Using the Deep Learning-Based Approaches for Program Debugging and Repair

Tzu-Yang Lin, Chin-Yu Huang*, and Chih-Chiang Fang
Department of Computer Science, National Tsing Hua University, Hsinchu, Taiwan
jesssorry@gmail.com, cyhuang@cs.nthu.edu.tw,neilfang112113@gmail.com
*corresponding author

Abstract—Maintenance is the most labor-intensive stage in the Software System Development Life Cycle (SDLC). When an error is detected, an Automatic Program Repair (APR) system can locate the error by utilizing fault prediction and then employ search-based or semantic-based repair techniques to attempt to fix it. Finally, the program repair system runs test cases to verify that the errors have been properly repaired and the program's functionality has not been compromised. In this study, we present a novel approach called Deep Learning Based GenProg (DLBGP), which combines deep learning with APR. Our proposed method involves extracting node vectors from the program's Abstract Syntax Tree (AST) and identifying semantic features using a Deep Brief Network (DBN). These semantic features are then used to predict the accuracy of the repair candidates. In experiments conducted on ten subject files in the IntroClass database, our proposed DLBGP model reduced execution time by 64% compared to GenProg while achieving the same number of successful test cases. In summary, we offer a fast, efficient, and easily deployable deep learning program repair method that could prove beneficial to the field of APR.

Keywords—automatic program repair; search-based repair; genetic programming; deep learning; deep brief network

1. Introduction

The information system is a crucial component in many modern companies, directly impacting their business operations. However, even major systems like Hotmail, Office 365, Gmail, and Amazon cloud have experienced serious errors, resulting in service interruptions ranging from minutes to hours, and in some cases, loss of customer data [1]-[3]. Thus, it is imperative for the industry to provide stable and high-quality information systems. Software Development Life Cycle (SDLC) is a common method for deploying information systems, consisting of stages such as requirement definition, system analysis, system design, system development, system testing, and system maintenance. Software maintenance is known to consume significant resources and time [4], with 90% of the software project cost attributed to this phase [5]. In the US alone, software maintenance costs can reach up to 70 billion annually [6][7], highlighting the need to reduce software maintenance costs as a critical issue in SDLC. Automatic Program Repair (APR) is a superior approach for software maintenance due to its efficient resource utilization and ability to ensure stable repair quality. The process of validating the fitness of repair candidates is a critical step in APR, and the most commonly used method is to utilize test cases to verify their performance. This approach, known as test-suite based repair [8], forms the foundation of our study, where candidates that successfully pass more test cases are retained while those that fail to meet the required standards are eliminated.

The main problem with test-suite based repair is time-consuming task. For example, the state-of-the-art APR technology GenProg [9] spent 62.75% of the execution time on testing on average for 16 target programs. For some target programs, it even spent over 90% of the execution time on testing. With the increasing scale of the program and the increasing number of test cases, testing may consume large amounts of the execution time. To solve this problem, we tried to use deep learning to accelerate testing and to reduce the whole execution time of the APR. We regarded deep learning as a solution because it is widely used in classifying large amounts of data, and it has achieved many impressive successes in lots of studies. Deep learning currently focuses on several aspects in the field of software testing: (1) test cases generation [10], (2) test suite reduction [11]-[13], (3) test suite optimization [14][15], and (4) test case execution scheduling [16]. Based on previous research, we propose Deep Learning Based GenProg (DLBGP) as our solution. Genetic programming was used as foundation of our test-suite based program repair system, and we used a deep learning model named Deep Brief Network (DBN) to predict the fitness of repair candidates. If fitness is smaller than the threshold that is determined by empirical statistics, our method regards this repair candidate as defective. This defective candidate would not execute test cases and it would not be retained to the next generation. That is the reason why our method has less execution time.

We use published IntroClass [17] based on ten subject files to validate our method. The experiment results have shown that our method spent 212.9003 seconds of repair time totally, and GenProg spent 597.9942 seconds of repair time totally. Our method reduced 64% of the execution time while passing the same number of test cases as traditional methods did. It is obvious that our method can reduce lots of time and has a better performance. The major contributions of this paper are listed below: (1) it combines deep learning technology with APR based on genetic programming, (2) the proposed method DLBGP can significantly reduce execution time compared to traditional search-based repair methods

2767-6684/23/$31.00 ©2023 IEEE
DOI 10.1109/DSA59317.2023.00059

while maintaining the similar repair quality. Avoid re-running test cases for each variant, and (3) it conducted sensitivity analysis on some genetic programming parameters and recorded the related change of modified repair time, respectively. The rest of the paper is organized as follows. Firstly, we briefly review the related process of APR in Section 2. In Section 3, we describe both the overall information and the detailed design and the implementation of our method. Some experimental results and discussions are presented in Section 4. Finally, some conclusions and future works are described in Section 5.

2. Related Work

Basically, bug fixing is a major part of the software maintenance phase. The following will focus on the current repair methods: 1) semantic-based repair, and 2) search-based repair, respectively.

2.1. Semantic-based repair

Semantic-based repair focuses on solving the function in the program. Figure 1 illustrates symbolic execution operation and constraint solving is shown in Figure 2 to analyze code and figure out the execution paths in all conditions. There are some related researches. For example, Nguyen et al. [18] presented SemFix, which solves the requirement to pass a given test suite as a constraint. Xuan et al. [19] proposed Nopol, which takes a buggy program with a test suite as input and generates a patch with a conditional expression as output. Mechtaev et al. [20] presented DirectFix, which generates the simplest patch such that the program structure of the buggy program is maximally preserved. Mechtaev et al. [21] presented Angelix, which is more scalable than the past methods.

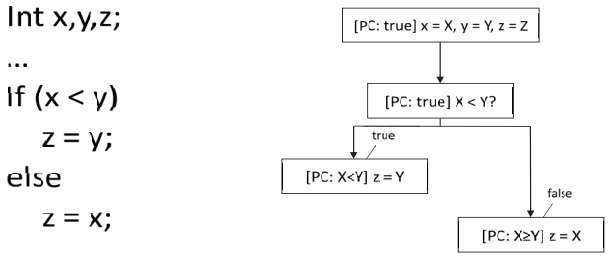

Figure 1. Symbolic execution

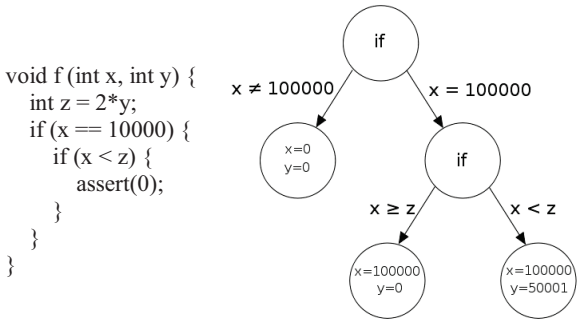

Figure 2. Constraint solving

2.2. Search-based repair

It is also known as the iterative generate-and-validate technique that is shown in Figure 3. The overall process can be divided into four steps:

- *Analyze the program to be repaired*

Convert the programming language into another format that is easy to analyze. For example, the abstract syntax tree (AST), which is now widely used in the compiler field, is a well-known representation [22]. Many automatic repair tools use AST to record the program.

- *Produce repair candidates*

This step tries to fix the error and results in many repair candidates. Some literatures indicate that certain specific errors can be fixed by copying and arranging the existing source codes [23]-[26]. Such a technique is known as redundancy-based repair. The method of arranging and combining is often done using genetic algorithms. Common operations include insertion, deletion, and exchange. There are also advanced discussions on the operation of mutation in recent literature [27].

- *Search for candidates*

This step searches for the right candidates. Most of the current repair methods randomly pick up candidates from the pool and evaluate their fitness. Some literatures attempt to rank candidates in order to shorten the search time [28][29].

- *Evaluate candidate fitness*

This step considers whether the repair candidate can correct the error. The most common method is executing test cases to check whether the program achieves the needed functions. If some test cases fail, some unrepaired errors exist in the program. On the other hand, if all test cases are executed correctly, the program is repaired with no error. The above process is called test-suite based repair [8]. This methodology seems to be great, but there are three major problems: (1) test case generation is not easy, (2) testing is time consuming, and (3) testing may produce an overfitted repair.

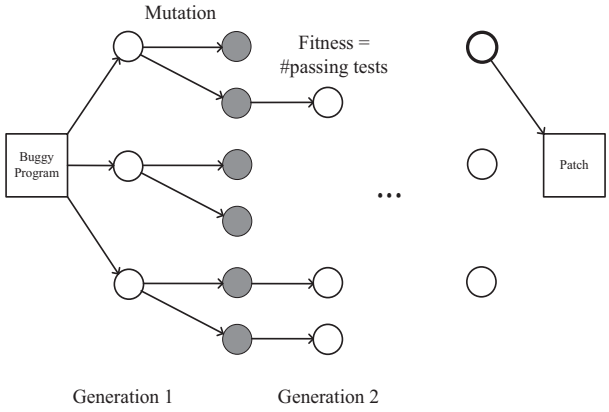

Figure 3. Generate-and-validate technique

There are some famous repair tools which use the search-based method. For example, Geoues et al. [9] proposed GenProg, which extends genetic programming to evolve a program variant and retains the required functionality but it is not susceptible to a given defect. Kim et al. [30] proposed PAR, which uses fix patterns learned from existing human-written patches to generate program patches automatically. Long et al. [31] proposed SPR, which combines the staged program repair and the condition synthesis together.

There are still some related researches on APR and software reliability. Lin and Huang [32] proposed a framework which described possible debugging behavior using queueing theory analysis during software development process. Wong et al. [33] designed a solution by employing the source code level technologies to debugging software designs represented in a high-level specification and description language such as SDL. You et al. [34] proposed a methodology of modifying similarity coefficients on spectrum-based fault localization (SBFL). This method can effectively identify the faults. Wong et al. [35]-[37] investigated different kinds of methods in fault localization and discussed key issues and concerns so that the related researchers can understand the trends and directions. Dutta et al. [38] presented an ensemble classifier based on fault localization to effectively identify the common and intrinsic faults. Li and Liu [39] further analyzed five distance metrics impact and evaluated five SBFL techniques in multiple fault localization. Their findings are obtained list of different criteria and a list of factors to evaluate the fault-focused clustering and SBFL performance. Lee et al. [40] proposed a fault-based genetic-like programming approach that heuristically searches all possible variants. This method can find the best repairs with fewer operations and less time than the previous genetic programming did.

Although there are many related researches about APR, all of them suffered from a long execution time, especially in the testing time. In actuality, we observed these methods and noticed that testing is the process of classifying repair candidates. This is similar to the situation in which it is proper to use deep learning. Deep learning techniques are used in variety of fields, and some studies have begun to apply this emerging technology to the field of software repair. Yokoyama et al. [28] applied deep learning to the sorting of repair ingredients. It is a quicker way to find suitable repair candidates than the traditional methods do. These studies not only allow us to see the potential of deep learning but also to motivate our research.

3. Deep Learning Based GenProg

3.1. Overview of deep learning based GenProg

In this study, we use DBN to determine if the program is wrong, and the result is going to help the program repair system to decrease execution time. In the beginning, we used a tool to parse and analysis source codes, which contains the correct program and the wrong program. Since DBN needs the integer vector as input data, we build a mapping between the analysis result, which are the AST nodes, and the integer. After preparing the integer vector, this data is used to train a DBN. Then, our method is able to recognize whether there is any defect in a program. The architecture of our method can be shown in Figure 4.

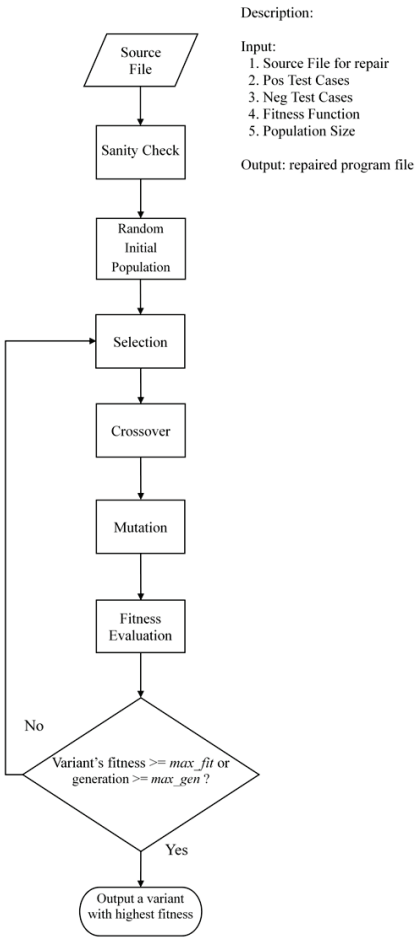

Figure 4. Work flow of DLBGP

Our method consists of several major steps: (1) sanity checking, (2) initializing population, (3) selecting candidates, (4) crossover, (5) mutation, (6) fitness evaluation, and (7) repeating above (3)-(6) steps until the termination condition is achieved. Now, we combine the model to the search-based program repair system. The system will parse the original program and generate lots of candidates in the first generation. Our method has to complete this real-time operation, which includes parsing candidates, encoding token vectors to integer vectors, interpreting vectors, and returning its prediction to the program repair system. If the model has a low confidence for this candidate, the program repair system will use test cases to check the correction of it, which is same as the traditional way. However, if the model has a high confidence for this candidate, the program repair system will adopt the model's suggestion and decrease the procedure of validation. This proposed framework can skip redundant testing and improve the efficiency of repair.

3.2. Sanity Check

The input data required by our method contains the source code file and a suite of test cases (Negative/Positive). Negative test cases can indicate errors in the program, in contrast to positive test cases can verify the function of the program. The first step in our program repair model is to perform a sanity check, which is to check whether the test case execution result is consistent with the tester's result. If the execution results are consistent, the program repair system will record the execution paths from positive test cases and negative test cases. These paths will be used during the subsequent repair process. If the execution results are inconsistent and the functions of the program are not clearly reflected. The program repair system will be terminated.

3.3. Program Representation

Our method employs a data-based representation for each variant (i.e., candidate program), comprising of: (1) an AST representation that encompasses each statement within program, and (2) Constructing a weighted path includes a sequence list of statements that are executed by this test case. The weighted path denotes a sequence of pairs <statement, weight> that can facilitate the program repair system in applying mutation operators to a smaller and more appropriate subset of the program tree. This unique representation enables us to address scaling challenges in the domain of genetic programming, allowing for the application of our method to larger programs. Moreover, our approach does not permit genetic operators to generate new statements; otherwise, statements from other parts of the program tree are borrowed. Statements that are not included in the weighted path remain unaltered, though they may be duplicated to the weighted path by the mutation operator. Each generated variant retains the same number of pairs and sequences in the weighted path.

Constructing a weighted path utilizes a transformation that each statement is to assign a unique number and record each statement visited during execution. Redundant statements in the list are removed and frequently accessed statements (e.g., for loop) are not suitable for modification. We respect the order of statements, determined when they are first accessed, resulting in a weighted path that is a sequence rather than a set. Each statement is considered a candidate for repair when executed in a negative test case, with an initial weight of 0.65. The weights of all other statements are set to 0.0 and remain unchanged. Statements in negative test cases are adjusted if they were executed in positive test cases. The objective is to prioritize statements that may cause undesirable behavior, while avoiding modification of statements that impact desirable behavior.

3.4. Selecting Candidates

Our method picks individual variants and copies them to the next generation. There are many possible selection algorithms [41][42] that have same characteristic: a suitable individual will be more distributed in the next generation than an unsuitable individual does. First, we discard individuals with a fitness of 0 (fitness=0 means the individual cannot be compiled or cannot pass any test cases) and use other individuals whose fitness is more than 0 to replace the remaining positions. Then we use tournament selection [41] to select the *pop_size*/2 number of members from the previous generation to a new generation.

3.5. Crossover

The crossover operation re-combines two variants to generate a new descendant that amalgamates information from both parents. Each variant can only serve as a parent once at most in the crossover operation within a given generation. We select a cutoff point on the path and exchange all the statements after the cutoff point. The detailed crossover operation is illustrated in Figure 5. Given that input variants A=[a1, a2, a3, a4, a5, a6, a7] and B=[b1, b2, b3, b4, b5, b6, b7], with a cutoff point set at 3, the resulting next generation variants would be C=[a1, a2, a3, b4, b5, b6, b7] and D=[b1, b2, b3, a4, a5, a6, a7]. In our method, only the statements on the weighted path undergo crossover. Although there are various types of crossover operators, literature suggests that their outcomes are comparable to those of single-point crossover [9].

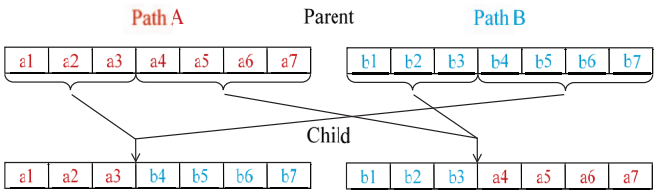

Figure 5. Crossover Operation

3.6. Mutation

Mutation has a low probability of modifying any given statement on the weighted path, and any changes made to the statements will be reflected in its AST. The likelihood of mutation for each statement is proportional to the weight of the path. Specifically, a statement that appears only in negative test cases has first choice considered for mutation, while a statement that appears in both positive and negative test cases has a lower chance of being mutated. The maximum probability of mutation for each statement is determined by the global mutation rate.

In general, there are two ways for traditional mutation operations such as single bit flips and simple symbolic substitutions. Here, two ways are not suitable to our situation. Our basic element is a statement and mutation operators are also complex, as illustrated in Figure 6. It includes insertion (where another statement is inserted after the current position), deletion (where the entire statement is removed), or replacement (where one statement is replaced by another). The probabilities of these mutation operations are determined based on empirical investigations or statistics on types of fault behavior. Specifically, we set the probabilities of insertion, deletion, and replacement to 0.24, 0.4, and 0.36, respectively.

In the insertion operation, we convert *Stmt1* into a statement with Stmt1 attached to *Stmt2*. In the deletion operation, we convert *Stmt1* into a blank statement instead of deleting it directly, in order to maintain a consistent path length for each variant. This allows a deleted statement to still be used in subsequent mutation operations. In the replacement operation, Stmt1 is replaced by *Stmt2*. For insertion and replacement, the second statement (*Stmt2*) is randomly chosen from anywhere in the program, not just from the weighted path. The weight of the statement does not affect the probability of it being selected as a candidate repair, as our intuition is that the repair candidate may be located anywhere in the program, not necessarily on the negative path. It should be noted that in some cases, a mutated variant may fail to compile successfully. In such cases, that variant will not be used further in the process.

3.7. Fitness Evaluation

The objective of fitness evaluation is to estimate the quality of repair candidates. Traditionally, executing test cases has been a common method for this purpose, where repair candidates that pass more test cases are considered higher priority. However, testing can be time-consuming, which is why we incorporate deep learning into our fitness evaluation process to address this issue. Our method can be divided into four parts: (1) source code analysis, (2) encoding token vectors to integer identifiers, (3) training a deep learning model, and (4) using our method to predict defects. The detailed flow chart of our fitness evaluation system is depicted in Figure 7.

Figure 6. Mutation flow chart

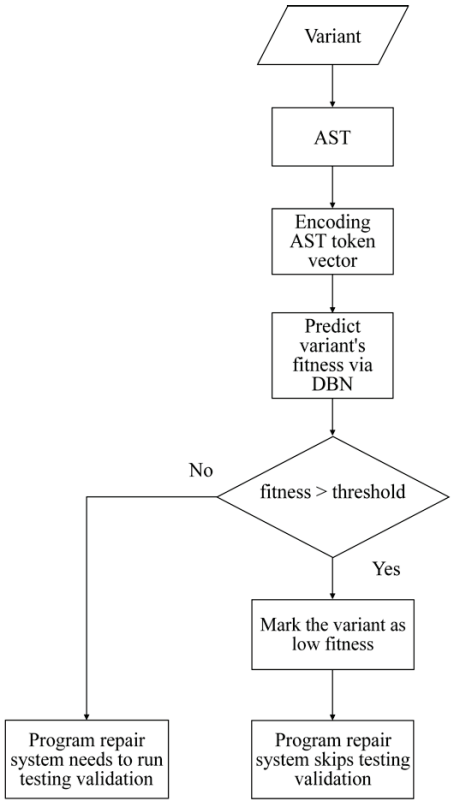

Figure 7. Work flow of our fitness evaluation system

- *Source code analysis*

In our study, it is necessary to figure out semantic and structural information from source code in order to train our deep learning model. One question we need to answer is what granularity of representation is appropriate for each source code to be represented as a vector. Common options for granularity include character-level, token-level, and nodes on Abstract Syntax Trees (ASTs), among others. Research has shown that using nodes on ASTs as the granularity for representation is suitable for building program representations [43]. ASTs are commonly used in the field of compilers and program repair as they retain both semantic and structural information of programs. ASTs serve as a structured representation of the code, capturing the syntactic structure, relationships between statements, and their corresponding semantics, making them a valuable tool for various program analysis and manipulation tasks. [44][45]. The above mentioned rationale is why we have opted for ASTs as our chosen representation for programs.

We utilize C Intermediate Language (CIL) [46] to parse and extract information from the C code. CIL is capable of performing various simplifying transformations to the C AST, making it highly suitable for rapid prototyping of new static and dynamic analyses, as well as designing and experimenting with new language extensions, as per the state-of-the-art method [47], we exclusively utilize three types of nodes: (1) method invocation and class instance creation nodes, (2) declaration nodes encompassing method declarations, data declarations, type declarations, and enum declarations, and (3) control-flow nodes including if statements, switch statements, and loop statements, among others. Nodes that are not mentioned above are not preserved as they may not be conducive to cross-project prediction. As function, class, and type names are often project-specific, we rely on their AST node types such as method declarations and method invocations instead of their names.

- *Encoding token vectors to integer identifiers*

Due to the requirement of a digital vector as input in DBN, the token vector cannot be directly fed into the DBN. Therefore, we must transform the token vector into an integer vector. Each token is a unique integer identifier that its range is between 1 and the maximum number of tokens. Due to our method needs to consider cross-project defect prediction (CPDP) architecture, we encode the type of token rather than the method name or class name of the token. Moreover, the converted integer vectors may have varying lengths, which is not compatible with DBN's requirement of fixed-length input vectors. To reconcile this, we append 0 to the integer vector so that each vector is of the same length as the longest vector. The value 0 does not hold any meaning in our case as we start encoding from 1, and this conversion is a straightforward approach to make the vector compatible with DBN. It is noted that the sequence of tokens still retains unchanged and the structural information of the program is preserved.

- *Training deep learning model*

We use the powerful capabilities of DBN to generate features and capture code semantic information and structural information. Based on the above information, DBN will make a defect evaluation of the program. We train our DBN's weight and biases through training data, and DBN will get the features which are difficult to be observed but can distinguish semantic differences. DBN learns the probability that each node traverses to the previous level. With backpropagation validation, DBN automatically adjusts the weights of nodes in different layers to reconstruct the input data used to generate features.

Considering that this study only uses DBN as a pure application, we use a standard architecture DBN instead of a sophisticated and complex experimental architecture [48][49]. In this paragraph, the parameters of our DBN will be described. For simplifying network architecture, the number of nodes in each layer will be set to the same. In particular, our DBN is a binary classifier that contains one input layer, three number of hidden layers, and one output layer. Each hidden layer consists of 100 neurons, and one neuron for output layer. All nodes on the previous layer will be connected to all nodes on the next layer by weighted edges. The higher the value of weight, the greater the influence of the upper node on the next layer of nodes. We use the sigmoid activation function in output layer. On the contrary, all other layers use ReLU activation functions. Binary_Crossentropy is used as the loss function of binary classification and Adam optimizer is selected. The epoch number is 50. The batch_size is 100, which is the same as the input layer's length. The probability of dropping out is 0, and the rate of validation is 0.2. The scale of our training data is 336000 tokens, which are extracted from the databases described in Section 4.1. Our method is implemented by Keras API. Keras API is high-level deep learning library written in Python programming language that runs on TensorFlow framework. Keras API is widely used in many related research [50] and the neural networks with these hidden layers and nodes can be built easily.

DBN requires input data with values ranging from 0 to 1, whereas our input vector contains integer values based on our mapping approach. To get better performance and meet the need of DBN, we separately employ min-max normalization in both training data and testing data vectors. After normalization, the normalized data can still effectively distinguish between different nodes as the same identifiers retain the same values.

- *Using our DBN to predict defects*

After calculation, our deep learning model will output a score, which represents the possibility that the individual contains defects. If this score exceeds our default threshold, our fitness evaluation system determines that the individual is the wrong program and sends a positive predictive result to our program repair system. On the other hand, if the score

does not exceed the threshold, our fitness evaluation system cannot confirm whether the individual is correct or not. It will send the negative predictive result to our program repair system. Our program repair system will make different treatments based on different predictive results. If the predictive result is positive, the program repair system will set the individual's fitness to 0, indicating that the individual is wrong. If the predictive result is negative, the program repair system will execute test cases for the individual, and use the fitness function to evaluate the fitness of the individual based on the test result.

Table 1 is the performance of our defect prediction model. According to this table, the value of recall in the negative scenario is the best. Therefore, we decide to use this model to filter out negative candidate repairs. Only positive candidate repairs need to execute test cases. Although this model is simple, we will still get much benefit from this architecture and we leave the defect prediction model as a future work.

Table 1. Performance of our defect prediction model

	Precision	Recall	F1-score
Positive Scenario	0.65	0.29	0.38
Negative Scenario	0.53	0.77	0.62
Avg/Total	0.54	0.53	0.51

4. EXPERIMENTS

In this section, we used DLBGP to repair errors in the defect program database. Our hardware platform is Intel 2.6GHz i7-6700HQ machine with 16GB RAM.

4.1. Experimental Setup

- *Programs and defects*

Two common databases are used in this study to train the deep learning model and evaluate program repair system performance are IntroClass [17] and ITSP [51]. The first IntroClass benchmark has totally six subject C programs and each one is written by novices. These programs are from programming assignments in an introductory, undergraduate C programming course. It consists of 259 repositories, 587 C files, and 13470 non-comment line of codes. Here, the authors want the IntroClass benchmark to be representative of both the type and frequency of the new developers' mistakes, they don't remove duplicate errors from their datasets.

Table 2 is the overall information of the IntroClass benchmark and the description of the subject programs. ITSP [51], an intelligent tutoring system for programming, is a system whose goal is to identify errors in student's programs and to provide appropriate feedback to students to help them fix their programs. Its dataset is obtained from an introductory C programming course (CA-101) offered by Amey Karkare who serves at Indian Institute of Technology Kanpur (IIT-K). The feature of this benchmark is that their programs are often severely incorrect. 60% of the programs in this benchmark fail more than half of the available test cases, and half of the programs require heavy modification to correct the errors. It consists of 661 repositories, 1395 C files, and 41878 non-comment line of codes. For the purpose of representation of both the type and frequency of novices' mistakes, this dataset keeps the duplicate errors from student programs. Table 3 is the overall information and description about the second ITSP dataset. P1 indicates weekly programming assignments termed Lab3, P2 indicates Lab4, P3 indicates Lab5, and P10 indicates Lab12 in the database, respectively. Table 4 lists some errors category about the ITSP dataset. This study compared the experimental results to two traditional repair methods. One method is GenProg [9], and another method is AE [52]. Both traditional methods are open-source so we can make sure that their performance is consistent as expected.

Table 2. The six IntroClass Benchmark subject programs

IntroClass Benchmark								
Program	**Repo**	**Files**	**Blank**	**Comment**	**Code**	**Defects**	**Unique Defects**	**Description**
Checksum	21	49	347	79	948	69	47	Compute a simple checksum of a string.
Digits	55	145	1186	352	4357	236	144	Compute the number of digits in an integer.
Grade	50	140	880	182	3080	268	136	Compute the letter grade corresponding to a percentage score.
Median	44	100	578	147	1975	232	98	Compute the median of three numbers.
Smallest	45	87	516	120	1686	177	84	Compute the smallest of three numbers.
Syllables	44	66	437	121	1424	161	63	Compute the number of English syllables in a string.
Total	259	587	3944	1001	13470	1143	572	Compute a simple checksum of a string.

Table 3. The ten ITSP Benchmark subject programs

ITSP Benchmark						
Program	**Repo**	**Files**	**Blank**	**Comment**	**Code**	**Topic**
P1	63	130	314	1553	1775	Simple Expressions, printf, scanf
P2	117	242	550	2443	5161	Conditionals
P3	82	172	635	4764	4181	Loops, Nested Loops
P4	79	166	759	8852	4719	Integer Arrays
P5	71	150	611	3400	5046	Character Arrays (Strings) and Functions
P6	33	71	551	2387	3550	Multi-dimensional Arrays (Matrices)

P7	48	104	523	31647	2761	Recursion
P8	53	114	664	3153	5098	Pointers
P9	55	118	458	6567	4334	Algorithms (sorting, permutations, puzzles)
P10	60	128	854	7833	5253	Structures (User-Defined data-types)
Total	661	1395	5919	72599	41878	Simple Expressions, printf, scanf

Table 4. Errors category of ITSP Benchmark subject programs

ITSP Benchmark	
Errors Category	**Frequency**
Wrong Conditions for Control-Flow	48
Missing Character	8
String Modifications	5
Array Accesses	4
User defined Functions	4
Missing Values in the Output	4
Library Functions	3
Others	3
Missing Whitespace in the Output	2
Floating Point Operations	1

- *Parameter of genetic programming*

We describe our genetic programming parameters set which works well in our experimental environment. The maximum number of generations is 10, which is smaller than the traditional genetic programming applications. We choose *pop_size* as 40 (also a small number). For pos-weight and neg-weight, we set it to 0.65 and 0.35 respectively. In related research, we know that it is possible to obtain more precise weights by the fitness distance correlation metric [53][54]. Because the simple parameters still performed well enough, we continued to use them and to leave the fitness distance correlation metric as a future work. We set 1 as the seed of the random number generator. This will make our experiment repeatable.

4.2. Repair Results

Table 5 shows the repair time in the IntroClass benchmark, and Table 6 is the repair time in the ITSP benchmark. We selected 10 subject files from 587 files in the IntroClass database and we selected 4 subject files from 1395 files in the ITSP dataset. We compared the execution time between AE [52], GenProg, and our method. The common parameters of the genetic programming in all program repair systems were the same, such as *pop_size*, *pos_weight*, and *neg_weight*. It should be noticed that the following comparisons are based on such a situation that a particular program repaired by our method is able to pass the same number of test cases as other methods. The result presents that our method has less execution time than GenProg in most of the subject files.

Besides, our method has less execution time than AE when it doesn't consider the 10th subject file in Table , which is unable to be repaired by AE. The improvement is not significant to the files whose execution time is little, because the overhead of utilizing the deep learning model to predict a fault may exceed the saving of testing time. However, the improvement is more significant to the files whose execution time is long. For example, our method took 144.231 seconds in the 10th file in Table 5. It reduces a 70% execution time compared to the traditional method. In the 1st subject file of Table 6, our method took 54.9693 seconds to repair the bugs. It reduces 24% repair time compared to GenProg and 50% repair time compared to AE, respectively. It should be noted that we used the same parameters of genetic programming for AE, GenProg, and our method, so we couldn't guarantee all program repair systems had their optimized performance. That may be the reason why AE failed to repair some subject files.

Table 5. Repair time of each repair systems in the IntroClass Benchmark

Subject files	GenProg (s)	AE (s)	Our method (s)
1	6.88042	5.45556	6.35275
2	6.05169	5.66346	2.31514
3	15.1142	19.8711	11.7605
4	17.9337	13.1493	8.36506
5	27.523	12.7939	15.9251
6	7.40185	2.36285	2.42728
7	7.65303	3.74622	3.22963
8	1.74478	2.54894	2.28522
9	27.0875	12.7178	16.0086
10	480.604	ends	144.231
Total	597.9942	78.30913	212.9003
Average	59.79942	8.701014	21.29003

Table 6. Repair time of each repair systems in the ITSP Benchmark

Subject files	GenProg (s)	AE (s)	Our method (s)
1	72.4508	109.392	54.9693
2	24.3634	fail	18.3466
3	3.99539	4.30927	5.26607
4	20.4068	1.89187	9.76669
Total	121.2164	115.5931	88.34866
Average	30.3041	38.53105	22.08717

Table 7 is the average CPU usage of 10 subject files in the IntroClass benchmark. According to this table, it is clear that our method has a less average CPU usage than both GenProg and AE do. For example, our method reduces 22% average CPU usage compared to both GenProg and AE for the first subject file. On average, our method reduces 12% average CPU usage compared to GenProg, and it reduces 11% average CPU usage compared to AE for 10 subject files, respectively. This result indicates that our method is able to reduce hardware resource cost in repair compared to GenProg and AE because of a lower average CPU usage.

Table 8 is the average CPU usage of 4 subject files in the ITSP benchmark. According to this table, it is clear that our method has less average CPU usage than both GenProg and AE do. For example, our method reduces 31% average CPU

usage compared to GenProg and 40% average CPU usage compared to AE for 2nd subject file. On average, our method reduces 21% average CPU usage compared to GenProg, and it reduces 23% average CPU usage compard to AE for 4 subject files, respectively. This result indicates that our method is able to reduce hardware resource cost in repair compared to both GenProg and AE because of a lower average CPU usage.

Table 7. Average CPU usage of IntroClass Benchmark

Subject files	GenProg (%)	AE (%)	Our method (%)
1	98.9911	98.6245	77.4131
2	98.8127	92.0419	86.8776
3	99.1214	98.7623	88.9947
4	99.643	98.9343	87.3346
5	98.9667	99.0151	88.592
6	98.2941	97.5107	79.8431
7	99.2769	97.8204	89.0364
8	97.4571	97.5871	91.3175
9	98.8774	98.9552	89.2382
10	99.8371	98.7902	96.0511
Average	98.92775	97.80417	87.46983

Table 8. Average CPU usage of ITSP Benchmark

Subject files	GenProg (%)	AE (%)	Our method (%)
1	98.3896	98.5263	94.3996
2	76.6799	89.1263	53.2712
3	94.2807	97.0273	63.1775
4	99.1853	95.0414	79.9593
Average	92.133875	94.930325	72.7019

4.3. Sensitivity Analysis

In this section, we conducted sensitivity analysis to our method in order to study the effect of the principal parameters, such as *pop_size*, *pos_weight*, and *neg_weight*. We noticed that the parameters of genetic programming do affect the repair time and repair result. Therefore, we have to evaluate these parameters for some subject files to observe the performance of our method. Although we did not analyze all parameters, this section helps us to estimate the optimal parameters to achieve the best performance of a particular subject file [55]-[57]. In this study, we define

$$\text{Relative Change } (RC) = \frac{MRT - ORT}{ORT} \quad (1)$$

where ORT is the original repair time, and MRT is the modified repair time [58].

- *Effect of variations on pop_size*

In Section 4.2, we have obtained the original repair time for several subject programs. In this paragraph, we present the modified repair time concerning the change of *pop_size*. We selected 3 subject files from the IntroClass benchmark, and we modified the value of *pop_size* by increasing or decreasing 40%, 30%, 20%, or 10%. For other genetic programming parameters, we kept the same value described in Section 4.1. Table 9 shows some numerical values of the modified repair time for the cases of 40%, 30%, 20%, and a 10% increase to *pop_size*. In the situation of increasing *pop_size*, our method's RC ranges from -0.056 to 0.14, and GenProg's RC ranges from -0.079 to 0.071. Table 10 shows some numerical values of the modified repair time for the cases of 40%, 30%, 20%, and a 10% decrease to *pop_size*. In the situation of decreasing *pop_size*, our method's RC ranges from -0.203 to 5.824, and GenProg's RC ranges from -0.418 to 20.541. The result indicates that the effect of pop_size is different for each subject file. For example, the MRT of subject program 2 in Table 10 is unstable, and RC is up to 20.541. However, the MRT of subject programs 1 and 3 remain steady, no matter whether *pop_size* is increasing or decreasing. It seems that there is no global optimal value of *pop_size* for multiple files. On the other hand, these tables show that our method has less MRT than GenProg has in most of the cases whether *pop_size* is increasing or decreasing.

Table 9. Repair time in seconds of GenProg and DLBGP for the case of 40%, 30%, 20%, and a 10% increase in *pop_size*

	***pop_size* × 1.4**		***pop_size* × 1.3**		***pop_size* × 1.2**		***pop_size* × 1.1**	
Subject File	**GenProg**	**DLBGP**	**GenProg**	**DLBGP**	**GenProg**	**DLBGP**	**GenProg**	**DLBGP**
1	7.0242	6.8109	7.0993	6.558	7.3657	7.2684	6.6386	6.1801
2	14.0957	11.4858	13.9267	12.0661	13.989	11.4309	14.4476	11.1637
3	18.2919	8.1408	17.9898	7.898	18.2461	7.9493	18.0489	8.1801

Table 10. Repair time in seconds of GenProg and DLBGP for the case of 40%, 30%, 20%, and a 10% decrease in *pop_size*

	***pop_size* × 0.6**		***pop_size* × 0.7**		***pop_size* × 0.8**		***pop_size* × 0.9**	
Subject File	**GenProg**	**DLBGP**	**GenProg**	**DLBGP**	**GenProg**	**DLBGP**	**GenProg**	**DLBGP**
1	7.2917	6.724	7.3212	6.2833	6.9382	6.6565	6.5636	6.5678
2	148.696	27.2938	15.7623	11.1674	23.5298	45.9627	583.425	95.6477
3	17.9612	fail	18.0112	8.72	18.0211	7.9451	17.9451	7.9232

- *Effect of variations on pos_weight and neg_weight*

In this paragraph, we present the modified repair time analysis with regards to changes in pos_weight and neg_weight. We conducted experiments on 3 subject files from the IntroClass benchmark, modifying the values of *pos_weight* and *neg_weight* by increasing or decreasing them by 40%, 30%, 20%, or 10%. Other genetic programming parameters were kept unchanged as described in Section 4.1. Table 11 displays the numerical values of the modified repair time for the cases of a 40%, 30%, 20%, and 10% increase in *pos_weight*. When *pos_weight* is increased, our method's RC ranges from -0.083 to 0.124, while

GenProg's RC ranges from -0.111 to 0.09. Table 12 presents the numerical values of the modified repair time for the cases of a 40%, 30%, 20%, and 10% decrease in *pos_weight*. When pos_weight is decreased, our method's RC ranges from -0.07 to 2.911, while GenProg's RC ranges from -0.417 to 0.099. Table 13 shows the numerical values of the modified repair time for the cases of a 40%, 30%, 20%, and 10% increase in *neg_weight*. When *neg_weight* is increased, our method's RC ranges from -0.089 to 3.06, while GenProg's RC ranges from -0.378 to 0.234. Table 14 displays the numerical values of the modified repair time for the cases of a 40%, 30%, 20%, and 10% decrease in *neg_weight*. When *neg_weight* is decreased, our method's RC ranges from -0.894 to 0.276, while GenProg's RC ranges from -0.903 to 0.087. The results indicate that the effect of *pos_weight* and *neg_weight* varies across different subject files. For instance, the range of MRT for subject program 2 in Table 12 is higher than for other subject files, with RC reaching up to 2.911. However, the range of MRT for subject files 1 and 3 remains stable regardless of whether pos_weight is increased or decreased. It appears that there are no global optimal values of pos_weight and neg_weight for multiple files. Additionally, these tables demonstrate that our method consistently has lower MRT compared to GenProg in most cases, regardless of whether *pos_weight* and *neg_weight* are increased or decreased.

Table 11. Repair time in seconds of GenProg and DLBGP for the Case of 40%, 30%, 20%, and a 10% increase in *pos_weight*

	***pos_weight* × 1.4**		***pos_weight* × 1.3**		***pos_weight* × 1.2**		***pos_weight* × 1.1**	
Subject File	**GenProg**	**DLBGP**	**GenProg**	**DLBGP**	**GenProg**	**DLBGP**	**GenProg**	**DLBGP**
1	7.3216	7.1223	6.9628	7.1301	7.0945	6.9621	7.4979	7.038
2	26.5344	13.8479	25.3756	13.7507	24.0897	14.8573	24.8189	12.8564
3	17.7299	9.4065	17.7668	9.1841	18.0882	8.6125	18.3667	8.1535

Table 12. Repair time in seconds of GenProg and DLBGP for the Case of 40%, 30%, 20%, and a 10% decrease in *pos_weight*

	***pos_weight* × 0.6**		***pos_weight* × 0.7**		***pos_weight* × 0.8**		***pos_weight* × 0.9**	
Subject File	**GenProg**	**DLBGP**	**GenProg**	**DLBGP**	**GenProg**	**DLBGP**	**GenProg**	**DLBGP**
1	7.0274	6.7494	7.5598	6.5084	7.3458	6.9846	6.9099	6.9944
2	15.7959	54.8165	16.8985	20.2647	17.3614	13.3629	24.5035	13.0390
3	18.2164	8.6375	18.2140	7.8163	18.0758	8.1645	18.1200	7.9047

TABLE 13. Repair time in seconds of GenProg and DLBGP for the case of 40%, 30%, 20%, and a 10% increase in neg_*weight*

	***neg_weight* × 1.4**		***neg_weight* × 1.3**		***neg_weight* × 1.2**		***neg_weight* × 1.1**	
Subject Program	**GenProg**	**DLBGP**	**GenProg**	**DLBGP**	**GenProg**	**DLBGP**	**GenProg**	**DLBGP**
1	8.4892	6.4053	8.3420	6.7461	7.8897	6.5830	7.2879	6.7048
2	17.0780	56.9087	16.8517	14.2788	24.5157	13.0488	24.8748	12.7664
3	17.9951	8.1328	18.0670	9.2514	17.9058	8.6669	18.2277	8.4560

Table 14. Repair time in seconds of GenProg and DLBGP for the case of 40%, 30%, 20%, and a 10% decrease in *neg_weight*

	***neg_weight* × 0.6**		***neg_weight* × 0.7**		***neg_weight* × 0.8**		***neg_weight* × 0.9**	
Subject Program	**GenProg**	**DLBGP**	**GenProg**	**DLBGP**	**GenProg**	**DLBGP**	**GenProg**	**DLBGP**
1	6.7492	6.2987	7.4819	6.3881	6.7959	6.2755	6.8023	6.9360
2	2.6342	1.4819	24.6171	13.0545	24.1398	13.3283	24.3401	12.8233
3	17.9115	9.5296	17.8956	10.6740	17.8754	9.8434	18.0636	10.4919

5. Conclusions and Future Work

In this paper, we introduce DLBGP, a novel APR technology that integrates GenProg, deep learning, and genetic programming. Our approach generates repaired programs progressively while preserving necessary functions and mitigating specific program errors. To address the challenge of a large search space in genetic programming, we employ various techniques, including limited attention to statements, focusing on genetic operations on weighted paths based on test case coverage, and leveraging repeat usage of existing program statements. To predict faults, we utilize a representation-learning methodology to extract semantic features from the source code. Specifically, we employ the Deep Brief Network (DBN) to automatically learn semantic features from node vectors extracted from the program's Abstract Syntax Tree (AST). In our experiments, our method significantly reduced execution time by 64% compared to traditional methods among the 10 subject programs. In modern software development environments, where understanding entire software packages and lacking sufficient test cases and time for verifying target programs can be challenging for software engineers, DLBGP can serve as a valuable tool for debugging and fixing software program errors, eliminating the need for spending days on repairs or resorting to risky temporary solutions. Although we did not

evaluate the performance of our method on larger defective programs due to limited resources and the scale of our deep learning model, we leave this as future work. Moving forward, we plan to explore different types of deep learning models to assess their effectiveness in APR.

ACKNOWLEDGMENT

The work described in this paper was supported by Ministry of Science and Technology, Taiwan, under Grants MOST 108-2221-E-007-033-MY3 and MOST 110-2221-E-007-035-MY3.

REFERENCE

[1] E. Barnett, "Gmail outage affected majority of users, says Google," Telegraph Media Group, [Online].Available: https://www.telegraph.co.uk/technology/google/6125689/Gmail-outage-affected-majority-of-users-says-Google.html. Accessed: 3 May 2019.

[2] W. Leonhard, "Hotmail fail: Microsoft lays an egg in the cloud," IDG Communications,[Online].Available:https://www.infoworld.com/article/2624887/saas/hotmail-fail--microsoft-lays-an-egg-in-the-loud.html. Accessed: 3 May 2019.

[3] D. Perry, "Microsoft and Amazon Cloud Services Struck by Lightning," Tom'sGuide,[Online].Available:https://www.tomsguide.com/us/amazon-ec2-microsoft-cloud-services-outage,news-12108.html.Accessed: 3 May 2019.

[4] B. Abdallah, Y. Benyssaad, D. Mohamed, B. Benaissa, and Y. Benabdellah, "Maintenance Optimization for Complex System using Evolutionary Algorithms under Reliability Constraints within the Context of the Reliability-Centered-Maintenance." *International Journal of Performability Engineering*, vol. 17, no. 1, pp. 1-13, January 2021.

[5] R. C. Scacord, D. Plakosh and G. A. Lewis, Modernizing Legacy Systems: Software Technologies, Engineering Processes, and Business Practices., Boston, USA: Addison-Wesley Professional, 2003.

[6] M. Jorgensen and M. Shepperd, "A Systematic Review of Software Development Cost Estimation Studies," *IEEE Trans. on Software Engineering*, Vol. 33, No. 1, pp. 33-53, Jan. 2007.

[7] J. Sutherland, "The Object Technology Architecture: Business Objects for Corporate Information Systems," in Business Object Design and Implementation, London, 1997.

[8] Z. Yu, M. Martinez, B. Danglot, T. Durieux and M. Monperrus, "Test Case Generation for Program Repair: A Study of Feasibility and Effectiveness," 2017, https://arxiv.org/pdf/1703.00198.pdf.

[9] C. L. Geoues, T. Nguyen, S. Forrest and W. Weimer, "GenProg: A Generic Method for Automatic Software Repair," *IEEE Trans. on Software Engineering*, Vol. 38, No. 1, pp. 54-72, 2012.

[10] M. Mossige, A. Gotlieb and H. Meling, "Using CP in Automatic Test Generation for ABB Robotics' Paint Control System," in Principles and Practice of Constraint Programming, Lyon, France, Springer, Cham, pp. 25-41,2014.

[11] A. Gotlieb and D. Marijan, "FLOWER: optimal test suite reduction as a network maximum flow," *Proceedings of the 2014 International Symposium on Software Testing and Analysis (ISSTA)*, San Jose, CA, USA, pp.171-180, 2014.

[12] S. Wang, S. Ali and A. Gotlieb, "Cost-effective test suite minimization in product lines using search techniques," *Journal of Systems and Software*, Vol. 103, No. C, pp. 370-391, 2015.

[13] A. Gotlieb, M. Carlsson, D. Marijan and A. Petillon, "A New Approach to Feature-based Test Suite Reduction in Software Product Line Testing," *Proceedings of the 11th International Conference on Software Engineering and Applications (ICSOFT-EA)*, Lisbon, Portugal, pp.48-58, 2016.

[14] H. Spieker, A. Gotlieb, D. Marijan and M. Mossige, "Reinforcement learning for automatic test case prioritization and selection in continuous integration," *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, Santa Barbara, CA, USA, pp.12-22, 2017.

[15] D. Marijan, M. Liaaen, A. Gotlieb, S. Sen and C. Leva, "TITAN: Test Suite Optimization for Highly Configurable Software," *Proceedings of the 10th IEEE International Conference on Software Testing, Verification and Validation (ICST)*, Tokyo, Japan, pp.524-531, 2017.

[16] M. Mossige, A. Gotlieb, H. Spieker, H. Meling and M. Carlsson, "Time-aware Test Case Execution Scheduling for Cyber-Physical Systems," *Proceedings of the 23rd International Conference on Principles and Practice of Constraint Programming*, Melbourne, Australia, 2017.

[17] C. L. Goues, N. Holtschulte, E. K. Smith, Y. Brun, P. Devanbu, S. Forrest and W. Weimer, "The ManyBugs and IntroClass Benchmarks for Automated Repair of C Programs," *IEEE Trans. on Software Engineering*, Vol. 41, No. 12, pp. 1236-1256, 2015.

[18] H. D. T. Nguyen, D. Qi, A. Roychoudhury and S. Chandra, "SemFix: Program repair via semantic analysis," *Proceedings of the 35th International Conference on Software Engineering (ICSE)*, San Francisco, CA, USA, pp.772-781, 2013.

[19] J. Xuan, M. Martinez, F. DeMarco, M. Clement, S. L. Marcote, T. Durieux, D. L. Berre and M. Monperrus, "Nopol: Automatic Repair of Conditional Statement Bugs in Java Programs," *IEEE Trans. on Software Engineering*, Vol. 43, No. 1, pp. 34-55, 2017.

[20] S. Mechtaev, J. Yi and A. Roychoudhury, "DirectFix: Looking for Simple Program Repairs," *Proceedings of the 37th IEEE International Conference on Software Engineering (ICSE)*, Florence, Italy, 2015.

[21] S. Mechtaev, J. Yi and A. Roychoudhury, "Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis," *Proceedings of the 38th International Conference on Software Engineering (ICSE)*, Austin, TX, USA, pp.691-701, 2016.

[22] W. Weimer, T. Nguyen, C. L. Goues and S. Forrest, "Automatically finding patches using genetic programming," *Proceedings of the 31st International Conference on Software Engineering(ICSE)*, Vancouver, BC, Canada, pp.364-374, 2009.

[23] M. Gabel and Z. Su, "A study of the uniqueness of source code," *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering (FSE'10)*, Santa Fe, New Mexico, USA, pp.147-156, 2010.

[24] A. Hindle, E. T. Barr, Z. Su, M. Gabel and P. Devanbu, "On the naturalness of software," *Proceedings of the 34th International Conference on Software Engineering (ICSE)*, Zurich, Switzerland, pp.837-847, 2012.

[25] E. T. Barr, Y. Brun, P. Devanbu, M. Harman and F. Sarro, "The plastic surgery hypothesis," *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, Hong Kong, China, pp.306-317, 2014.

[26] X. Kong, L. Zhang, W.E. Wong. and B. Li, "Experience report: How do techniques, programs, and tests impact automated program repair?, " *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*, pp. 194-204, Nov. 2015.

[27] Y. Qi, X. Mao, Y. Lei and C. Wang, "Using automated program repair for evaluating the effectiveness of fault localization techniques," *Proceedings of the 2013 International Symposium on Software Testing and Analysis (ISSTA)*, Lugano, Switzerland, pp.191-201, 2013.

[28] H. Yokoyama, Y. Higo, K. Hotta, T. Ohta, K. Okano and S. Kusumoto, "Toward improving ability to repair bugs automatically: a patch candidate location mechanism using code similarity," *Proceedings of the 31st Annual ACM Symposium on Applied Computing (SAC'16)*, Pisa, Italy, pp.1364-1370, 2016.

[29] M. Martinez and M. Monperrus, "ASTOR: a program repair library for Java (demo)," *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA)*, Saarbrücken, Germany, pp.441-444, 2016.

[30] D. Kim, J. Nam, J. Song and S. Kim, “Automatic patch generation learned from human-written patches,” *Proceedings of the 35th International Conference on Software Engineering (ICSE)*, San Francisco, CA, USA, pp.802-811, 2013.

[31] F. Long and M. Rinard, “Staged program repair with condition synthesis,” *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015)*, Bergamo, Italy, pp.166-178, 2015.

[32] C. T. Lin and C. Y. Huang, “Staffing Level Analysis of Software Debugging through Rate-Based Simulation Approaches,” *IEEE Trans. on Reliability*, Vol. 58, No. 4, pp. 711-724, Dec. 2009.

[33] W. E. Wong, T. Sugeta, Y. Qi, and J. C. Maldonado, “Smart debugging software architectural design in SDL,” *Journal of Systems and Software*, vol. 76, no. 1, pp.15-28, Apr. 2005.

[34] Y. S. You, C. Y. Huang, K. L. Peng, and C. J. Hsu, “Evaluation and Analysis of Spectrum-Based Fault Localization with Modified Similarity Coefficients for Software Debugging,” *Proceedings of the 37th Annual IEEE International Computer Software and Applications Conference (COMPSAC 2013)*, Kyoto, Japan, pp. 180-189, July 2013.

[35] W. E. Wong, R. Gao, Y. Li, R. Abreu and F. Wotawa, "A Survey on Software Fault Localization," *IEEE Trans. on Software Engineering*, vol. 42, no. 8, pp. 707-740, 1 Aug. 2016

[36] W. E. Wong and V. Debroy, "Software Fault Localization," Encyclopedia of Software Engineering, vol. 1, pp. 1147-56, Sep. 2010

[37] W. E. Wong and T. H. Tse, "Handbook of Software Fault Localization: Foundations and Advances," Edition 1, Wiley-IEEE Computer Society Press, May 2023

[38] A. Dutta, "Poster: EBFL-An Ensemble Classifier based Fault Localization," *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*, Valencia, Spain, pp. 473-476,2022.

[39] Y. Li and P. Liu, "A Preliminary Investigation on the Performance of SBFL Techniques and Distance Metrics in Parallel Fault Localization," *IEEE Trans. on Reliability*, vol. 71, no. 2, pp. 803-817, June 2022.

[40] C. H. Lee, C. Y. Huang, and T. Y. Lin, "A Study of Applying Fault-Based Genetic-Like Programming Approaches to Automatic Software Fault Corrections," *International Journal of Performability Engineering*, Vol. 14, No. 9, pp. 2090-2104, Sept. 2018.

[41] B. L. Miller and D. E. Goldberg, “Genetic Algorithms, Selection Schemes, and the Varying Effects of Noise,” *Evolutionary Computation*, Vol. 4, No. 2, pp. 113-131, 1996.

[42] A. E. Eiben and J. E. Smith, Introduction to Evolutionary Computing, Heidelberg: Springer, 2003.

[43] H. Peng, L. Mou and G. Li, “Building Program Vector Representations for Deep Learning,” *in International Conference on Knowledge Science, Engineering and Management (KSEM 2015)*, Chongqing, China, pp.547-553, 2015.

[44] W. Weimer, T. Nguyen, C. L. Goues and S. Forrest, “Automatically finding patches using genetic programming,” *Proceedings of the 31st International Conference on Software Engineering(ICSE)*, Vancouver, BC, Canada, pp.364-374, 2009.

[45] G. E. Hinton, S. Osindero and Y.-W. Teh, “A Fast Learning Algorithm for Deep Belief Nets,” *Neural Computation*, Vol. 18, No. 7, pp. 1527-1554, 2006.

[46] G. C. Necula, S. McPeak, S. P. Rahul and W. Weimer, “CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs,” *Proceedings of the 11th International Conference on Compiler Construction (CC'02)*, Grenoble, France, pp.213-228, 2002.

[47] S. Wang, T. Liu and L. Tan, “Automatically Learning Semantic Features for Defect Prediction,” *Proceedings of the 38th International Conference on Software Engineering (ICSE)*, Austin, TX, USA, pp.297-308, 2016.

[48] A. Krizhevsky, I. Sutskever and G. E. Hinton, “ImageNet classification with deep convolutional neural networks,” *Communications of the ACM*, Vol. 60, No. 6, pp. 84-90, 2017.

[49] M. Chhabra, M.K. Shukla. and K.K. Ravulakollu, “Intelligent Optimization of Latent Fingerprint Image Segmentation using Stacked Convolutional Autoencoder, ” *International Journal of Performability Engineering*. vol. 17, no. 4, April 2021.

[50] J. Li, P. He, J. Zhu and M. R. Lyu, “Software Defect Prediction via Convolutional Neural Network,” *Proceedings of the International Conference on Software Quality, Reliability and Security (QRS)*, Prague, Czech Republic, pp. 318-328, 2017.

[51] J. Yi, U. Z. Ahmed, A. Karkare, S. H. Tan and A. Roychoudhury, “A feasibility study of using automated program repair for introductory programming assignments,” *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017)*, Paderborn, Germany, pp.740-751, 2017.

[52] W. Weimer, Z. P. Fry and S. Forrest, "Leveraging program equivalence for adaptive program repair: models and first results," *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering (ASE'13)*, Silicon Valley, CA, USA, pp.356-366, 2013.

[53] E. Fast, C. L. Goues, S. Forrest and W. Weimer, "Designing better fitness functions for automated program repair," *Proceedings of the 12th annual conference on Genetic and evolutionary computation (GECCO '10)*, Portland, Oregon, USA, pp.965-972, 2010.

[54] T. Jones and S. Forrest, "Fitness Distance Correlation as a Measure of Problem Difficulty for Genetic Algorithms," *Proceedings of the 6th International Conference on Genetic Algorithms*, San Francisco, CA, USA, pp.184-192,1995.

[55] S. S. Gokhale and K. S. Trivedi, "Reliability prediction and sensitivity analysis based on software architecture," *Proceedings of the 13th International Symposium on Software Reliability Engineering (ISSRE 2002)*, Annapolis, MD, USA, pp.1-12, 2002.

[56] A. Pasquini, A. N. Crespo and P. Matrella, "Sensitivity of reliability-growth models to operational profile errors vs. testing accuracy," *IEEE Trans. on Reliability*, Vol. 45, No. 4, pp. 531-540, 1996.

[57] C. Y. Huang, J. H. Lo, J. W. Lin, C. C. Sue and C. T. Lin, "Optimal resource allocation and sensitivity analysis for modular software testing," *in Proceedings of the IEEE Fifth International Symposium on Multimedia Software Engineering*, Taichung, Taiwan, 2003.

[58] C. Y. Huang and M. R. Lyu, "Optimal Testing Resource Allocation, and Sensitivity Analysis in Software Development," *IEEE Trans. on Reliability*, Vol. 54, No. 4, pp. 592-603, 2005.