCS. Then, we took the token sequence with the highest LCS values from the lines.

We produced each solution by utilizing a random operator, except for keywords and names of variables and functions from the token sequence. A constant in a line can be adopted from both the buggy code and fixed code. Then, we produced a new line from the changed token sequence and swapped that line with the buggy line. This step is repeated until the number of the initial patches is equal to the size of the population.

4.3.2. GP Operation

To produce each patch, we first selected two candidate patches from the population using a selection operator, and executed crossover and mutation operators. It is expected that the nodes in the buggy line were changed by the GP operations. In the crossover operation, we start with ASTs of the two selected candidate patches. We describe the details of GP operations as follows:

- Selection: Fitness normalizing process was applied to guarantee the probabilities for patches that have low fitness. To normalize fitness, we used the following process.

$$\text{FitNormal(p)} = \frac{(\text{largestFitness} - \text{smallestFitness})}{n} + (\text{fitness}(p) - \text{smallestFitness}), \quad (2)$$

- We first sorted the candidate patches in the population in descending order from the previous generation using the fitness values. Then, we computed the difference between the largest and smallest fitness values. We divided the difference by the normalizing factor "n" and added the difference between the fitness value and the smallest fitness value in the generation to obtain the normalized fitness value. The smaller the value of the normalizing factor "n", the greater the probability that a patch with a small fitness value will be selected. This normalized fitness value was used only for selection. The two ASTs from the current population were selected to construct two children using a roulette wheel selection operator with the normalized fitness value. According to roulette wheel selection, the patch that has the greater normalized fitness value will be selected with greater probability. Then, we used the two parent patches to generate two new children patches.
- Crossover: The crossover aims to exchange the sub-tree nodes in two ASTs. To do this, we first selected the changeable target nodes in each parent tree. We then constructed a set of target nodes from the buggy line to the nodes within "r" lines, including the lines above and below. The factor "r" refers to the range of the patch target code. After selecting the target nodes, we verified whether the two nodes can be exchanged with each other by tracing the parent's node. Then, we exchanged the changeable nodes.
- Mutation: In this operation, the target node will be removed, added, or modified. The node was selected from the target set. The set is the same as the target set in the crossover, but the set can have different a code range, and the set is not dealt with as crossover target set. There are three sub-operations: deletion, addition, and modification. Deletion removes the target node from the AST. Addition inserts a copied node of its own tree. Modification changes the operator of the target node. When the modification is executed, an operator is selected from an operator list, except for the original operator. However, the mutation operator introduces variety into the population in a positive or negative way; thus, the operator is adjusted by a suitable probability parameter.

4.3.3. Fitness Function

Owing to the random generation characteristic of GP, the generated candidate patches should be evaluated by a fitness function to determine whether the patches are adoptable. The fitness function is expressed as follows:

$$\text{Fitness}(Program_{\text{candidate}}) = \frac{|Passed\ WhiteB| + |Passed\ BlackB|}{|WhiteB| + |BlackB|}, \quad (3)$$

- $Program_{\text{candidate}}$ is a candidate program patch from the result of the GP operation.
- *Passed WhiteB* is the number of test cases passed among white box test cases from a given test suite. *Passed BlackB* is the number of test cases passed among black box test cases from a given test suite.
- *WhiteB* and *BlackB* are the total number of test cases in white box and black box, respectively.

From Equation (3), we consider the generated patch as correct if the fitness value is 1.0. If not, the patch cannot be adopted to fix the given buggy program. The larger fitness values are better than the smaller fitness values. Our approach can be described in the following pseudocode as shown in Figure 3.

```
p: a target buggy code, p_sim : a patched code of the most similar buggy code
l: a sequence line number, l_buggy : a sequence buggy line number
L: a set of target lines, TokenSeqSet: a set of token sequence
ConstantSet: a set of constants in the original program
VarNameSet: a set of the names of the variables in the original program
1   l_buggy ← FaultLocalization(p)
2   for from i = 0 until i ≤ codeRegionRange
3   L ← GetLine(l_buggy + i)
4   L ← GetLine(l_buggy − i)
5   i ← i + 1
6   for from currentPopSize = 0 until currentPopSize ≤ initialPopSize
7   str = GetOne(L)
8   ConstantSet ← GetConstant(str)
9   VarNameSet ← GetVarName(str)
10  longestLCS = 0
11  longestSeq = null
12  for from m = 1 until m ≤ GetLineNum(p_sim)
13  currentLCS ← GetLCS(TokenTypeSeq(p, l),TokenTypeSeq(p_sim, m))
14  if (currentLCS > longestLCS)
15  longestLCS ← currentLCS
16  LongestSeqSet = ∅
17  LongestSeqSet ← TokenTypeSeq(p_sim, m)
18  else if (currentLCS == longestLCS)
19  LongestSeqSet ← LongestSeqSet ∪ TokenTypeSeq(p_sim, m)
20  longestSeq = GetOne(LongestSeqSet)
21  str = GetModifiedLine(longestSeq, ConstantSet, VarNameSet)
22  ModifyLine(p, l, str)
23  currentPopSize ← currentPopSize + 1
```

Figure 3. A pseudocode of our approach.

To generate the initial population, we first set the target lines (lines 1–5). In this process, we assume that we already know where the fault is (line 1). Then, we set the target lines around the buggy line (lines 2–5). One of these lines will be modified to generate the initial population (line 7). After setting the target line, we collect the constants and the names of the variables in the line (lines 8 and 9). This set is used to construct a new line (line 21). To select a line from patched code, we compute the LCS value between the token type sequence of the target line and the token type sequence of lines from the correct patched code of the similar bug (line 13). In detail, variables are changed to "*varname*", operators are changed to "*op*", and constants are changed to "*constant*" (e.g., "*sum = sum% 64 + 32*" will be changed to [*varname = varname*, *op*, *constant*, *op*, *constant*]). Then, we obtain the LCS values of each line. The line that has the largest LCS value is selected to write the new line. If there are several lines with the same LCS value, we select a line randomly (line 20). We construct a new code line from the token type sequence that has the largest LCS value (line 21). The token type "*varname*" is changed to one of the names of the variables in the original code, and the token type "*constant*" is changed to one of the constants of the original buggy code line and the selected patched code line. The token type "*op*" is changed to a random operator. The selected line of buggy code is alternated by a new line (line 22).

5. Case Study

In this section, we present a case of study of our approach. First, we utilized a benchmark dataset from IntroClass [11]. The dataset consists of checksum, digits, grade, median, smallest, and syllables programs. In this case study, we used the checksum program as shown in Figure 4.

```
1 //
2 #include <stdio.h>
3 #include <math.h>
4 int main()
5 {
6 int add, sum;
7 sum = 0;
8 printf("Enter an abitrarily long string, ending with carriage return > ");
9 while(add != '\n')
10 {
11 add = getchar();
12 if(add != '\n')
13 sum = add + sum;
14 }
15 //
16 sum = sum - 32;
17 printf("Check sum is ");
18 putchar(sum);
19 printf("\n");
20 return 0;
21 }
```

Figure 4. A sample of buggy code.

In Figure 4, an *add* variable in Line 6 can be an initial value problem in a runtime execution; thus, the variable cannot be affected by the test case. However, Line 16 can be affected by the test case, because a wrong number (e.g., −32) in the formula is subtracted from the sum of local variables.

5.1. Similar Buggy Detection

To detect a similar buggy code, we utilized a tool named CCFinder [23]. With the CCFinder, we can analyze a token-based source code to find the similar fixed files from the new given buggy file. Thus, we adopted these similar files to retrieve the fixed buggy information for constructing the initial population. If we use more similar fix information, we can get a variety of information. However, this results in a rapid increase in overhead; hence, we adopted one of the most similar buggy files from the result. In this case, we get a similar value (0.246377) from the given buggy code (e.g., File ID: 98d873cde . . .) and find the similar fixed buggy code (e.g., File ID: 36d8008b1 . . .), as shown in Figure 5.

```
1 /**/
2 #include <stdio.h>
3 #include <math.h>
4 //**/
5 int main() {
6 //
7 int sum;
8 char next;
9 sum=0;
10 printf("Enter an abitrarily long string, ending with carriage return > ");
11 while (next != '\n'){
12 scanf("%c", &next);
13 sum = sum + next;
14 //
15 //
16 }
17 //
18 sum=sum%64+22;
19 //
20 printf("Check sum is %c\n", sum);
21
22 return 0;
23 }
```

Figure 5. A sample of similar fixed buggy code.

According to Figure 5, we can verify the similarity with Figure 4 by the following: (1) applying *while* loop statement; (2) the *while* loop statement contains a *getchar* function related to the *scanf* function and an accumulation equation; (3) in the *sum* formula, they utilize an *add* (e.g., "+") operator and a *module* (e.g., "%") operator.

5.2. AST Conversion

To apply GP, we transformed source codes into the ASTs as shown in Figure 6 (from Figure 4) and Figure 7 (from Figure 5).

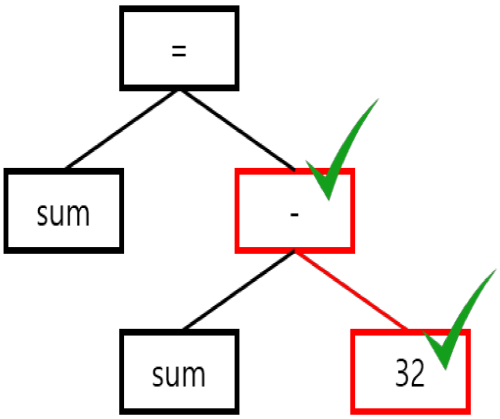

Figure 6. Result of AST Conversion from Figure 4.

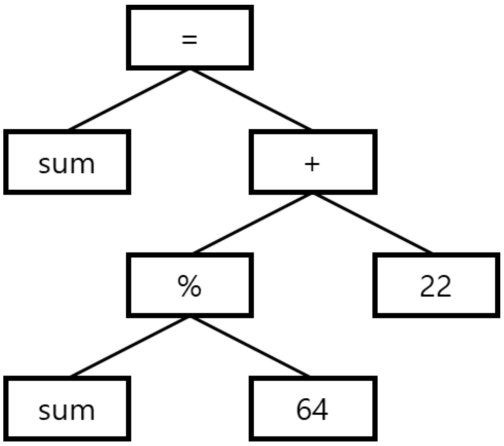

Figure 7. Result of AST Conversion from Figure 5.

In Figure 6, the AST is related to the buggy line (e.g., Line 18) in Figure 4. We show the buggy node of the tree using a red box and a check mark in the figure. When we transformed the source codes into AST, we utilized a mapping concept (e.g., line numbering) between the buggy nodes of AST in Figure 6 and the source codes of related nodes in Figure 7 in order to verify the target node in the GP operation. Then, we show the AST result of the similar fixed buggy code in Figure 7. The nodes of tree are related to the line (e.g., Line 19) in Figure 5.

5.3. Applying GP

By applying GP to the previous tree, we can get a result as shown in Figure 8. Each operation was executed by the probability from the previous operation. We can verify that the target nodes were changed by GP in Figure 8.

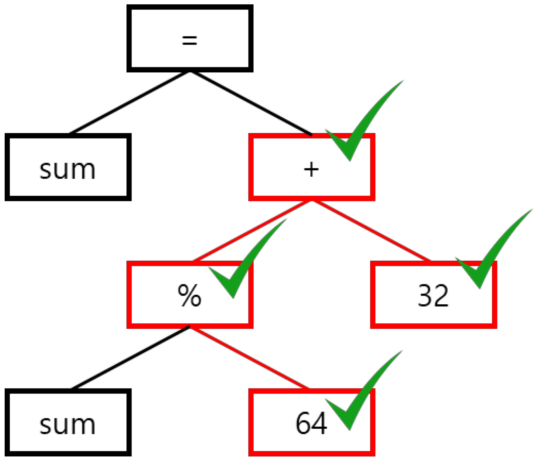

Figure 8. The AST Result from the GP.

The buggy code does not contain an *add* (e.g.,"'+") operator and a *module* (e.g., "%") operator. If we do not use the similar fixed code, the program cannot be fixed. However, we have to consider the same type of variable in the similar fixed code and the original buggy code when we perform the crossover operation. Finally, we can get a result from Figure 8, as shown in Figure 9.

```
1 //
2 #include <stdio.h>
3 #include <math.h>
4 int main()
5 {
6 int add, sum;
7 sum = 0;
8 printf("Enter an abitrarily long string, ending with carriage return > ");
9 while(add != '\n')
10 {
11 add = getchar();
12 if(add != '\n')
13 sum = add + sum;
14 }
15 //
16 sum = sum % 64 + 32 ;
17 printf("Check sum is ");
18 putchar(sum);
19 printf("\n");
20 return 0;
21 }
```

Figure 9. A result of patched source code from Figure 8 (e.g., the fitness value is 1.0).

5.4. Fitness Function Computation

Owing to the random generation characteristic of GP, we have to verify whether the patch can be adopted by using a fitness function, because the GP operations perform tasks including random combination, insertion, deletion, and edition. We computed the fitness function using test cases. After patch generation, we executed the patch program by compiling the patch source code. Then, we inserted the related test case into the running program. Finally, we observed the results. By using the fitness function, we verified the total number of passed test cases. If the result of the fitness function is 1.0, we consider the patch as successfully generated. However, if the result of one generation is not 1.0, we removed the patch that holds the lowest fitness function value according to the parameters of the parent population. Then, we applied the GP operations to the remaining patches to generate a new patch. The new patch was also computed by the fitness function. In this step, we expect the fitness value of the patch population to increase during the iteration of these steps. The patch generation

for the buggy program will not be executed (e.g., patch fail) if the patch generation is larger than the generation parameter, and if the generation time reaches the timeout parameter.

5.5. Adjusting Mutation Parameter

In this section, we performed experiments by adjusting mutation parameters (0.3, 0.5, and 0.7). This experiment was executed on a server machine with Xeon CPUs (10 cores, 2.40 GHz) and 256 GB RAM. Due to a time limit (e.g., a random generation characteristic of GP), we set the time limit for 8 h, which means the corrected patch generation should be made within 8 h. If not, the patch is fail. The results showed that the corrected patch was made within 8 h when we set the mutation parameter to 0.5. Also, we keep the sub-mutation parameters (e.g., insertion, deletion, swapping, and modifying) to default (each for 0.25) in all the cases. In addition, if we have a larger population size than 100, we can attain plentiful information. However, some of them might not be useful. Moreover, we cannot generate the corrected patches when we adjust the mutation parameters (0.3 and 0.7). In the future, we will investigate the correlation between the GP parameter and population size.

6. Discussion

6.1. Experiment Analysis

In the case study, we presented an approach for program fault repair using similar bug fix information on GP. Then, we generated the patch for a buggy program in IntroClass. In Figure 4, we added a formula (e.g., $sum = sum - 32$) so that the program returns a value of " " (e.g., *whitespace*) when we take the #1 black box test case (e.g., *1234567890*). However, our patch returned "-" (e.g., hyphen) with the same test case; hence, we generated the correct patch in Figure 9. In this case, we accept all the black box test cases.

6.2. Threats to Validity

Fault Localization: We investigated buggy code lines manually in IntroClass. The size of each project is small, so we can trace the buggy code manually using black box and white box test cases. In the future, we would like to adopt an automatic fault localization technique or tool to increase the accuracy of this study in large-scale projects.

Dataset: We utilized a benchmark dataset called IntroClass. However, the dataset was created by students in the class. Thus, the size and complexity of the code are small, and the code contains small keywords for programming. In the future, we will utilize an open source code to verify our study of effectiveness.

Adjusting Mutation Parameter: In this paper, we performed a case of study by adjusting GP parameters (mutation parameters = 3, 5, and 7). The result shows that the corrected patch was made when we set the mutation parameter to 0.5 (population size = 100). However, we cannot conclude that the parameters are always appropriate. In the future, we would like to investigate the correlation between GP parameters and others.

7. Conclusions

This study proposed a method to fix a buggy program automatically. First, we found suspicious buggy code lines by manual checking. Then, we generated candidate patches by applying GP with similar bug fix information. Next, we verified whether the candidate patches were adoptable. Finally, we generated the patch to fix the program fault. With our approach, we expect that the time and effort spent on fixing bugs can be reduced. In the future, we would like to utilize a test case prioritization algorithm [16] and large-scale bug fix information. Moreover, we plan to create a tool for automatic fault repair.

Acknowledgments: This research was supported by Next-Generation Information Computing Development Program (NRF-2014M3C4A7030504) and by Basic Science Research Program (NRF-2017R1A2B4009937) through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT, and Future Planning.

Author Contributions: Geunseok Yang conceived the research subject and background and wrote the paper. Youngjun Jeong and Kyeongsic Min performed the experiments. Jung-won Lee contributed to the discussion and analysis of the results. Byungjeong Lee supervised the paperwork.

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

References

1. Yang, G.; Baek, S.; Lee, J.W.; Lee, B. Analyzing Emotion Words to Predict Severity of Software Bugs: A Case Study of Open Source Projects. In Proceedings of the Symposium on Applied Computing (SAC 2017), Marrakech, Morocco, 3–7 April 2017; pp. 1280–1287.
2. Zimmermann, T.; Premraj, R.; Bettenburg, N.; Just, S.; Schroter, A.; Weiss, C. What Makes A Good Bug Report? *IEEE Trans. Softw. Eng.* **2010**, *36*, 618–643. [CrossRef]
3. Le Goues, C.; Nguyen, T.; Forrest, S.; Weimer, W. Genprog: A Generic Method for Automatic Software Repair. *IEEE Trans. Softw. Eng.* **2012**, *38*, 54–72. [CrossRef]
4. Kim, D.; Nam, J.; Song, J.; Kim, S. Automatic Patch Generation Learned from Human-written Patches. In Proceedings of the International Conference on Software Engineering (ICSE 2013), San Francisco, CA, USA, 18–26 May 2013; pp. 802–811.
5. Qi, Y.; Mao, X.; Lei, Y.; Dai, Z.; Wang, C. The Strength of Random Search on Automated Program Repair. In Proceedings of the International Conference on Software Engineering (ICSE 2014), Hyderabad, India, 31 May–7 June 2014; pp. 254–265.
6. Weimer, W.; Fry, Z.P.; Forrest, S. Leveraging Program Equivalence for Adaptive Program Repair: Models and First Results. In Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE 2013), Silicon Valley, CA, USA, 11–15 November 2013; pp. 356–366.
7. Samimi, H.; Schafer, M.; Artzi, S.; Millstein, T.; Tip, F.; Hendren, L. Automated Repair of HTML Generation Errors in PHP Applications using String Constraint Solving. In Proceedings of the International Conference on Software Engineering (ICSE 2012), Zurich, Switzerland, 2–9 June 2012; pp. 277–287.
8. Son, S.; McKinley, K.S.; Shmatikov, V. Fix Me Up: Repairing Access-Control Bugs in Web Applications. Available online: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.307.1928 (accessed on 18 February 2018).
9. Long, F.; Rinard, M. Automatic Patch Generation by Learning Correct Code. *ACM SIGPLAN Not.* **2016**, *51*, 298–312. [CrossRef]
10. Long, F.; Rinard, M. Staged Program Repair with Condition Synthesis. In Proceedings of the Joint Meeting on Foundations of Software Engineering (FSE 2015), Bergamo, Italy, 30 August–4 September 2015; pp. 166–178.
11. Le Goues, C.; Holtschulte, N.; Smith, E.K.; Brun, Y.; Devanbu, P.; Forrest, S.; Weimer, W. The ManyBugs and IntroClass Benchmarks for Automated Repair of C Programs. *IEEE Trans. Softw. Eng.* **2015**, *41*, 1236–1256. [CrossRef]
12. Forrest, S. Genetic Algorithms: Principles of Natural Selection Applied to Computation. *Science* **1993**, *261*, 872–878. [CrossRef] [PubMed]
13. Le Goues, C.; 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 International Conference on Software Engineering (ICSE 2012), Zurich, Switzerland, 2–9 June 2012; pp. 3–13.
14. Nguyen, H.D.T.; Qi, D.; Roychoudhury, A.; Chandra, S. Semfix: Program Repair via Semantic Analysis. In Proceedings of the International Conference on Software Engineering (ICSE 2013), San Francisco, CA, USA, 18–26 May 2013; pp. 772–781.
15. 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 ACM Symposium on Applied Computing (SAC 2016), Pisa, Italy, 4–8 April 2016; pp. 1364–1370.
16. Qi, Y.; Mao, X.; Lei, Y. Efficient Automated Program Repair through Fault-recorded Testing Prioritization. In Proceedings of the International Conference on Software Maintenance (ICSM 2013), Eindhoven, The Netherlands, 22–28 September 2013; pp. 180–189.

17. Guo, S.; Chen, R.; Li, H. Using Knowledge Transfer and Rough Set to Predict the Severity of Android Test Reports via Text Mining. *Symmetry* **2017**, *9*, 161. [CrossRef]
18. Singh, J.; Singh, G.; Singh, R. Optimization of Sentiment Analysis using Machine Learning Classifiers. *Human Centric Comp. Inf. Sci.* **2017**, *7*, 32. [CrossRef]
19. Souri, A.; Hosseini, R. A State-of-the-art Survey of Malware Detection Approaches using Data Mining Techniques. *Human Centric Comp. Inf. Sci.* **2018**, *8*, 3. [CrossRef]
20. Sabharwal, S.; Aggarwal, M. Test Set Generation for Pairwise Testing using Genetic Algorithms. *J. Inf. Proc. Syst.* **2017**, *13*, 1089–1102.
21. Youm, K.C.; Ahn, J.; Lee, E. Improved Bug Localization based on Code Change Histories and Bug Reports. *Inf. Softw. Technol.* **2017**, *82*, 177–192. [CrossRef]
22. Lukins, S.K.; Kraft, N.A.; Etzkorn, L.H. Bug Localization using Latent Dirichlet Allocation. *Inf. Softw. Technol.* **2010**, *52*, 972–990. [CrossRef]
23. Kamiya, T.; Kusumoto, S.; Inoue, K. CCFinder: A Multilinguistic Token-based Code Clone Detection System for Large Scale Source Code. *IEEE Trans. Soft. Eng.* **2002**, *28*, 654–670. [CrossRef]
24. Nakatsu, N.; Kambayashi, Y.; Yajima, S. A Longest Common Subsequence Algorithm Suitable for Similar Text Strings. *Acta Inf.* **1982**, *18*, 171–179. [CrossRef]