PROCEEDINGS OF SCIENCE

Fault Localization Method Based on Enhanced GA-BP Neural Network

Bei Zhang[1]
Institute of Information Engineering , Capital Normal University
Beijing, 100048, China
E-mail: godandzb@163.com

Shudong Zhang
Institute of Information Engineering , Capital Normal University
Beijing, 100048, China
E-mail: zsd@.cnu.edu.cn

In the process of software development and maintenance, software debugging is the most complicated and expensive part. In recent years, automated software fault localization technology has attracted many scholars' attention, various approaches have been proposed. In this paper, a technique named EGA-BPN is proposed which can provide suspicious locations for fault localization automatically without requiring any prior information of program structure or semantics. EGA-BPN is a software fault localization method based on enhanced Genetic Algorithm-Back Propagation neural network. Firstly, through processing running traces of the program, coverage information of test cases is converted to the training samples of neural network; secondly, the initial weights and thresholds of the neural network are computed by GA, the training data are substituted in neural network in training orderly, and then use orthogonal experimental design helping to adjust the parameters of the neural network; finally, test matrix is calculated by the neural network to count the suspiciousness of each statement, and the fault is located at the statements with higher suspicious value. Through comparative experiments between the proposed method, GA-BPN and BPN, the experiment results show that the enhanced GA-BP neural network-based fault localization technology has certain validity.

ISCC 2015
18-19, December 2015
Guangzhou, China

[1]Speaker

PoS(ISCC2015)054

1. Introduction

In the last few years, foreign and Chinese scholars have done a variety of studies about the automation of software fault localization, and have obtained a lot of achievements. According to the difference of principle, these theories can be divided into: methods based on program slicing, methods based on program spectrum and methods based on program state.

Program slicing-based methods[1-3]: program slicing, as the name suggests, is to reduce the scope of the program. The main idea of this method is to construct a collection of code in a program that may be associated with the error outputs. The code set includes two contents: fault statements and debugging context related to the fault statements that can help staff to understand the program, as far as possible to narrow the scope of suspicious statements, reduce the number of code lines to be detected, and improve the efficiency of software debugging.

Program spectrum-based methods: by locating different elements in the program, for example, an executable statement or a statement block, a predicate and an information flow path [4-8], obtain different program spectrum. The main idea of this method is to use the differences of program spectrum between successful cases and unsuccessful cases to obtain the suspiciousness of each element of the program then sort. Calculation of suspiciousness is based on statistical data or a variety of mathematical calculation method.

Program state-based methods [9-10]: the central idea of program state-based methods is as follows: first of all, count the running state of successful test cases and unsuccessful cases in the execution process, draw the difference between them, and then modify the running state of unsuccessful case based on different rules, and find out the location of the key statement by revised test results.

In addition to the traditional fault localization methods, some scholars have applied neural network to the field of fault localization in recent years. Such as in 2009, Wong et al proposed BP neural network-based fault localization algorithm [11]. In this paper, a new technique based on the idea of genetic algorithm and orthogonal experiment design is proposed -- fault localization method based on enhanced GA-BP neural network; EGA-BPN uses genetic algorithm to calculate the initial setting of the parameters of back propagation neural network, and use the orthogonal experiment design method to adjust the value of parameters, in order to achieve a better fault localization result.

2.Enhanced GA-BP Neural Network Model

2.1An Overview of the GA-BP Neural Networks and Orthogonal Experiment Design

Propagation back neural network is divided into: the input layer, the hidden layer and the output layer. Fig. 1 shows the structure of a three-layer BP neural network [12]. It needs to train a lot of times for the identification of nonlinear or other more complex relationships, and it can't rule out the local minimum points. Aiming at solving these problems of BP neural network, one of the improved methods is using genetic algorithm's ability "survival of the fittest" to overcome the defects and deficiencies of the BP neural network, and thus to upgrade it.

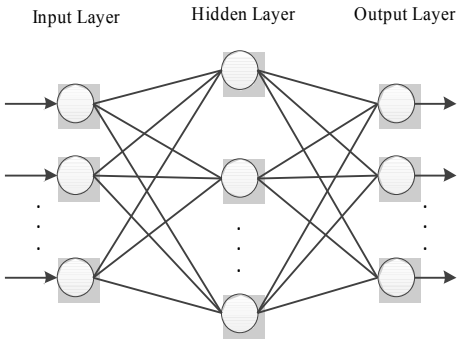

Figure 1: Structure of a BP Neural Network

Orthogonal experiment design is a method of designing a multi factor and multi-level test. It selects some representative points from the whole test according to the orthogonal experiment. These representative points are "uniformly dispersed, neat comparable characteristics", scientifically arranged and analyzed by using the orthogonal table. The main advantage is that it can be selected from a large number of test schemes, and the results of these tests are analyzed, and the optimal scheme is deduced. It is an efficient, rapid and economical method of test design.

2.2 Execution Flow of EGA-BPN Model

The central idea of EGA-BPN is to use GA to deal with the connection weights and thresholds of each layer of the neural network, select the best individual, which is used as the initial setting of the weights and thresholds of the neural network, and after the input values into neural network, compare the predicted results with the expected output values; according to the deviation continue to use OED to adjust the network parameter values, and then perform again until the neural network training is completed. Input test data and summarize the results of the output. Fig. 2 shows the execution flow of EGA-BPN model:

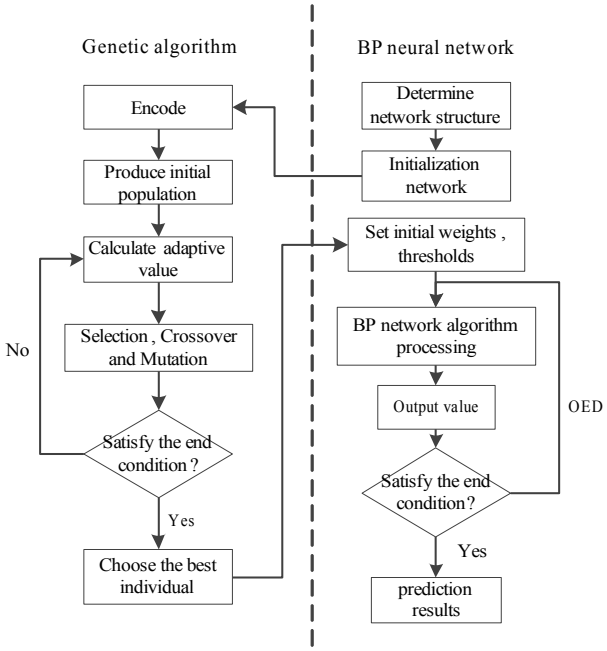

Figure 2: Execution Flow of EGA-BPN Model

2.3 Training Process of EGA-BPN Model

EGA-BPN model training process is as follows:

- encode to get the initial population of the genetic, each encoded string, that is, eachchromosome contains the weights and the thresholds of each layer of a neural network;
- determine the structure of the neural network, and calculate the possibility of gene of the chromosomes inherited to future generations, namely the adaptive value F. F is defined by the error between the predicted value y_i and the expected value o_i of the training data. The larger the adaptive value of the chromosome, the greater the genetic potential of the gene, and the calculation formula of F is as follows:

$$F = k\left(\sum_{i=0}^{n} abs\left(y_i - o_i\right)\right) \tag{2.1}$$

Where n is the number of output nodes, and k is the coefficient. After crossover and mutation operation, the best chromosome is selected as the initial setting value of neural network weights and thresholds;

- set the number of nodes in the input layer, the hidden layer and the output layer of the neural network as *m*, *p* and *q*. The connection weights among the three layers are w_{ij} and v_{jk}, threshold of the hidden layer is α_i, threshold of the output layer is β_k, and determine learning parameters and neuron excitation functions $f(x)$. The network input data is represented as a matrix:

$$X=[X_1, X_2, \ldots, X_m] \tag{2.2}$$

- the hidden layer output Y can be calculated by the input vector X, w_{ij} and α_i. The output layer result Z can be calculated by *Y*, v_{jk} and β_k. The specific formulas are as follows:

$$Y_j=f\left(\sum_{i=1}^{m} w_{ij}X_i-\alpha_i\right) j=1,2,\ldots,p \tag{2.3}$$

$$Z_k=f\left(\sum_{j=1}^{p} v_{jk}Y_j-\beta_k\right) k=1,2,\ldots,q \tag{2.4}$$

- if the expected output value O_k of neuron X_i, the error *E* can be calculated according to the formula of the mean square error function:

$$E=\frac{1}{2}\sum_{k}(O_k-Z_k)^2 \tag{2.5};$$

- compare the error *E* with default values. If *E* is less than the default value or training number has reached the preset number, the training is completed; on the contrary, the error correction should be carried out according to the following formula:

$$\begin{pmatrix} w_{ij}(n+1)=w_{ij}(n)+\Delta w_{ij} \\ v_{jk}(n+1)=v_{jk}(n)+\Delta v_{jk} \end{pmatrix} \tag{2.6}$$

Where n represents the number of training times of the network, Δw_{ij} and Δv_{ik} respectively represent revised weights and thresholds, and the formula is as follows:

$$\begin{pmatrix} \Delta w_{ij}=-\sigma\dfrac{\partial E}{\partial w_{ij}} \\ \Delta v_{jk}=-\sigma'\dfrac{\partial E}{\partial v_{kj}} \end{pmatrix} \tag{2.7};$$

- in the formula (2.7), σ and σ' indicate learning parameters, whose values are combined with OED to adjust, and the formula is as follows:

$$\begin{pmatrix} \sigma_{.1}=\sigma_{.}+\Delta\sigma \\ \sigma_{.1}=\sigma_{.}-\Delta\sigma \end{pmatrix} \tag{2.8}$$

Where $\Delta\sigma$ is the learning parameter adjustment value, and generally is a small number [13];

- use the modified parameters as a new network connection weights and thresholds. Return to step 3, until reach the ideal state or training number reaches the preset number.

3. Fault Localization Method Based on EGA-BPN Neural Network

The process of software fault localization algorithm based on EGA-BPN is as follows:

- get program execution information. For a wrong program P which can run normally, for example, the summation procedure in Table 1[14]. There are 9 statements in this program, in accordance with the order, the statement numbers are S_1, S_2,…, S_9, and the error is in statement S_3. There are 8 test cases in this program and the numbers are t_1, t_2,…, t_8.

	Program	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8
S_1	read(a,b);	1	1	1	1	1	1	1	1
S_2	if (a<10&&b<10)	1	1	1	1	1	1	1	1
S_3	result=a-b;// correct: a+b	1	1	1	1	0	1	1	1
S_4	if(result>0)	1	1	1	1	0	1	1	1
S_5	print("positive");	1	0	0	0	0	1	1	0
S_6	else if (result==0)	0	1	1	1	0	0	0	1
S_7	print("zero");	0	1	0	0	0	0	0	0
S_8	else print("negative");	0	0	1	1	0	0	0	1
S_9	else print("invalid input");	0	0	0	0	1	0	0	0
$flag$		1	1	1	1	0	0	0	0

Table 1: An Example of Coverage Information

For a test case $t_k=\langle in_k, out_k, C_k, flag\rangle$, in_k represents the input data of t_k; out_k represents the output data of t_k; C_k represents the program coverage information. In the running process of t_k, if statement S_i is executed, set the cover identifier for the statement C_{ki} to 1, otherwise set it to 0; $flag$ is the program execution result identifier. If out_k is equal to the expected output of in_k, set $flag$ of t_k to 0, otherwise set it to 1;

- encode weights and thresholds of each layer of the neural network to form initial population of genetic algorithm. Use crossover and mutation operator to calculate the highest fitness chromosome, and the final result is the initial setting of the parameters of the BP neural network;
- the next step is to train the neural network. The coverage information of each test case are the input layer neurons of the neural network. For the example in Table 1, the number of EGA-BPN input layer is 9 and there are 8 sets of test cases in total. The output value of the neural network is compared with the $flag$ of each test case to get the error. If the error value is greater than the preset error value, then OED will be combined to adjust parameter values. Loop the process until the error value is less than or equal to the preset error value or the number of cycles has exceeded the maximum cycle number;
- once the neural network training is completed, a good map is established between the input data and the output data. We use a set of virtual test cases v_1, v_2, ..., v_9, whose coverage vectors are C_{v1}, C_{v2}, ..., C_{v9} [15], where

$$\begin{pmatrix} C_{v1} \\ C_{v2} \\ \vdots \\ C_{v9} \end{pmatrix} = \begin{pmatrix} 10\cdots0 \\ 01\cdots0 \\ \vdots\ \vdots\ \ddots\ \vdots \\ 00\cdots1 \end{pmatrix} \quad (3.1);$$

- the virtual test case data are assigned to the input layer of the network, and the prediction results are f_1, f_2, ..., f_9. The value of f_i is closer to 0, so the result of the test case v_i is more likely to be successful, and the covered statement s_i is less likely to be wrong;
- the output of the neural network, that is, the suspiciousness of statements provides a reference to the staff. Programmers can directly begin to check from the most suspicious statements, thereby save a lot of time and improve work efficiency.

4.Experimental Results and Analysis

4.1A Sample Program

The experiment of the EGA-BPN algorithm needs four elements of test cases: input data, output data, coverage information and program execution result identifier. If the scale of the test procedure is large, first of all, the program should be divided into code blocks, and the block is the smallest unit of the experiment. There are many ways to bolck the program, for example, we can set all statements of a fuction as a block, so the program can be divided into blocks according to the fuction, or we can simply set 10 statements as a block. We should record the mapping relationship between blocks and statements, that is, the range and number of sentences of each block, in order to position errors accurately.

Then execute test cases to get the coverage information of test cases, compare the output data of the system with the expected value to obtain the execution result identifier; finally, all the information is put together to get a data file similar to Table 1. As the input data of the EGA-BPN algorithm, the data file is assigned to the neural network which has been trained before. The prediction results of the network are arranged in order. The higher ranking of the code block, the more likely the block is wrong. Table 2 shows the result of the sample program in Table 1 executed by the EGA-BPN algorithm:

Statement	Output	Statement	Output	Statement	Output
S_1	0.8356	S_4	0.35486	S_7	0.00915
S_2	0.9775	S_5	0.09429	S_8	0.0756
S_3	0.9998	S_6	0.68125	S_9	0.0996

Table 2: Execution Result of Sample Program

According to the data in Table 2, we can see that suspiciousness of statement S_3 is the highest, which suggests that S_3 is most likely to be wrong, and this is in conformity with the actual situation.

4.2 Siemens Suite

In order to further demonstrate the effectiveness of this method, the Siemens suite is introduced as a data source for further experiments. The Siemens suite is an open source which is provided by Software-artifact Infrastructure Repository , and most of it is written by C language. Each program has a correct version, several wrong versions, and many test cases. Specific information is shown in the following Table 3. However, not all of the wrong versions in the Siemens suite are suitable for this experiment, such as, the wrong statement of error version 4 of program Print_tokens exists in the head file, and cannot get the wrong statement coverage information; the error version 27 and error version 32 of program Replace, an error occurs in the process of executing test cases, so as to produce abnormal program termination, and cannot conclude the output data.

Program	**Number of faulty versions**	**Number of executable statements**	**Number of test cases**
Print_tokens	7	4130	175
Print_tokens2	10	4115	128
Replace	32	5542	216
Schedule	9	2650	121
Schecule2	10	2710	112
Teas	41	1608	55
Tot_info	23	1052	113

Table 3: Table Summary of The Siemens Suite

In order to test the effectiveness of the algorithm, the localization efficiency is introduced as the evaluation index [16]. The localization efficiency of an algorithm is the energy spend in the process of positioning error, that is the ratio of percentage of faulty versions where fault is located and percentage of executable statements needs to be examined. For example, suppose A algorithm and B algorithm, by examining less than 20% of the code, A can locate 30% of the faults in test program whereas B can only locate 15%, or 50% of the faulty versions have been

located fault, and A has examined 30% of the code whereas B has examined 40%. So obviously, the localization efficiency of A is higher than thatof B, and algorithm A is more efficient than algorithm B. Fig. 3 shows a comparison of the localization efficiency of the EGA-BPN with GA-BPN and BPN:

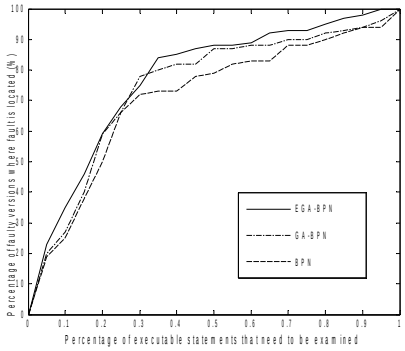

Figure 3: Comparison of the Localization Efficiency of EGA-BPN with GA-BPN and BPN

In Fig. 3, x axis represents percentage of executable statements that needs to be examined, and y axis represents percentage of faulty versions where fault is located. It can be seen from the curve that by examining the same percent of code, EGA-BPN can locate more faulty versions than GA-BPN and BPN, so EGA-BPN has the best performance.

For further details, we use the percentage Imp [17] to compare the three algorithms. The percentage Imp represents in a single program, in the case of all the error versions being found out, and the percentage of the total number of statements spending on the search and the total number of statements of error versions. The lower the percentage Imp is, the less the search time is consumed, and the higher the efficiency is. Fig. 4 shows a comparison of the percentage Imp of the EGA-BPN with GA-BPN and BPN:

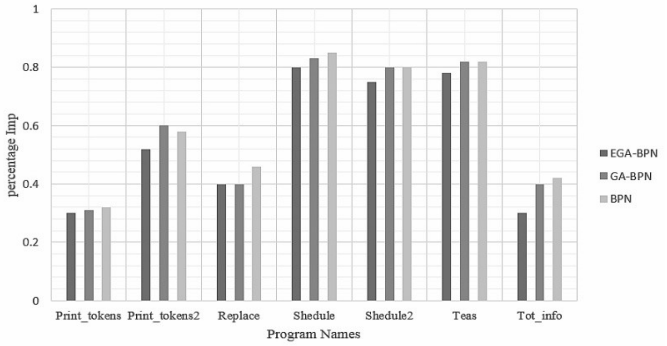

Figure 4: Comparison of the Percentage Imp of EGA-BPN with GA-BPN and BPN

In Fig. 4, x axis represents program names in the Siemens suite, and y axis represents the value of the percentage Imp. It can be seen from the Figure that fault localization method based on EGA-BPN can locate errors faster than fault localization method based on GA-BPN and BPN.

5. Conclusion

In this paper, genetic algorithm, BP neural network and orthogonal experiment design are integrated and applied to the field of fault localization. A new fault localization technique based on enhanced GA-BP neural network is proposed. Experiments show that the EGA-BPN algorithm is better than the GA-BPN algorithm and the BPN algorithm in the efficiency and accuracy of locating faults.

Although the fault localization algorithm has achieved good results, the algorithm also has many shortcomings, for example, the running results only provide a suspiciousness rank of statements. Programmers have to check statements one by one according to the rank, manually modify the error and execute the program again to make sure that the error has been modified; furthermore, errors in the experimental objects used in experiments, the Siemens suite, are single error, without considering the program may have multiple errors or the relationship between the multiple errors. In addition, the accuracy of the neural network prediction has a

great relationship with the training data, the more training data are, and the more accurate prediction results. However, in actual engineering, due to the complexity of the structure or the logic of the program, it will take a lot of time to obtain test cases, coverage information, and execution results of a program. These problems will continue to be optimized in the follow-up study.

References

[1] C. D. Sterling, R. A. Olsson. *Automated bug isolation via program chipping* [J]. Software: Practice and Experience, 2007, 37(10): 1061-1086.

[2] X. Zhang, N. Gupta, R. Gupta. *Locating faulty code by multiple points slicing* [J]. Software: Practice and Experience, 2007, 37(9): 935-961.

[3] W. E. Wong, Y. Qi. *Effective program debugging based on execution slices and inter-block data dependency* [J]. Journal of Systems and Software, 2006, 79(7): 891-903.

[4] R. Abreu, P. Zoeteweij, A. J. C. Van Gemund. *On the accuracy of spectrum-based fault localization* [C]. Testing: Academic and Industrial Conference Practice and Research Techniques Mutation, ACM, USA, 2007: 89-98.

[5] J. A. Jones, M. J. Harrold, J. Stasko. *Visualization of test information to assist fault localization*[C]. Proceedings of the 24th international conference on Software engineering, ACM, USA2002: 467-477.

[6] W. Masri. *Fault localization based on information flow coverage* [J]. Software Testing: Verification and Reliability, 2010, 20(2): 121-147.

[7] S. S. Murtaza, N. Madhavji, M. Gittens and et al. *Diagnosing new faults using mutants and prior faults* (NIER track)[C]. IEEE 33rd International Conference on Software Engineering, IEEE, USA, 2011: 960-963.

[8] K. Yu, M. Lin, Q. Gao and et al. *Locating faults using multiple spectra-specific models*[C]. Proceedings of the 2011 ACM Symposium on Applied Computing, ACM, USA , 2011: 1404-1410.

[9] X. Zhang, N. GuPta, and T. GuPta. *Locating Faults through Automated Predicate Switching*[C]. In the 28th International Conference on Software Engineering (ICSE.06), ACM/IEEE , USA, May 2006:272-281.

[10] H. Cleve and A. Zeller. *Locating Causes of Program Failures*[C].In the 27th International Conference on Software Engineering (ICSE.05), ACM/IEEE , USA, 2005:342-351.

[11] W. E. Wong and Y. Qi. *BP neural network-based effective fault localization* [J]. International Journal of Software Engineering and Knowledge Engineering, 2009,19(4):573–597.

[12] Z. ZZ. Shi. *Neural Nerwork*. Beijing. Higher Education Press, 2009:43-45.(In Chinese)

[13] K. Zhang, D. P. Zhang, and S. Wang. *Fault localization method based on enhanced radial basis function neural network*. [J]. Appication Research of Computers. 2015, 32, 3.(In Chinese)

[14] W. Eric Wong, Vidroha Debroy and et al. *The DStar Method for Effective Software Fault Localization* [J]. IEEE Transactions on Reliability, 2014, 63(1): 290-308.

[15] W. E. Wong, Debroy V, *Thuraisingham B and et al. RBF Neural Network-based fault location* [J]. Technical report UTDCS-20-10, Department of Computer Science, University of Texas at Dallas, 2010.

[16] Y. Lei, X. G. Mao, Z. Y. Dai and et al. *Effective statistical fault localization using program slices* [C]. IEEE 36th International Conference on Computer Software and Applications, IEEE, USA,2012.pp:1-10

[17] V. Debroy, W. E. Wong, X. Xu and et al. *A Grouping-Based Stategy to Improve the Effectiveness of Fault Localization Techniques*[C]. In the 10th International Conference on Quality Software, Zhangjiajie, China, 2010, pp: 13-22.