

BP NEURAL NETWORK-BASED EFFECTIVE FAULT LOCALIZATION

W. ERIC WONG* and YU QI

*Department of Computer Science, University of Texas at Dallas
Richardson, TX 75080, USA*

**ewong@utdallas.edu*

In program debugging, fault localization identifies the exact locations of program faults. Finding these faults using an ad-hoc approach or based only on programmers' intuitive guesswork can be very time consuming. A better way is to use a well-justified method, supported by case studies for its effectiveness, to automatically identify and prioritize suspicious code for an examination of possible fault locations. To do so, we propose the use of a back-propagation (BP) neural network, a machine learning model which has been successfully applied to software risk analysis, cost prediction, and reliability estimation, to help programmers effectively locate program faults. A BP neural network is suitable for learning the input-output relationship from a set of data, such as the inputs and the corresponding outputs of a program. We first train a BP neural network with the coverage data (statement coverage in our case) and the execution result (success or failure) collected from executing a program, and then we use the trained network to compute the *suspiciousness* of each executable statement,^a in terms of its likelihood of containing faults. Suspicious code is ranked in descending order based on its suspiciousness. Programmers will examine such code from the top of the rank to identify faults. Four case studies on different programs (the Siemens suite, the Unix suite, grep and gzip) are conducted. Our results suggest that a BP neural network-based fault localization method is effective in locating program faults.

Keywords: Fault localization; program debugging; BP (Back-Propagation) neural network; suspiciousness of code; successful test; failed test.

1. Introduction

During program debugging, fault localization is the activity of identifying the exact locations of program faults.^b It is a very expensive and time consuming process. Its effectiveness depends on developers' understanding of the program being debugged, their ability of logical judgment, past experience in program debugging, and how

^aAll the comments, blank lines, non-executable statements (e.g., function and variable declarations) are excluded for analysis. When there is no ambiguity, we refer to "executable statements" as simply "statements" from this point on.

^bIn this paper, we use "bugs," "faults," and "defects" interchangeably. In addition, "a statement is covered by a test case" and "a statement is executed by a test case" are used interchangeably.

suspicious code, in terms of its likelihood of containing faults, is identified and prioritized for an examination of possible fault locations. It is very often that programmers have a huge amount of data collected from program testing available in hand while they are performing program debugging. The challenge is how to use such data to help them effectively locate program faults. A summary of several popular fault localization methods can be found in Sec. 6: Related Studies.

In this paper we propose a fault localization method based on a back-propagation (BP) neural network which is one of the most popular neural network models in practice [10]. A BP neural network has a simple structure, which makes it easy to implement using computer programs or circuits. At the same time, BP neural networks have the ability to approximate complicated nonlinear functions [13]. They have been successfully applied in software engineering. For example, Neumann [25] proposes a technique for combining principal component analysis and BP neural networks for software risk analysis. Tadayon [31] presents a BP neural network approach for software cost estimation. Su and Huang [30] report a BP neural network-based study for software reliability estimation. Anderson, Mayrhauser and Mraz [4] apply BP neural networks to predict the severity levels of program faults that are likely to be uncovered, if any, by each test case. However, to our best knowledge, no studies have used BP neural networks for fault localization.

Before we can use a BP network to identify suspicious code for possible locations of program bugs, we need to run the program on a set of test cases and collect the coverage data with respect to each test execution. This is done by using a coverage measurement tool such as χ Suds [38]. In our proposed method, the coverage data of each test case is focused on the statement coverage in terms of which statements are executed by which test case.^c The execution result (success or failure) of each test case is also collected. Together, the coverage data and the execution result are used to train a BP neural network so that the network can learn the relationship between them. We also use a set of virtual test cases that each covers only one statement in the program. When these coverage data are input into a trained BP network, the outputs can be regarded as the likelihood (i.e., suspiciousness) of each statement of containing the bug. Suspicious code is ranked in descending order based on its suspiciousness. Programmers can examine these statements from the top of the rank one by one until they find the bug(s). Four case studies using the Siemens suite [29], the Unix suite [34], the grep and gzip programs, respectively, are conducted to demonstrate the effectiveness of our method. The results when compared with those from other studies are very promising.

The remainder of this paper is organized as follows. Section 2 gives an overview of BP neural networks. Section 3 explains the proposed fault localization method. In Sec. 4 we report four case studies and the comparison of effectiveness between our method and others. Discussions about the proposed method appear

^cCoverage with respect to other criteria such as decision, c-uses, and p-uses [7] are also collected and available for program debugging.

in Sec. 5. Section 6 lists some related studies. The conclusion and future work are presented in Sec. 7.

2. An Overview of the BP Neural Networks

An artificial neural network can be viewed as a mathematical model that can simulate certain behaviors of biological nervous systems [10]. The modern definition of this term is an artificial construct whose behavior is based on that of a network of simple elements called neurons. Neurons are connected together with weights on the connections so that they can process information collaboratively and store the information on these weights. Neural networks have many advantages over other models, such as:

- (1) They have the ability to learn highly complex models due to their expressiveness.
- (2) They are fairly robust to noisy training data. Because the information is distributed among the weights on the connections, a few faults in the training data have little influence on the model.
- (3) They have the ability to adapt themselves to time-variant models.

These characteristics have led to neural networks becoming an important artificial intelligence technique. They have been successfully applied to many areas such as pattern recognition [11], system identification [6], and intelligent control [24]. A BP neural network is a kind of feed forward neural network, in which the neurons are organized in layers, and each neuron can be connected only with the neurons in the next layer. Consequently, no directed cycles exist in such a network. A BP network can learn a complicated nonlinear input-output function from a set of sample data (including inputs and the corresponding expected outputs). Figure 1 shows the structure of a three-layer BP neural network. The data flow in a BP neural network is delivered from the input layer, through hidden layer(s), to the output layer, without any feedback.

An *error back-propagation* algorithm is an iterative procedure typically used to train a BP neural network. Specifically, the procedure adjusts the weights in the network in an iterative fashion so that the resulting network fits the training data well. However, the longer we train the network, the more likely the resulting network will over-fit the training data. Over-fitting is a condition where the neural network achieves high accuracy on the training data but possesses poor generalization capability, owing to the acquisition of patterns that are specific to the training data (e.g., noise). Techniques such as early stopping and Bayesian regularization have been used to mitigate the over-fitting problem [3, 9, 26].

Early stopping can be implemented by separating the input data into two disjoint sets: a training set and a validation set [3]. The neural network is trained using the training set and after each training iteration, the prediction accuracy of the network

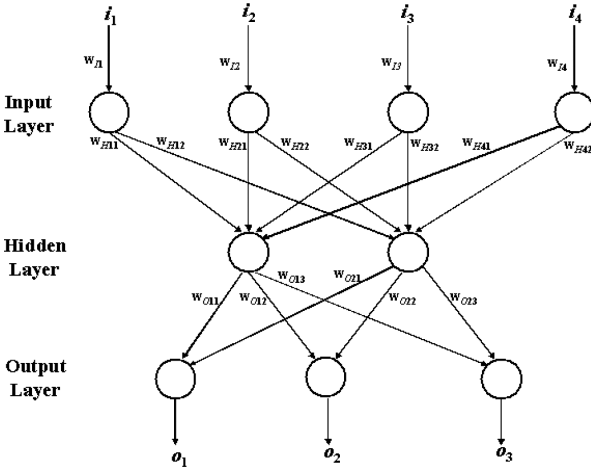


Fig. 1. Structure of a BP neural network.

is measured using the validation set. When the accuracy against the validation set begins to decrease, training is halted.

Modifying the objective function of the neural network can improve generalization, using a process known as regularization [26]. Training feedforward neural networks typically involves minimizing an objective function based on the mean sum of squares of the network errors, similar to the following:

$$F = E_D = \frac{1}{N} \sum_{i=1}^N (t_i - a_i)^2 = \frac{1}{N} \sum_{i=1}^N e_i^2 \tag{1}$$

where N is the number of inputs in the training set, t_i is the expected output of the i th input, a_i is the corresponding neural network response, and e_i is the difference between t_i and a_i (i.e., the error between the expected output and the neural network response). An additional term E_W representing the sum of squares of the network weights is added to the above function to prevent over-fitting by “smoothing” the network weights [9]:

$$F = \alpha E_W + \beta E_D. \tag{2}$$

In this expanded objective function, α and β are parameters to be optimized using a Bayesian framework described by MacKay [22, 23].

We approach the over-fitting problem using a combination of Bayesian regularization and the Levenberg-Marquardt optimization algorithm. When dealing with relatively small data sets, Bayesian regularization typically yields better generalization results than early stopping [26].

After the neural network is set up, the BP algorithm uses a set of sample data to train the network by the following steps:

- (1) Provide the sample inputs to the neural network to generate the actual outputs. This is a forward propagation.

- (2) Compute the error of the network given by Eq. (2). Then, propagate the error backward to the input layer. In this error back-propagation process, the weights on connections are changed to reduce the error.

These steps will be repeated until the network error is small.

In this paper, we use a BP neural network for fault localization for the following reasons:

- (1) BP neural networks have been proved to be broadly applicable models and have been successfully used in the solutions of several problems such as software risk analysis [25], reliability estimation [30], software cost prediction [31], and severity levels of program faults that could be detected [4].
- (2) BP neural networks have the ability to approximate complex nonlinear functions. For example, they can be used to simulate the relationship between statement coverage and execution results (success or failure).
- (3) BP neural networks are trained by a supervised learning algorithm (e.g., an error back-propagation algorithm). When expected outputs are known, a supervised learning algorithm can train the network more accurately and efficiently than an unsupervised learning algorithm can. Since in program debugging, we know the expected output of each sample input (i.e., whether the program execution succeeds or fails with respect to each sample input), a neural network is more suitable for the fault localization problem.

3. Proposed Method

3.1. Fault localization with a BP neural network

Suppose we have a program P with m executable statements and exactly one fault.^d Suppose also that P is executed on n test cases of which k tests are successful and $n - k$ are failed. Table 1 lists notations that are used for the rest of the paper. Figure 2 gives an example of coverage data (statement coverage in this case) and execution results that we need for the proposed fault localization method. Each row contains the statement coverage (1 means the statement is covered and 0 means not covered) and the execution result (0 means the execution is successful and 1 means

Table 1. Symbols used in this section.

m	number of executable statements
n	number of test cases
t	a test case executed on P
c_t	the coverage vector of t
$S(t)$	the set of the executable statements covered by the execution of t
r_t	the execution result of t ("successful" or "failed")
s_i	the i th executable statement of the program

^dRefer to Sec. 5.3 for a discussion of program with multiple bugs.

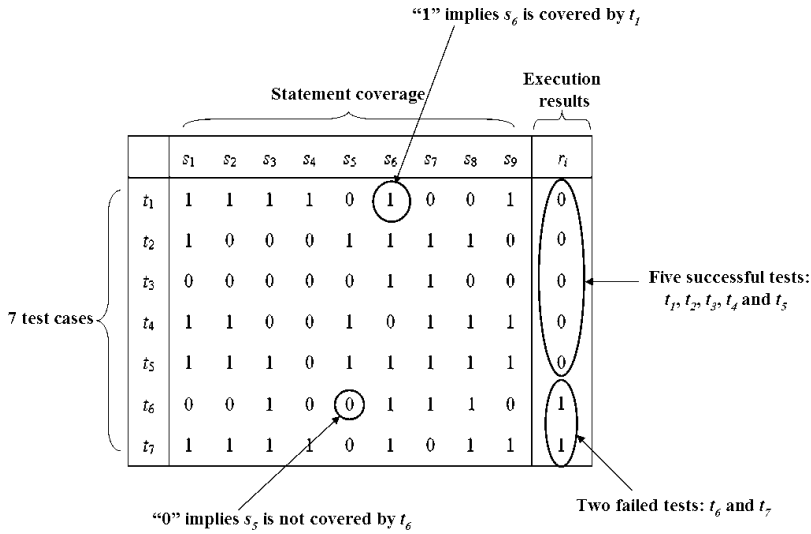


Fig. 2. Sample coverage data and execution results.

failed) of a test case. For example, statement s_6 is covered by a successful test t_1 , and statement s_5 is not covered by a failed test t_6 .

Vector c_{t_i} denotes the coverage data obtained from the execution of test case t_i . For example, $c_{t_1} = (1, 1, 1, 1, 0, 1, 0, 0, 1)$ extracted from the first row in Fig. 2 gives the statement coverage of the execution of t_1 . We refer to c_{t_i} as the *coverage vector* of t_i . Assume there is a set of *virtual test cases* v_1, v_2, \dots, v_m whose coverage vectors are c_{v_1}, \dots, c_{v_m} , where

$$\begin{bmatrix} c_{v_1} \\ c_{v_2} \\ \vdots \\ c_{v_m} \end{bmatrix} = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{bmatrix} \quad (3)$$

The execution of virtual test case v_i ($i = 1, 2, \dots, m$) covers only one statement s_i . If the execution of v_i fails, the probability that the fault is contained in s_i is high. This implies that during the fault localization, we should first examine the statements whose corresponding virtual test case fails. However, we cannot find v_1, v_2, \dots, v_m in the real world. In order to estimate the execution results of these virtual test cases, we build a three-layer BP neural network with m input-layer neurons, three hidden-layer neurons^e and one output-layer neuron, and train it using c_{t_i} and r_{t_i} ($i = 1, 2, \dots, n$) as the input data and corresponding expected output, respectively. The structure of such a BP neural network is shown in Fig. 3. The *transfer* functions of the neurons are set to the sigmoid function $y = 1/(1+e^{-x})$.

^eRefer to Sec. 5.2 for a discussion on the number of neurons in the hidden layer.

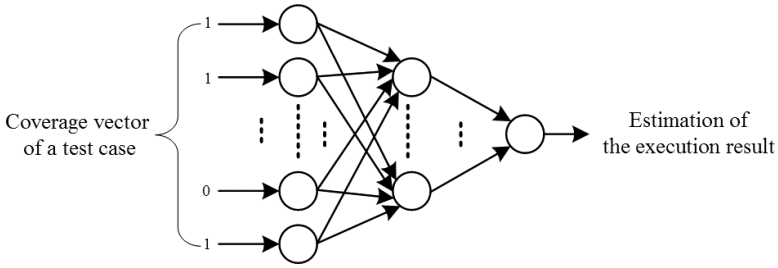


Fig. 3. The BP neural network used in our method.

When the coverage vector c_{v_i} ($i = 1, 2, \dots, m$) of a virtual test case v_i is input to the trained neural network, the output of the neural network (denoted by r'_{v_i}) is an estimation of the execution result of v_i . The value of r'_{v_i} is between 0 and 1. The larger the value of r'_{v_i} , the more likely it is that s_i contains the bug. We can treat r'_{v_i} as the suspiciousness of s_i in terms of its likelihood of containing the bug.

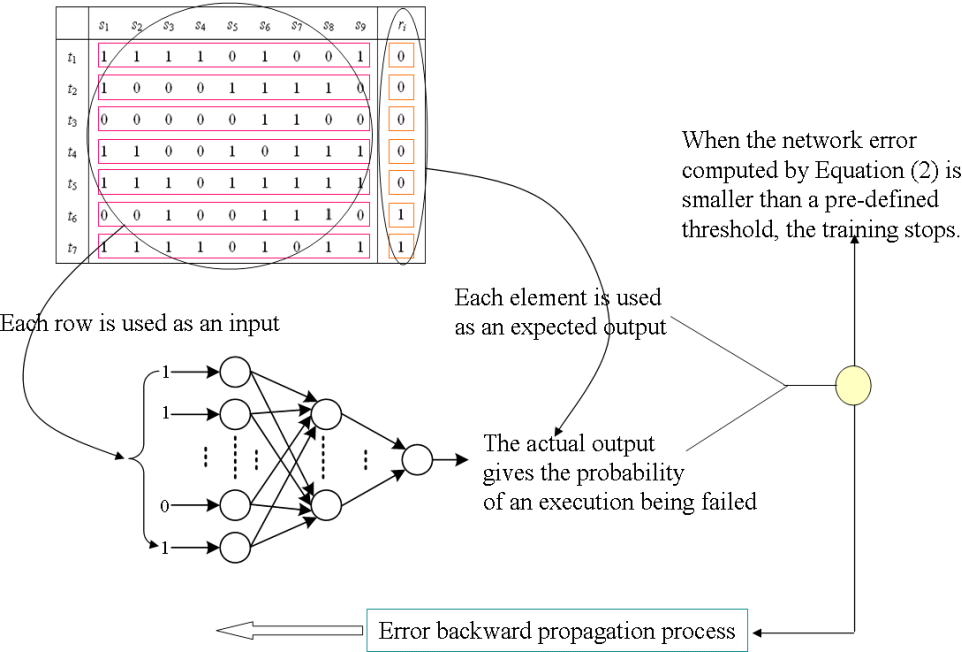
Figure 4 shows the process of computing the suspiciousness of each executable statement using a BP neural network. We summarize this part as follows:

Procedure I

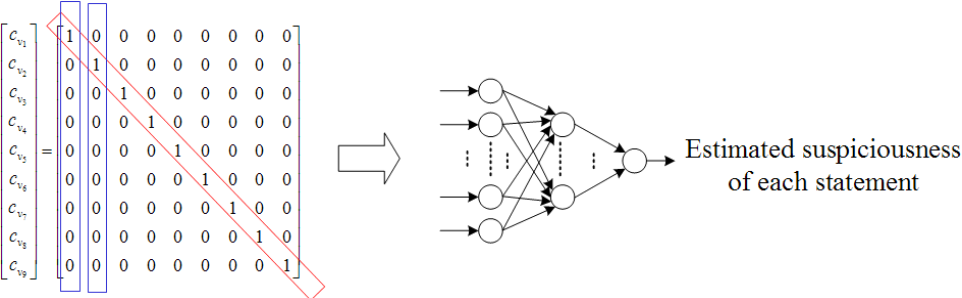
- (1) Build up a BP neural network with m input-layer neurons, three hidden-layer neurons, and one output-layer neuron. The *transfer* function of each neuron is set to the sigmoid function $y = 1/(1 + e^{-x})$.
- (2) Train the neural network using c_{t_i} and r_{t_i} ($i = 1, 2, \dots, n$) as the input data and the corresponding expected output data, respectively. The over-fitting problem is handled by the Bayesian regularization.
- (3) Input c_{v_1}, \dots, c_{v_m} in Eq. (3) into the trained neural network and get the outputs $r'_{v_1}, r'_{v_2}, \dots, r'_{v_m}$.
- (4) Rank s_1, s_2, \dots, s_m based on $r'_{v_1}, r'_{v_2}, \dots, r'_{v_m}$ in descending order and examine the statements one by one from the top until the fault is located.

Let us demonstrate our method through an example. Suppose we have a program with nine statements ($m = 9$) and one fault in statement s_8 . We executed seven test cases ($n = 7$) on it, in which two of them failed. Figure 2 shows the coverage data and execution result of each test case. We do the following:

- Build up a BP neural network with nine inputs neurons, one output neuron and three middle-layer neurons. The *transfer* functions of neurons are set to the sigmoid function.
- Train the BP neural network with the coverage data. The first input vector is (1, 1, 1, 1, 0, 1, 0, 0, 1) and the expected output is 0. The second input vector is (1, 0, 0, 0, 1, 1, 1, 1, 0) and the expected output is 0, and so on. Repeat training the network with these data until the network error is small enough (e.g., 10^{-3} in our case).



Part (a)



Part (b)

Fig. 4. (a) Train a BP neural network and (b) estimate the suspiciousness of each statement using a trained BP neural network.

- Input the coverage vectors of the virtual test cases into the trained neural network. The output with respect to each statement is shown in Table 2.
- After ranking the statements based on their suspiciousness, we get $s_4, s_3, s_9, s_8, s_6, s_1, s_5, s_7, s_2$. That is, s_4 is most likely to contain the fault.
- Examine the statements one by one in order of $s_4, s_3, s_9, s_8, s_6, s_1, s_5, s_7, s_2$. The fault will be found when s_8 is examined. In this example, we examined four statements before we found the location of the fault.

Table 2. Actual output with respect to each statement.

Statement	Output	Statement	Output	Statement	Output
s_1	0.11187	s_4	0.72339	s_7	0.01806
s_2	0.01254	s_5	0.03651	s_8	0.20847
s_3	0.49732	s_6	0.18736	s_9	0.21079

3.2. Reduce the number of candidate suspicious statements

Our BP neural network-based fault localization method can be further improved by considering execution slices of failed tests. An execution slice with respect to a given test case in our study contains the set of statements executed by this test.^f Such a slice can be constructed very easily if we know the coverage of the test because the corresponding execution slice of the test can be obtained simply by converting the coverage data collected during the testing into another format. More specifically, instead of reporting the coverage percentage, it reports which statements are executed. Since many of the statements are unrelated to the faults, we can develop additional heuristics to reduce the number of candidate suspicious statements. By doing so, we also reduce the number of statements whose suspiciousness have to be computed using our neural network. This makes our method more efficient.

In general, the fault should be covered by failed tests, or at least related to the statements covered by failed tests. This implies the most suspicious statements are the statements covered by all the failed executions. Let S_I denote the set of the statements covered by all failed executions. We have

$$S_I = S(t_{f_1}) \cap S(t_{f_2}) \cap \cdots \cap S(t_{f_{n-k}})$$

where $t_{f_1}, t_{f_2}, \dots, t_{f_{n-k}}$ are the failed test cases and $S(t_{f_i})$ is the execution slice of t_{f_i} , i.e., the set of statements covered by t_{f_i} . In special cases, S_I does not contain the faulty statement(s). A good solution to this problem is to find a failed test t_{f_M} which covers the fewest statements and examine the un-checked statements covered by t_{f_M} (i.e., those in the execution slice of t_{f_M} but not in S_I). For simplicity, let us use S_M as the set of the statements covered by t_{f_M} (i.e., $S_M = S(t_{f_M})$). Based on the above discussion, when we are looking for a fault, the statements in S_I should first be examined; if the faulty statement is not there, the statements in $S_M - S_I$ should be examined next. For example, with respect to the sample in Fig. 2, $S_I = S(t_6) \cap S(t_7) = \{s_3, s_6, s_8\}$, $S_M = S(t_6) = \{s_3, s_6, s_7, s_8\}$, and $S_M - S_I = \{s_7\}$. This implies that we should first search for the fault in s_3, s_6 and s_8 . The order of these three statements to be examined is decided by their suspiciousness. If the fault is not in these three statements, s_7 is the next statement that should be examined. In our example, we only had to examine three statements before we found the

^fAn execution can also represent the set of all basic blocks, decisions, c-uses, or p-uses executed by this test, if necessary [35].

location of the fault. In other words, with the additional heuristic discussed above, the number of statements that had to be examined is reduced from four to three.

Integrating this execution slice-based heuristic with the neural network-based method discussed in Sec. 3.1, we summarize our fault localization method as follows:

- Step 1:* Get the intersection of all failed execution slices (S_I) and the smallest failed execution slice (S_M).
- Step 2:* Apply Procedure I (in Sec. 3.1) to the statements in S_I to examine whether the fault is in these statements. If the fault location is found, go to Step 4.
- Step 3:* Apply Procedure I to the statements in $S_M - S_I$. That is, compute the suspiciousness of these statements, rank them in descending order based on their suspiciousness and examine them one by one from the top until the fault is found. Refer to Sec. 5.4 for the discussion of two special cases.
- Step 4:* Stop.

For discussion purposes, we refer to this method as “BPNN method” in the following sections.

4. Four Case Studies

Four studies using the Siemens suite, the Unix suite, grep and gzip were conducted to demonstrate the effectiveness of using the BPNN method in fault localization. Such effectiveness is measured by a score *EXAM* in terms of the percentage of executable statements that *have to be examined* until the first statement containing the bug is reached. A similar score defined as the percentage of the program that *need not* be examined to find a faulty statement is used in [8, 16, 28]. Although these two scores provide the “same” information, the *EXAM* score seems to be more direct and easier to understand. The effectiveness of different fault localization methods can be compared based on *EXAM*. For a faulty version Ω , if the *EXAM* assigned to Ω by method *A* is smaller than that assigned by method *B* (that is, method *A* can guide the programmer to the fault in Ω by examining less code than method *B*), then *A* is more effective than *B* for locating the bug in Ω . If there is more than one faulty version, then *A* is more effective than *B* if *A* assigns a smaller *EXAM* to more faulty versions than *B*.

For studies on the Siemens suite, gzip and grep, all the programs (implemented in the C language), test cases, and defect data used in our studies are downloaded from the web sites <http://www-static.cc.gatech.edu/aristotle/Tools/subjects> at Georgia Tech and <http://sir.unl.edu/portal/index.html> at the University of Nebraska — Lincoln. These programs can be good candidates for a benchmark comparison between the effectiveness of different fault localization methods. Similar to [16], multi-line statements are combined as one source code line so that they will be counted only as one executable statement. Each faulty version has exactly one bug and is executed against all its corresponding available test cases. For the Siemens and Unix suites, all executions (correct and faulty versions) were on a PC with a 2.13 GHz Intel Core

2 Duo CPU and 8 GB physical memory. The operating system is SunOS 5.10 (a.k.a. Solaris 10) and the compiler is GCC 3.4.3. For `grep` and `gzip`, the executions were on a Sun-Fire-280R machine with SunOS 5.10 as the operating system and GCC 3.4.4 as the compiler. A revised version of χ Suds [38] which could collect runtime trace correctly even if a program execution was crashed due to a “segmentation fault” was used to help us record how a program was executed by the failed and the successful tests. The information collected was then used to decide how many successful tests and how many failed tests cover each statement, respectively. The success or failure of an execution was determined by comparing the outputs of the faulty version and the correct version of a program.

Although a bug may span multiple statements which may not be contiguous or even multiple functions, the fault localization stops when the first statement containing the bug is reached. This is because our focus is to help programmers find a *starting* point to fix a bug rather than provide the complete set of code that has to be modified/deleted/added with respect to each bug. We also assume perfect bug detection, that is, a bug in a statement will be detected by a programmer if the statement is examined. If such perfect bug detection does not hold, then the number of executable statements that need to be examined in order to find the bug may increase. This concern applies to all the fault localization methods discussed in Sec. 6.

Results of our method are compared with those of the Tarantula method [16] for their effectiveness. The reason why the Tarantula method is chosen is because it was reported that Tarantula is more effective in fault localization (in terms of less code that has to be examined before the first faulty statement is found) than other methods such as set union, set intersection, nearest neighbor, and cause transitions [8, 28] with respect to the Siemens suite. Hence, if we can show that BPNN is more effective than Tarantula, then BPNN is also more effective than those to which Tarantula is superior. For the Siemens suite, since the number of faulty versions and the number of test cases in our study are slightly different from those in [16] (referring to Sec. 4.1 for a more detailed discussion) and statistics such as fault revealing behavior and statement coverage of each test can vary under different compilers, operating systems, and hardware platforms, it is necessary to recompute the effectiveness of Tarantula using our test data and their ranking mechanism to arrive at a fair comparison. More precisely, we instrumented the programs, reran all the tests, collected the coverage information, identified which statements were executed by which test(s), determined whether each execution failed or succeeded, and computed the suspiciousness of each statement. We also performed a careful cross check between the reported effectiveness in [16] and the recomputed effectiveness whenever possible to ensure that the effectiveness of the Tarantula method was computed correctly. For the Unix suite, `grep` and `gzip`, we did not find the effectiveness of Tarantula ready to be used; therefore, we computed such effectiveness using their ranking mechanism on our data.

4.1. The Siemens suite

Table 3 gives a summary of the seven programs in the Siemens suite including the name and a brief description of each program, the number of faulty versions, LOC (the size of the program before any non-executable code is removed), the number of executable statements, and the number of test cases. More descriptions of the Siemens suite can be found in [14].

Of the 132 faulty versions, three were not used in our study because (1) none of the test cases downloaded can reveal the fault in version 9 of “schedule2,” and (2) the faults in versions 4 and 6 of “print_tokens” are in the header files instead of in the C files. Compared with a previous study [16], ten faulty versions were excluded. Note that versions 27 and 32 of “replace” and versions 5, 6, and 9 of “schedule” were not used because the tool used in [16] (gcc with gcov) does not dump its coverage before the program crashes. We did not have this problem because we used a revised version of χ Suds [38] which can collect runtime trace correctly even with a “segmentation fault.” Note also the number of test cases downloaded from Georgia Tech at <http://www.static.cc.gatech.edu/aristotle/Tools/subjects> (i.e., the number of test cases used in our study) is slightly larger than the number of test cases reported in [16]. In our study, each faulty version of print_tokens, print_tokens2, replace, schedule, schedule2, tcas, and tot_info was executed against 4130, 4115, 5542, 2650, 2710, 1608, and 1052 test cases, respectively. As a result, version 10 of “print_tokens” and version 32 of “replace” could be used in our study even though they had to be excluded in [16] because no test cases fail.

Different executable statements may have the same suspiciousness computed by using the Tarantula method. This gives two different types of effectiveness: the “best” and the “worst.” The “best” effectiveness assumes that the faulty statement is the first to be examined among all the statements of the same suspiciousness. For instance, supposing there are ten statements of the same suspiciousness of which one is faulty, the “best” effectiveness is achieved if the faulty statement is the first to be examined of these ten statements. Similarly, the “worst” effectiveness occurs if the faulty statement is the last to be examined of these ten statements. This is not the case for BPNN. In all our studies, we found no two statements with the same suspiciousness computed by using the BPNN method. Hence, there

Table 3. Summary of the Siemens suite.

Program	Description	Number of faulty versions	LOC	Number of executable statements	Number of test cases
print_tokens	Lexical analyzer	7	565	175	4130
print_tokens2	Lexical analyzer	10	510	178	4115
replace	Pattern replacement	32	563	216	5542
schedule	Priority scheduler	9	412	121	2650
schedule2	Priority scheduler	10	307	112	2710
tcas	Altitude separation	41	173	55	1608
tot_info	Information measure	23	406	113	1052

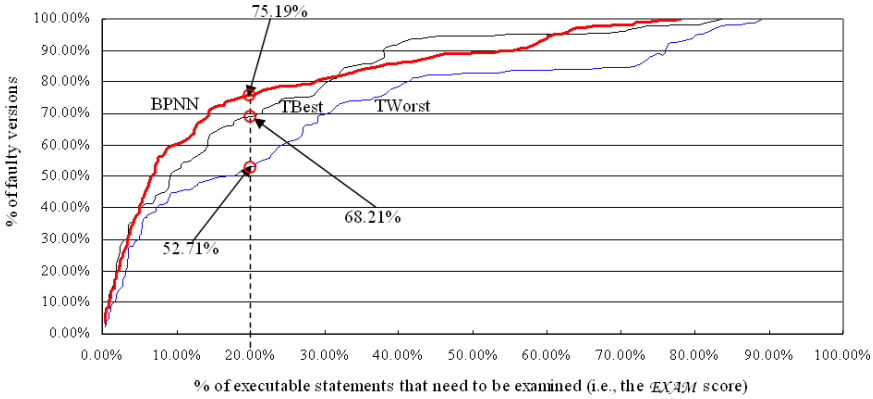


Fig. 5. Effectiveness comparison between the BPNN method and the Tarantula method for the Siemens suite.

is just one level of effectiveness for BPNN instead of the so-called “best” and “worst” effectiveness.

Figure 5 gives the effectiveness of Tarantula and BPNN methods. The red curve labeled BPNN is for the BPNN method and those labeled TBest (in black) and TWorst (in blue) are for the best and the worst effectiveness of the Tarantula method.^{g,h} The curves in this figure are drawn by connecting all the individual data points collected in our study. This is different from the curves in [16] where for each segment, a straight line is drawn between the beginning and the ending points of that segment. For a given x value, its corresponding y value is the percentage of the faulty versions whose *EXAM* score is less than or equal to x . For example, the three points in Fig. 5 indicate that the bug in 75.19% of the faulty versions can be located by examining less than/equal to 20% of all the executable statements if the BPNN method is used. The corresponding percentages of faulty versions for TBest and TWorst are 68.21% and 52.71%, respectively. Clearly, BPNN is not only more effective than TWorst but also more effective than TBest in this case. The same observation also applies to many other points with different x values.

Table 4 presents the effectiveness comparison from a different perspective by showing the number of faulty versions that BPNN is more effective than, as effective as, and less effective than TBest and TWorst. From this table, we observe that BPNN is more effective than or as effective as TBest for 58.14% of the 129 faulty versions in the Siemens suite. This percentage goes up to 83.72% for TWorst.

^gSince curves in Fig. 5 are displayed in different colors, it is better viewed in color. This is also the case for many other figures in the paper.

^hThe experimental data for the Siemens suite in an earlier version of this paper presented at the 19th International Conference on Software Engineering and Knowledge Engineering (SEKE 2007) were collected using BPNN without considering the over-fitting problem, whereas data reported here were collected using BPNN with the Bayesian regularization to mitigate the over-fitting.

Table 4. Pairwise comparison between the BPNN method and the Tarantula method for the Siemens suite.

	BPNN versus TBest	BPNN versus TWorst
More effective	49	91
Same effectiveness	26	17
Less effective	54	21

Table 5. Total number of executable statements examined for all 129 faulty versions in the Siemens suite.

	BPNN	TBest	TWorst
Number of executable statements	1919	2453	3311

A third way to compare the effectiveness of BPNN and Tarantula is based on the total number of executable statements that have to be examined by each method for all 129 faulty versions in the Siemens suite. Since the number of statements examined is closely related to the cost of fault localization, the more statements examined, the higher the cost. So, it is reasonable to use this measure to compare the effectiveness of BPNN and Tarantula. Referring to Table 5, the total number of statements examined by the BPNN method is 1919, whereas the total number of statements examined using Tarantula is between 2453 and 3311. This means we can examine 21.77% to 42.04% fewer statements if we use BPNN instead of Tarantula.

4.2. The Unix suite

A suite of ten Unix utility programs written in C was used in our second case study. Since these programs have been so thoroughly used, they can serve as a reliable basis for evaluating the behavior of fault injected programs derived from them. Table 6 gives the summary of this suite including the name and a brief description of each program, the number of faulty versions, LOC, the number of executable statements and the number of test cases. The number of faulty versions of some programs used in our study is slightly different from that reported in [34] for a few reasons. First, the execution environments are different. Second, a revised version of χ Suds was used which could collect runtime trace correctly even if a program execution was crashed due to a “segmentation fault.” More descriptions of the test case generation, fault set, and erroneous program preparation can be found in [34].

Figure 6 shows the effectiveness of BPNN and Tarantula for the Unix suite where all the legends, the horizontal axis, and the vertical axis have the same meaning as in Fig. 5. We observe that for any x value, the BPNN curve has a higher y value than TWorst. This implies that BPNN has a higher percentage of the faulty versions than TWorst for any *EXAM* score. It also implies that BPNN is more effective than TWorst. When compared with TBest, the above observation is true for many x values, i.e., BPNN is more effective than TBest with respect to many *EXAM* scores.

Table 6. Summary of the Unix suite.

Program	Description	Number of faulty versions	LOC	Number of executable statements	Number of test cases
Cal	Print a calendar for a specified year or month	20	202	88	162
Checkeq	Report missing or unbalanced delimiters and .EQ/.EN pairs	20	102	57	166
Col	Filter reverse paper motions from nroff output for display on a terminal	30	308	165	156
Comm	Select or reject lines common to two sorted files	12	167	76	186
Crypt	Encrypt and decrypt a file using a user supplied password	14	134	77	156
Look	Find words in the system dictionary or lines in a sorted list	14	170	70	193
Sort	Sort and merge files	21	913	448	997
Spline	Interpolate smooth curve based on given data	13	338	126	700
Tr	Translate characters	11	137	81	870
Uniq	Report or remove adjacent duplicate lines	17	143	71	431

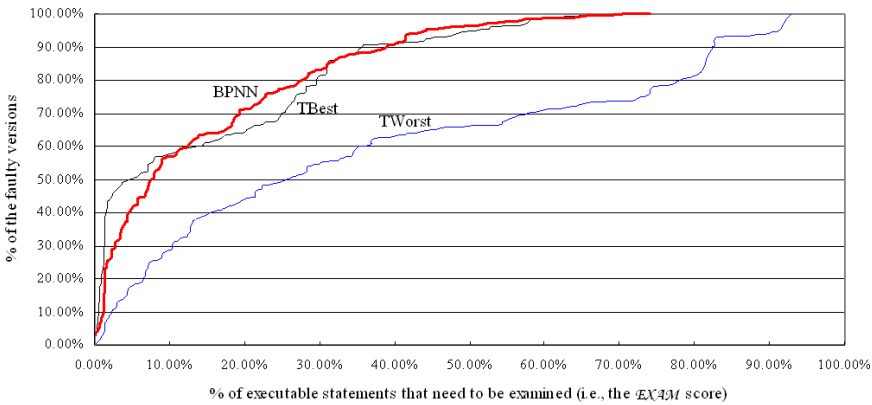


Fig. 6. Effectiveness comparison between the BPNN method and the Tarantula method for the Unix suite.

From the pairwise point of view, we observe, based on Table 7, that BPNN is more effective than or as effective as TBest for 53.49% of the 172 faulty versions in the Unix suite. This percentage increases to 94.19% for TWorst.

Table 8 gives the total number of executable statements that have to be examined for all 172 faulty versions in the Unix suite by BPNN and Tarantula. It suggests that we can examine 16.32% to 63.10% fewer statements if we use BPNN instead of Tarantula.

Table 7. Pairwise comparison between the BPNN method and the Tarantula method for the Unix suite.

	BPNN versus TBest	BPNN versus TWorst
More effective	72	148
Same effectiveness	20	14
Less effective	80	10

Table 8. Total number of executable statements examined for all 172 faulty versions in the Unix suite.

	BPNN	TBest	TWorst
Number of executable statements	2815	3364	7629

4.3. The grep program

The grep program searches a file for a pattern. The source code of version 2.2 was downloaded from <http://sir.unl.edu/portal/index.html> at the University of Nebraska — Lincoln. It has 12653 lines of code including comments, blank lines, etc. and 3306 executable statements. Also downloaded were a suite of 470 test cases and 18 bugs of which 14 could not be detected in our environment by any test case in the suite. Two additional bugs used in [19] were also included. We followed a similar approach as Liu *et al.* did in [19] to create 13 more bugs of different types. Altogether, there are 19 faulty versions each of which contains exactly one bug. A list of these bugs is available upon request.

A similar comparison between the effectiveness of BPNN and Tarantula as we did for the Siemens and Unix suites was also conducted for the grep program. All the legends, figures and tables have their usual meaning. From Fig. 7, we observe that BPNN is not only more effective than TWorst but also more effective than TBest for any given *EXAM* score. This is a very powerful indication that BPNN is better than Tarantula in fault localization with respect to the grep program.

From Table 9, we observe that BPNN is more effective than or as effective as TBest for 78.95% of the 19 faulty versions of the grep program. This percentage increases to 100% for TWorst.

Table 10 gives the total number of executable statements that have to be examined for all 19 faulty versions of the grep program. BPNN examines 1540 statements, or 26.58% of the 5793 statements examined by TBest and 19.71% of the 7812 statements examined by TWorst.

4.4. The gzip program

The gzip program reduces the size of named files using the Lempel-Ziv coding. From the web site <http://sir.unl.edu/portal/index.html>, we downloaded version 1.1.2 with 16 seeded bugs and 217 test cases. We discarded six test cases which could not be executed in our environment; nine bugs were also not used since none of them had

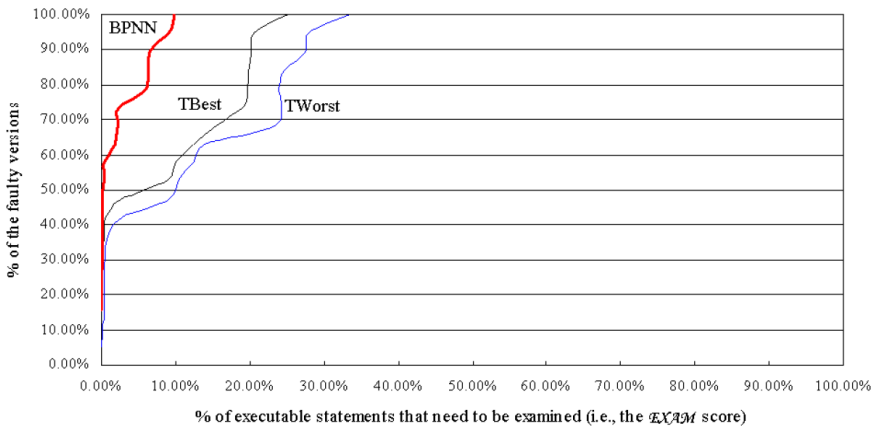


Fig. 7. Effectiveness comparison between the BPNN method and the Tarantula method for the grep program.

Table 9. Pairwise comparison between the BPNN method and the Tarantula method for the grep program.

	BPNN versus TBest	BPNN versus TWorst
More effective	14	17
Same effectiveness	1	2
Less effective	4	0

Table 10. Total number of executable statements examined for all 19 faulty versions of the grep program.

	BPNN	TBest	TWorst
Number of executable statements	1540	5793	7812

any failures among the remaining 211 test cases. The program has 6573 lines of code and 1670 executable statements. After examining the 16 seeded bugs, we followed a similar approach to inject 21 additional bugs of different types. This makes a total of 28 faulty versions and each version contains exactly one fault. A list of these bugs is available upon request.

Similarly, we compare the effectiveness of BPNN and Tarantula for the gzip program. Figure 8 shows that BPNN is more effective than TWorst and TBest for any given *EXAM* score. This gives us a clear suggestion that BPNN is better than Tarantula in fault localization with respect to the gzip program used in this study.

From Table 11, we observe that BPNN is more effective than or as effective as TBest for 57.14% of the 28 faulty versions of the gzip program. Compared with TWorst, BPNN is 100% more effective.

Similarly, we can compare the total number of executable statements that have to be examined for all 28 faulty versions of the gzip program. Referring to Table 12,

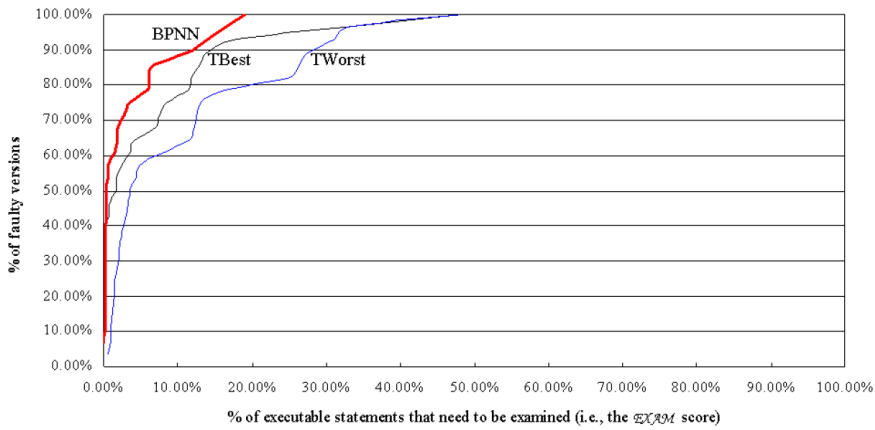


Fig. 8. Effectiveness comparison between the BPNN method and the Tarantula method for the gzip program.

Table 11. Pairwise comparison between the BPNN method and the Tarantula method for the gzip program.

	BPNN versus TBest	BPNN versus TWorst
More effective	14	28
Same effectiveness	2	0
Less effective	12	0

Table 12. Total number of executable statements examined for all 28 faulty versions of the gzip program.

	BPNN	TBest	TWorst
Number of executable statements	1581	3110	5032

we can examine 49.16% to 68.58% fewer statements if we use BPNN instead of Tarantula.

4.5. Overall observations

In general, we would expect that the effectiveness of the Tarantula method lies between TBest and TWorst as it is impractical to assume that the best effectiveness could be achieved every time. In fact, our intuition suggests that when the size of the program being debugged increases, it is more likely for Tarantula to group more statements together with the same suspiciousness. This also makes it even less likely to have the faulty statement to be the first one being examined among all the statements with the same suspiciousness. With this understanding and also based on the comparisons using the experimental data collected from our studies on the Siemens suite, the Unix suite, the grep and gzip programs, it is clear that

BPNN is more effective in fault localization (i.e., examining less code before the first faulty statement is located) than Tarantula.

5. Discussion

In this section, we discuss some important aspects related to the BPNN fault localization method.

5.1. Multiple statements with the same suspiciousness

One significant disadvantage of using the Tarantula method in fault localization is that the same suspiciousness may be assigned to more than one statement. In fact, when the size of the program being examined increases, the likelihood of different statements having the same suspiciousness also increases. As a result, for large-sized programs we may find groups of many statements such that those in the different groups have different suspiciousness, but those in the same group have the same suspiciousness. This problem is because for a given statement s , Tarantula computes its suspiciousness as

$$\frac{\text{failed}}{\text{totalfailed}} \bigg/ \left(\frac{\text{failed}}{\text{totalfailed}} + \frac{\text{successful}}{\text{totalsuccessful}} \right)$$

where “failed” and “successful” are the number of failed and successful tests that execute s , and “totalfailed” and “totalsuccessful” are the total number of failed and successful tests in the entire test suite. A close examination shows that two statements s_1 and s_2 will have the same suspiciousness assigned by Tarantula if they are executed by the same number of failed tests (but not necessarily exactly the same failed tests) and the same number of successful tests (but not necessarily exactly the same successful tests). Clearly, the probability of having two such statements increases when the size of the program increases. This is why large-sized programs may have more groups with each group having more statements with the same suspiciousness than small-sized programs.

It is not reasonable to assume that the faulty statement will always be the first to be examined among a group of statements with the same suspiciousness. Similarly, it is also impractical to assume such a statement will always be examined the last among that group. Hence, in practice we should expect that the effectiveness of the Tarantula method lies between TBest and TWorst. Nevertheless, it is also expected that when the size of the program being examined becomes larger, the chance to have TBest as the actual effectiveness goes down.

The BPNN method does not suffer from this problem in our studies. Referring to our experimental data, we notice that even if two statements are executed by the same number of failed and successful tests, they still have different suspiciousness assigned by BPNN. To conclude, BPNN can assign different suspiciousness to different statements with less duplication than Tarantula. This indicates that the BPNN

method may provide a more precise guideline than the Tarantula method in terms of which suspicious statement should be examined next for locating a program bug.

5.2. The number of hidden-layer neurons

There is no standard solution about how many hidden-layer neurons should be used. For the BPNN method reported here, three neurons are used in the hidden layer. This configuration works well in our four case studies. We have also tried higher numbers of neurons in the hidden layer. The results are similar to those obtained when using three hidden-layer neurons, but the training of the BP neural network takes much more time.

5.3. Programs with multiple bugs

Although in this paper BPNN is only applied to programs with a single bug, it can also be applied to program with multiple bugs. To do so, we need to first cluster failed executions according to the bugs that cause them [27]. In other words, failed test cases are grouped into fault-focusing clusters such that those in the same cluster are related to the same fault [15]. Failed tests in each cluster are then combined with the successful tests for debugging a single fault. As a result, the BPNN method discussed here can be extended for debugging programs with multiple bugs. Different methods have been proposed to cluster failed executions or failed tests. For example, Zheng *et al.* performed a clustering on failing executions based on fault-predicting predicates [41], and Liu and Han used two distance measures to cluster failed test cases [20].

A future goal of our research is to determine whether it is possible to simply apply the BPNN method iteratively to locate one bug on each iteration without needing to cluster, or is an appropriate cluster on failed test executions required for programs with multiple bugs? If the latter, we will also study how exactly the clustering should be conducted.

5.4. Two special cases

If the current test execution fails only because a previous test did not set up an appropriate execution environment, the two test cases should be bundled as a single failed test. For example, consider a test case t_α that, due to a bug in the program, incorrectly modifies values at certain memory locations but otherwise successfully completes execution. Test case t_β , executed consecutively, fails only because it references the incorrect memory values written by t_α . In this situation, t_α and t_β should be combined as a single failed test.

Another potential argument is that bugs caused by missing code cannot be located using BPNN. Logically, if the code related to the bug is not even in the program, it cannot be identified as a suspicious statement by BPNN. This may be true, but the absence of such code might create some unexpected effects later in the

program, such as the traversal of the wrong branch at a decision statement. Missing code may therefore trigger the assignment of high suspiciousness to unexpected statements, which could lead to the deduction that the code necessary for choosing the correct branch has been omitted.

5.5. *Expandability of BPNN*

An important advantage of the proposed BPNN method is its expandability. The proposed method is based on a neural network, which does not depend on a particular mathematical model. Given the statement coverage data and the execution results of testing, the BPNN method can identify the relationship between the input and output and thus the suspiciousness of each statement. The coverage data used in our method can be changed, for example, to the coverage data of a module with respect to various adequacy criteria (e.g., basic blocks, decisions, c-uses, p-uses, mutation, etc.) where a module can be a “function” of a C program or a “class” of a Java or C++ program. Using the coverage at the module level of each test case and other structure related procedural/object-oriented static metrics as well as the corresponding result of test execution, a BP neural network can be trained to identify fault-prone modules — the modules which are likely to contain program bugs. This can be very useful because due to the resource constraints and time pressure, not all modules of a software system can be extensively reviewed and tested. Statistical data have shown that Pareto’s principle applies to software, that is, approximately 20% of a software system is responsible for 80% of the faults [5]. This clearly indicates the need for development of an effective method to help practitioners identify only a small percentage of the modules in a software system as fault-prone. In the last decade, many studies have been reported on investigating the predictive capabilities of various procedural and/or object-oriented metrics on fault-proneness. Different sets of metrics have been proposed for the construction of predictive models that can identify fault-prone software modules. Many of them suffer from common drawbacks such as only using static metrics without considering dynamic testing information like the test coverage of each module. Another problem is that since there is no single set of metrics or single model that fits all software, many statistical models for predicting fault-prone software modules are applicable to only a limited number of applications. The extended BPNN method can overcome both problems. It uses static and dynamic data for fault-proneness prediction. Also, the self-learning capability makes BP networks trainable for various applications.

Execution count data (the number of times each statement is executed by a test) can also be used in the BPNN method to localize the fault by replacing the 0/1’s in the coverage data with the actual execution counts. Other program spectra [12] are also feasible to be used in our proposed BPNN method, such as Branch Hit Spectra, Branch Count Spectra, Path Hit Spectra, Path Count Spectra, Data-dependence Hit Spectra, and Data-dependence Count Spectra.

6. Related Studies

Jones and Harrold present a fault localization method named Tarantula in [16]. It ranks all the executable statements in a program by their suspiciousness of containing the bug, which is estimated by an experimental formula based on statement coverage collected in execution of both successful and failed tests. Developers can examine the statements one by one in descending order of their suspiciousness to find the location of program bugs.

Renieris and Reiss [28] propose a nearest neighbor debugging method. It finds the most similar successful execution to a given failed execution by the distance between two executions. The code in the difference set between the failed execution and its most similar successful execution should first be examined. If the fault is not there, the code in the adjacent node of the examined nodes in the program dependence graph need to be examined until the fault is localized. Set union method and set intersection method are also presented in [28]. Set union method examines the code that is executed by a failed test but not by any of the successful tests. Such code is more suspicious of containing bugs. Set intersection method reduces the size of the code that needs be examined by excluding the code that is executed by all the successful tests but not by the failed tests. Cleve and Zeller [8] report a program state-based debugging approach, cause transition, to identify the locations and times where a cause of failure changes from one variable to another. This approach is based on their previous research on delta debugging [39, 40].

There are also slicing-based methods for locating program bugs. Static slicing proposed by Weiser [32] is one of these. Reduction of the debugging search domain via slicing is based on the idea that if a test case fails due to an incorrect variable value at a statement, then the defect should be found in the static slice associated with that variable-statement pair. We can therefore confine our search to the slice rather than looking at the entire program [33]. Lyle and Weiser extended the above approach by constructing a program dice (as the set difference of two groups of static slices) to further reduce the search domain for possible locations of a fault [21]. Other methods based on dynamic slicing [1, 17] and execution slicing [2, 35, 36] have also been proposed to improve static slice-based fault localization methods.

Researchers have also proposed various statistical debugging methods including an algorithm by Liblit *et al.* [18] for isolating bugs in the programs with instrumented predicates at particular points, the SOBER method to rank suspicious predicates by Liu *et al.* [19], and a crosstab-based method to prioritize executable statements based on their likelihood of containing program bugs by Wong *et al.* [37]. Some recent studies on fault localization such as Jones *et al.* [15], Liu and Han [20], and Zheng *et al.* [41] have a focus on programs with multiple bugs. They use different approaches to cluster failed executions and/or failed test cases.

7. Conclusion and Future Work

A BP neural network-based method (BPNN) is presented for effective fault localization. The network is first trained by using the statement coverage of each test case and the corresponding execution result (success or failure). The Bayesian regularization method is also used to mitigate the over-fitting problem. Coverage vectors of a set of virtual test cases each of which covers only one statement in the program are input into the trained BP network. The outputs are the suspiciousness of executable statements. Such statements are ranked in descending order based on their suspiciousness. They are examined from the top of the rank one by one for the possible location of program bugs. This is because statements with a higher suspiciousness should be examined before those with a lower suspiciousness, as the former are more likely to contain program bugs than the latter. Execution slices are used to reduce the number of candidate suspicious statements, which can improve the performance of BPNN.

Four case studies using different programs (the Siemens suite, the Unix suite, grep and gzip) were conducted to demonstrate the effectiveness of BPNN on fault localization. The results of these studies indicate that BPNN is more effective than Tarantula (a popular fault localization method which uses the same set of input data as BPNN) because the former examines less code than the latter before the first faulty statement is found. Nevertheless, our conclusion may not be generalized to arbitrary programs without more evaluations. To do so, additional case studies on programs from different application domains are to be used in our next step of study.

Our future objectives also include how to combine the information of different program spectra to improve the performance of fault localization and how to apply BPNN to identify fault-prone modules as explained in Sec. 5.5 (expandability of BPNN). Finally, for programs with multiple bugs, we will explore whether it is possible to apply the BPNN method iteratively to locate one bug on each iteration (i.e., possibly identify multiple bugs without needing to cluster). If clustering is deemed necessary in this situation, we will also study exactly how the clustering should be conducted.

Acknowledgments

The authors wish to thank Yan Shi of the Software Technology Advanced Research (STAR) Lab at the University of Texas at Dallas for her comments while this paper was still a work in progress.

References

1. H. Agrawal, R. A. DeMillo and E. H. Spafford, Debugging with dynamic slicing and backtracking, *Software — Practice & Experience* **23**(6) (1996) 589–616.

2. H. Agrawal, J. R. Horgan, S. London and W. E. Wong, Fault localization using execution slices and dataflow tests, in *Proceedings of the 6th IEEE International Symposium on Software Reliability Engineering*, Toulouse, France, 1995, pp. 143–151.
3. S. Amari, N. Murata, K. R. Muller, M. Finke and H. H. Yang, Asymptotic statistical theory of overtraining and cross-validation, *IEEE Transactions on Neural Networks* **8**(5) (1997) 985–996.
4. C. Anderson, A. Mayrhauser and R. Mraz, On the use of neural networks to guide software testing activities, *Proceedings of the IEEE International Test Conference on Driving Down the Cost of Test*, 1995, pp. 720–729.
5. B. Boehm and P. Papaccio, Understanding and controlling software costs, *IEEE Transactions on Software Engineering* **14**(10) (1988) 1462–1477.
6. S. R. Chu, R. Shoureshi and M. Tenorio, Neural networks for system identification, *IEEE Control Systems Magazine* **10**(3) (1990) 31–35.
7. L. A. Clarke, A. Podgurski, D. J. Richardson, and S. J. Zeil, A formal evaluation of data flow path selection criteria, *IEEE Transactions on Software Engineering* **15**(11) (1989) 1318–1332.
8. H. Cleve and A. Zeller, Locating causes of program failures, *Proceedings of the 27th International Conference on Software Engineering*, St. Louis, Missouri, 2005, pp. 342–351.
9. C. D. Doan and S. Y. Liong Generalization for multilayer neural network: Bayesian regularization or early stopping, *Proceedings of the 2nd Conference on Asia Pacific Association of Hydrology and Water Resources*, Singapore, 2004.
10. L. Fausett, *Fundamentals of Neural Networks: Architectures, Algorithms, and Applications* (Prentice-Hall, 1994).
11. K. Fukushima, A neural network for visual pattern recognition, *Computer* **21**(3) (1988) 65–75.
12. M. J. Harrold, G. Rothermel, K. Sayre, R. Wu and L. Yi, An empirical investigation of the relationship between spectra differences and regression faults, *Journal of Software Testing, Verification and Reliability* **10**(3) (2000) 171–194.
13. R. Hecht-Nielsen, Theory of the backpropagation neural network, *Proceedings of 1989 International Joint Conference on Neural Networks*, Washington, DC, 1989, pp. 593–605.
14. M. Hutchins, H. Foster, T. Goradia and T. Ostrand, Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria, *Proceedings of the 16th International Conference on Software Engineering*, Sorrento, Italy, 1994, pp. 191–200.
15. J. A. Jones, J. Bowring and M. J. Harrold, Debugging in parallel, *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, London, UK, 2007, pp. 16–26.
16. J. A. Jones and M. J. Harrold, Empirical evaluation of the Tarantula automatic fault-localization technique, *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, Long Beach, California, 2005, pp. 273–282.
17. B. Korel and J. Laski, Dynamic program slicing, *Information Processing Letters* **29**(3) (1988) 155–163.
18. B. Liblit, M. Naik, A. X. Zheng, A. Aiken and M. I. Jordan, Scalable statistical bug isolation, *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, Chicago, Illinois, 2005, pp. 15–26.
19. C. Liu, L. Fei, X. Yan, J. Han and S. P. Midkiff, Statistical debugging: a hypothesis testing-based approach, *IEEE Transactions on Software Engineering* **32**(10) (2006) 831–848.

20. C. Liu and J. Han, Failure proximity: a fault localization-based approach, *Proceedings of the 14th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Portland, Oregon, 2006, pp. 286–295.
21. J. R. Lyle and M. Weiser, Automatic program bug location by program slicing, *Proceedings of the 2nd International Conference on Computer and Applications*, Beijing, China, 1987, pp. 877–883.
22. D. J. C. MacKay, A practical Bayesian framework for backpropagation networks, *Neural Computation* **4**(3) (1992) 448–472.
23. D. J. C. MacKay, Bayesian interpolation, *Neural Computation* **4**(3) (1992) 415–447.
24. K. S. Narendra and S. Mukhopadhyay, Intelligent control using neural networks, *IEEE Control Systems Magazine* **12**(2) (1992) 11–18.
25. D. E. Neumann, An enhanced neural network technique for software risk analysis, *IEEE Transactions on Software Engineering* **28**(9) (2002) 904–912.
26. Neural Network Toolbox, User Guide, The MathWorks Inc., 2006.
27. A. Podgurski, D. Leon, P. Francis, W. Masri, M. Minch, J. Sun and B. Wang, Automated support for classifying software failure reports, *Proceedings of the 25th International Conference on Software Engineering*, Portland, Oregon, 2003, pp. 465–475.
28. M. Renieres and S. P. Reiss, Fault localization with nearest neighbor queries, *Proceedings of 18th IEEE International Conference on Automated Software Engineering*, Montreal, Canada, 2003, pp. 30–39.
29. The Siemens Suite, <http://www-static.cc.gatech.edu/aristotle/Tools/subjects/>, January 2007.
30. Y. S. Su and C. Y. Huang, Neural-network-based approaches for software reliability estimation using dynamic weighted combinational models, *Journal of Systems and Software* **80**(4) (2007) 606–615.
31. N. Tadayon, Neural network approach for software cost estimation, *Proceedings of International Conference on Information Technology: Coding and Computing*, 2005, pp. 815–818.
32. M. Weiser, Program slicing, *IEEE Transactions on Software Engineering* **10**(4) (1984) 352–357.
33. M. Weiser, Programmers use slices when debugging, *Communications of the ACM* **25**(7) (1982) 446–452.
34. W. E. Wong, J. R. Horgan, S. London and A. P. Mathur, Effect of test set minimization on fault detection effectiveness, *Software — Practice and Experience* **28**(4) (1998) 347–369.
35. W. E. Wong and Y. Qi, Effective program debugging based on execution slices and inter-block data dependency, *Journal of Systems and Software* **79**(7) (2006) 891–903.
36. W. E. Wong, T. Sugeta, Y. Qi and J. C. Maldonado, Smart debugging software architectural design in SDL, *Journal of Systems and Software* **76**(1) (2005) 15–28.
37. W. E. Wong, T. Wei, Y. Qi and L. Zhao, A Crosstab-based statistical method for effective fault localization, *Proceedings of The First International Conference on Software Testing, Verification and Validation*, Lillehammer, Norway, April 2008.
38. χ Suds User's Manual, Telcordia Technologies, 1998.
39. A. Zeller, Isolating cause-effect chains from computer programs, *Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering*, Charleston, South Carolina, 2002, pp. 1–10.
40. A. Zeller and R. Hildebrandt, Simplifying and isolating failure-inducing input, *IEEE Transactions on Software Engineering* **28**(2) (2002) 183–200.
41. A. X. Zheng, M. I. Jordan, B. Liblit, M. Naik and A. Aiken, Statistical debugging: simultaneous identification of multiple bugs, *Proceedings of the 23rd international Conference on Machine Learning*, Pittsburgh, Pennsylvania, 2006, pp. 1105–1112.