


Research Article

Multi-Error Location Method Based on Path Clustering and Failure Weighting

Xiaoyin Wang ^{1,2,3}, Chunyang Hu,⁴ Jiaze Sun,^{1,2,3} and Shuyan Wang¹

¹*Xi'an University of Posts & Telecommunications, Xi'an, Shaanxi 710121, China*

²*Shaanxi Provincial Key Laboratory of Network Data Analysis and Intelligent Processing, Xi'an, Shaanxi 710121, China*

³*Xi'an Key Laboratory of Big Data and Intelligent Computing, Xi'an, Shaanxi 710121, China*

⁴*Hubei University of Arts and Science, 441053 Xiangyang, Hubei, China*

Correspondence should be addressed to Xiaoyin Wang; wangxiaoyinxy@126.com

Received 17 May 2022; Revised 4 July 2022; Accepted 22 July 2022; Published 28 August 2022

Academic Editor: Tahir Mehmood

Copyright © 2022 Xiaoyin Wang et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

Since the traditional multiple-error locating method in software testing is difficult to achieve, and its information analysis is inaccurate, a software multiple-error locating method based on path clustering and failure weighting is proposed in this paper. In an environment with complete test cases, the program execution context information is dynamically captured by running test cases, and a path matrix of execution trajectory information is constructed. The cluster analysis is used to divide clusters and expand the weight of failed execution and added to suspicious in the process of degree evaluation to troubleshoot multiple errors. Experiments are implemented on four benchmark programs. The results show that compared with the five methods based on the equivalent evaluation function, the error location cost of the proposed method was reduced by 19.15% on average and effectively improved the efficiency of error location.

1. Introduction

Software testing is an important stage in the process of software development and software quality assurance [1]. It is a very time-consuming and energy-consuming work, which needs to consume almost 50% of software system development resources [2]. The problem of software error location is widely concerned by the industry and academia [3]. Error location is exploited to detect the existing error software statements i and improve the efficiency of software testing. The techniques of error location are frequently explored in the field of software research in recent years, and any optimization will decrease the cost of software development [4].

Lots of research studies in the field of software error location are conducted. Peng et al. [5] have put forward the ABFL method using spectrum-based (SBFL) technology to accurately locate code error. Zheng et al. [6] used the genetic algorithm to achieve a highly flexible software multi-fault location FSMFL framework. Feyzi and Parsa [7] have put forward a statistical method of fault tendency based on

elastic network regression namely FPA-FL. Zhu [8] used supervised and semisupervised learning techniques for software testing. Gao et al. [9] defined the DStar method to find the focus of solving the problem of wrong location by improving the experimental results with the increase of parameter values. Wong et al. [10] introduced a method to predict software error location based on test Hamming distance and the K-means algorithm. Huang et al. [11] proposed a software error location method FGAF based on the function call path and the genetic algorithm. Li et al. [12] have proposed a top-down software error location algorithm based on the weakest precondition. Wang and Sun [13] devised a software error location technique based on program variation analysis to reduce the impact of accidental successful test cases by analyzing program variation. Jiang et al. [14] proved the equivalence relationship of 30 suspicion formulas and the pros and cons of the positioning effect based on genetic algorithm analysis; Diguseppe and Jones [15] proposed an error location based on the combination of program dynamic slicing and the Bayesian method.

The existing software multi-error locating method [16] and the multi-objective optimization algorithm [17] has a relatively larger difficulty coefficient and inaccurate information analysis. When software testing is carried out in a real environment, the type, number, and distribution of errors cannot be known in advance. When there are multiple errors in the program, the single error locating strategy, which ignores the interaction between them, becomes less applicable. At this stage, there are relatively few research works focusing on the problem of multiple error localization in software. Each method has its special scope of application. How to improve the limitations of the method from different angles and reduce the high cost is worth thinking and exploring.

This paper proposes a method to troubleshoot errors from the perspective of multiple errors in software. First, cluster analyzing model processes the path matrix of the program execution context, increases the weight of the failure execution and is applied to the calculation process of suspicious sentence suspicion to search error. On one hand, it circumvents the problems of limited positioning results and high calculation difficulty coefficients of the existing multiple-error positioning methods. On the other hand, it enriches the theories and methods of data mining technology in the field of error positioning.

2. Path Cluster

2.1. Principles of Cluster Analysis. Clustering analysis is an unsupervised learning process in which a data object is divided into clusters with different attribute characteristics to realize the similarity coefficient of the same cluster elements and the similarity coefficient of different cluster elements. Single error location method is mostly a repeated test in one-bug-at-a-time way with low efficiency. And when the number and distribution of errors cannot be learned in advance, the effectiveness of the software single error location method to achieve multi-error location will be limited. Therefore, it is necessary to use cluster analysis to locate multiple errors.

In this paper, a clustering analysis algorithm is introduced by using the similar relationship between program execution trajectories to divide the huge execution trajectory information set into several smaller information sets and calculate the suspicion based on the coverage information according to the execution trajectories in the subsets. The efficiency of strategy execution is improved by reducing redundant data. At the same time, the size and type of the sample set are not limited, and targeted constraints can be added, which is beneficial to the selection of the execution trajectory.

2.2. Path Clustering Generation. When setting parameters, a path matrix \mathbf{P} with dimension $m \times (n + 1)$ from the program execution path and execution result information is constructed, m represents the number of test cases, n represents the number of lines of the sentence, and the element $x_{i(n+1)}$ ($1 \leq i \leq m$) in the $n + 1$ column of the matrix represents the actual output obtained by executing the test case

number i . If the execution result is consistent with expectations, $x_{i(n+1)} = 1$, and if the execution result deviates from expectations, $x_{i(n+1)} = 0$. The matrix element x_{ij} ($1 \leq i \leq m, 1 \leq j \leq n$) represents the statement coverage information of the line number j when the i th test case is executed. If the j th sentence is covered, $x_{ij} = 1$ is obtained by applying the binary vector method, otherwise $x_{ij} = 0$. \mathbf{P} is formally represented as

$$\mathbf{P} = \begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1n} & x_{1(n+1)} \\ x_{21} & x_{22} & \cdots & x_{2n} & x_{2(n+1)} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ x_{m1} & x_{m2} & \cdots & x_{mn} & x_{m(n+1)} \end{bmatrix}, \quad (1)$$

where \mathbf{P} abstractly expresses the execution trajectory and covers information of all test cases in the form of a path matrix, and then a series of operations after clustering can be established on the basis of matrix operations in the field of numerical analysis, which expands the idea of solving problems within a limited range and vividly conveys the essence of the data object. The process of path clustering is illustrated in Figure 1.

In the process of path clustering, the value of k and the definition of the initial centroid determine the performance of the clustering strategy. The connection of data objects in the cluster and the proximity of the cluster determine the classification effectiveness of clusters. The multi-dimensional Euclidean distance method is selected to measure the distance between the execution trajectories. When the distance is shorter, the similarity coefficient between the trajectories is higher, so that the probability that they belong to the same cluster will be greater, respectively.

3. Software Error Positioning Model

3.1. Path Cluster Analysis. Assuming P_f is a Java program containing multiple errors and n is the number of executable code lines, $T = \{t_1, t_2, \dots, t_n\}$ is a set of test cases, t_i ($1 \leq i \leq n$) $\in T$, and $T = T_f \cup T_p$, where T_f is a test case that fails to execute. T_p is a test case that executes successfully.

To reduce the computational overhead of NP-hard intra-cluster variation optimization, the complexity will be $O(nkt)$ (where n is the number of data objects, k is the number of clusters to be clustered, t is the number of cumulative iterations, $k \ll n$ and $t \ll n$). The K-means clustering strategy uses multi-dimensional Euclidean distance measurement method for cluster analysis. Suppose execution profile vectors are $X = \{s_1, \dots, s_i, \dots, s_n, s_{(n+1)}\}$. and $Y = \{s'_1, \dots, s'_i, \dots, s'_n, s'_{(n+1)}\}$, then the N -dimensional Euclidean distance formula is

$$\begin{aligned} \text{Distance} &= \sqrt{\sum_{i=1}^{n+1} d_i^2} \\ &= \sqrt{(s_1 - s'_1)^2 + \dots + (s_i - s'_i)^2 + \dots + (s_{(n+1)} - s'_{(n+1)})^2}. \end{aligned} \quad (2)$$

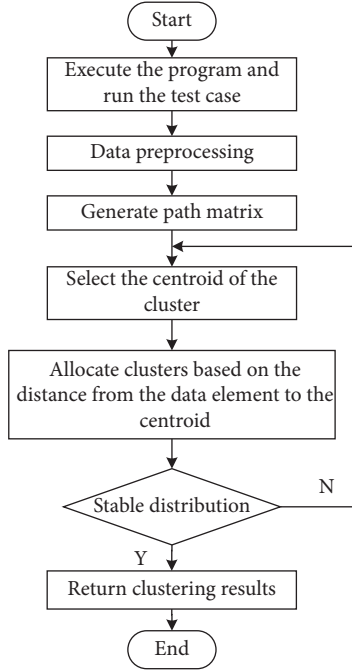


FIGURE 1: Path clustering process.

Among them, s_i is the i th coordinate of the first point and s'_i is the i th coordinate of the second point. If $s_i = s'_i$, then $d_i = 0$, otherwise $d_i = 1$.

When clustering, the determination of the k value changes according to the scale of the data object and is empirically selected as 0.5%–2% of the scale.

3.2. Failure Weighting. When troubleshooting errors based on spectral characteristics, the number of test cases will continue to vary. Successful test cases are negatively

correlated with suspicion, while failed test cases are positively correlated with corresponding suspicion. Therefore, this article adopts an approach inspired by Wong et al.'s claim that "when more and more test cases are run, the weight of successful test cases should be gradually reduced in stages," and proposes a technique to increase failure's weight. The suspicion calculation method increases the proportion of failed executions for the sentence of a certain line number, instead of directly counting the frequency of failed test cases, to improve the shortcomings of the Wong method, which is difficult to select the weight interval and the weight reduction factor cannot be adaptive.

When the process of the path matrix clustering is completed, the ratio of failure to successful test cases in each cluster is far less than 1. This results in the probability of statement execution failure being much less than the execution success. The influence of different sentence coverage and the same proportion on the calculation of suspicion are weakened and different test case sets are adapted only by changing the failed execution weight.

When the total number of test cases in a certain cluster is m , the number of coverage for the statement execution failure of line number i is

$$\text{fail}(i) = \sum_{x=1}^m \text{Cov}_{x,i} \times (1 - t(x)). \quad (3)$$

The number of successful coverage is

$$\text{Pass}(i) = \sum_{x=1}^m \text{Cov}_{x,i} \times t(x), \quad (4)$$

where x is the sequence identifier of the test case, the calculation formula of the statement coverage $\text{Cov}_{x,i}$ and the execution result $t(x)$ is

$$\text{Cov}_{x,i} = \begin{cases} 1, & \text{If the } i \text{ th statement is covered by the } x \text{ th test case,} \\ 0, & \text{If the } i \text{ th statement is not covered by the } x \text{ th test case,} \end{cases} \quad (5)$$

$$t(x) = \begin{cases} 1, & \text{If the } x \text{ th test case is executed successfully,} \\ 0, & \text{If the execution of the } x \text{ th test case fails.} \end{cases}$$

At this time, the failure weight factor of the statement with line number i is defined as

$$w(i) = \begin{cases} \frac{\sum_{x=1}^m t(x)}{\sum_{x=1}^m (1 - t(x))}, & \sum_{x=1}^m (1 - t(x)) \neq 0, \\ 0, & \sum_{x=1}^m (1 - t(x)) = 0. \end{cases} \quad (6)$$

Among them, $w(i)$ is the failure weight factor of the statement with line number i .

The weight factor changes with the ratio of successful test cases to failed test cases. The reason for increasing the failure

rather than the success weight factor is that adding a failed test case has a greater impact on the suspiciousness of the statement.

Furthermore, the formula for the total weight of failed test cases covers the sentence with line number i is

$$\text{Weigh}(\text{fail}(i)) = \sum_{x=1}^{\text{fail}(i)} w(i). \quad (7)$$

For any sentence in the cluster, the frequency of incorrect coverage is positively correlated with the suspicion, and the frequency of successful coverage is negatively correlated with the suspicion. Therefore, the method of increasing the failure weight factor is selected for a sentence, so

the weight of the failed test case increases with the number of sentence coverage and improves the efficiency of error location.

3.3. *Suspicion Calculation.* Under the setting of the weighting factor, a formula for calculating the suspicion of failure weight is proposed as follows:

$$\begin{aligned}
 QW01 &= \text{Weigh}(\text{fail}(i)) - \text{pass}(i) \\
 &= \sum_{x=1}^{\text{fail}(i)} w(i) - \text{pass}(i) \\
 &= \sum_{x=1}^{\text{fail}(i)} w(i) - \sum_{i=1}^m \text{Cov}_{x,i} \times t(x). \\
 QW02 &= \frac{\text{Weigh}(\text{fail}(i)) + 0.001}{\text{Weigh}(\text{fail}(i)) + \text{pass}(i)} \\
 &= \frac{\sum_{x=1}^{\text{fail}(i)} w(i) + 0.001}{\sum_{x=1}^{\text{fail}(i)} w(i) + \text{pass}(i)} \\
 &= \frac{\sum_{x=1}^{\text{fail}(i)} w(i) + 0.001}{\sum_{x=1}^{\text{fail}(i)} w(i) \sum_{i=1}^m \text{Cov}_{x,i} \times t(x)}.
 \end{aligned} \tag{8}$$

Among them, in order to prevent $\text{Weigh}(\text{fail}(i)) = 0$, the sentence suspicion degree is 0, and then the error location priority ranking cannot be performed, and $QW02$ introduces an adjustment factor of 0.001. In particular, the premises of establishing $QW01$ and $QW02$ are

$$\begin{aligned}
 \sum_{j=1}^n \text{fail}(j) + \sum_{j=1}^n \text{pass}(j) &= \sum_{j=1}^n \sum_{i=1}^m \text{Cov}_{x,i} \times (1 - t(x)) \\
 &+ \sum_{j=1}^n \sum_{i=1}^m \text{Cov}_{x,i} \times t(x) > 0.
 \end{aligned} \tag{10}$$

When calculating the sentence suspicion degree, if the j th sentence is not covered by the test case, the sentence will be deleted from the suspicious sentence set.

4. FCW Method

The implementation steps of FCW method are as follows:

Input: A Java program P_f with a total number of lines of executable code n and multiple errors

Output: Check statement priority order

Step 1: Run P_f to obtain program execution trajectory and execution results and extract feature elements to construct a coverage information matrix \mathbf{P} with dimension $m \times (n + 1)$. The parameter m represents the

$$\text{number of test cases } \mathbf{P} = \begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1n} & x_{1(n+1)} \\ x_{21} & x_{22} & \cdots & x_{2n} & x_{2(n+1)} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ x_{m1} & x_{m2} & \cdots & x_{mn} & x_{m(n+1)} \end{bmatrix}.$$

Step 2: Use the k -means algorithm to divide the test case set into k ($k \leq n$) subsets $C_1, \dots, C_j, \dots, C_k$,

where $i \geq 1, j \leq k, C_i \cap C_j = \emptyset$, and use formula (2) to calculate the similarity between program entities.

Step 3: In each test case set, calculate the suspicious degree of program entities separately and increase the proportion of failed test cases at the same time. Add the weight factor in formula (7) to the calculation process to obtain an improved suspicious degree calculation with equations (9) and (10).

Step 4: Arrange the suspicious sentences in descending order of suspicion and then generate a check sentence priority sequence Priority Sequence. According to the principle that the highest Priority Sequence is, firstly to be checked, the execution trajectory of multiple errors in P_f is tracked. Finally, the method's performance and the pros and cons are compared.

5. Experiment and Evaluation

5.1. *Experimental Setup and Design.* The FCW method uses EvoSuite to generate test cases. Cobertura obtains the execution path and coverage information and uses path clustering and failure weighting to troubleshoot errors.

According to the basic idea of the FCW method, taking further verification of its performance as a starting point, four Java benchmark programs with inconsistent versions of wrong versions were selected to carry out the experiment. The benchmark program descriptions are shown in Table 1.

5.2. *Comparison and Analysis of Error Positioning Methods.* When verifying whether the basic idea of the method is correct, several representative equivalence classes are selected to avoid repetitive operations and steps.

It can be practically proved that the risk assessment function applied in the direction of software error location can be divided into six equivalent categories (ER1~ER6), and two different methods are shown in each category in Table 2.

T_f represents the test cases that failed to execute, T_p represents the test cases that executed successfully, and T_{ef} represents the number of covered program entities and test cases failed to execute. T_{nf} represents the number of program entities that are not covered and the number of test cases that failed to execute. T_{ep} represents the number of entities covered and successfully executed by test cases. T_{np} represents the number of program entities that are not covered and successfully executed by test cases.

By performing path clustering operation on the benchmark program, the obtained experimental data are shown in Table 3. The columns 2~5 in the table indicate the number of errors in the program, the time required to execute the test case, the number of clustering of the path matrix, and the time spent in clustering, respectively. The experimental results imply that for different programs, although the number of errors is different, the number of path clusters may be identical. A reasonable number of clusters should be set according to the program's scale.

After cluster analysis, each class cluster was taken as a basic unit, and the failure weight factor was set as a parameter in the process of suspicion calculation for the

TABLE 1: Description of experimental objects.

Experimental procedures	Description	Number of lines of executable code	Number of cases tested
Next day	Date issues	132	22
TCAS	Air collision system	163	27
Sorting	Sorting algorithm	215	29
Tetris	Tetris	2397	65

TABLE 2: Six equivalent risk assessment functions.

Equivalence class	Equivalent risk assessment function	
ER1	Naish1 = $\begin{cases} -1, & \text{if } T_{ep} < T_f \\ T_p - T_{ep}, & \text{if } T_{ep} = T_f \end{cases}$	Naish2 = $T_{ef} - (T_{ep}/T_{ep} + T_{np} + 1)$
ER2	Jaccard = $(T_{ef}/T_{ef} + T_{nf} + T_{ep})$	Dice = $(2T_{ef}/T_{ef} + T_{nf} + T_{ep})$
ER3	Tarantula = $((T_{ef}/T_{ef} + T_{nf})/(T_{ef}/T_{ef} + T_{nf})) + (T_{ep}/T_{ep} + T_{np})$	CBI = $(T_{ef}/T_{ef} + T_{ep}) - (T_{ef} + T_{nf}/T_{ef} + T_{nf} + T_{ep} + T_{np})$
ER4	Wong1 = T_{ef}	Binary = $\begin{cases} 0, & \text{if } T_{ef} < T_f \\ 1, & \text{if } T_{ef} = T_f \end{cases}$
ER5	Wong2 = $T_{ef} - T_{ep}$	Sokal = $(2(T_{ef} + T_{np})/2(T_{ef} + T_{np}) + T_{nf} + T_{ep})$
ER6	Scott = $(4T_{ef}T_{np} - 4T_{nf}T_{ep} - (T_{nf} - T_{ep})^2)/(2T_{ef} + T_{nf} + T_{ep})(2T_{np} + T_{nf} + T_{ep})$	Rogot1 = $(1/2)((T_{ef}/2T_{ef} + T_{nf} + T_{ep}) + (T_{np}/2T_{np} + T_{nf} + T_{ep}))$

TABLE 3: Path clustering results.

Experimental program name	Number of errors	Test case execution time/s	Number of clusters	Clustering time/ms
Next day	3	0.256	2	14
TCAS	2	0.722	2	21
Sorting	4	1.617	3	20
Tetris	2	2.923	2	19

statements in the cluster, so that the track generated by errors and multiple errors in the program were traced in descending check order. Various methods are applied to all benchmark programs, and the comparison result is illustrated in Table 4. Columns 1~2 of Table 4 are the names of the program, the average number of sentences needs to be checked and the cost of error location. Columns 3~7 are the cost results of commonly used methods for error location. Columns 8~9 are presented in this section. The cost result of the FCW method required the error positioning shows the results by setting two different weighting factors for the QWo1 method and the QWo2 method. The data displayed in bold indicates the positioning cost of this method is higher than the FCW method, and the performance is not as good as the FCW method. As the scale of the program gradually increases, the positioning effect of the FCW method is better than Wong1 and Naish2, but the cost is higher than Tarantula, Jaccard, and Wong2.

By comparing the error location cost between different methods, the result is shown in Figure 2. The performance of the FCW method is improved. Under the setting of arithmetic average, it can be assured by checking about 25% of the code. The wrong location is reduced by 7.68%, 15.85%, 34.23%, 13.33%, and 25.83% compared with the wrong

location cost of Tarantula, Jaccard, Wong1, Wong2, and Naish2 methods. In contrast, Tarantula, Jaccard, Wong1, Wong2, and Naish2 methods need to check 31%~59% of the code to locate the error and require more time and material resources, but the efficiency obtained is not as expected.

In order to evaluate the effectiveness of the error location method proposed in this paper, experiments use the error location accuracy (called Acc) and the relative improvement value (called RImp) as the evaluation criteria [18]. Acc is defined as the percentage of executable statements that should be checked before a true error statement is found. RImp is the total number of statements that need to be checked to find all errors using Context-FL divided by the total number of statements that need to be checked using the compare method. The lower the value of RImp, the better the positioning effect [19].

As shown in Figure 3, a comparison of the localization efficiency of the five commonly used methods and the FCW method in this paper. The horizontal coordinate represents the percentage of executable statements checked in all programs, and the vertical coordinate represents the percentage of errors found by the locating method, and each point in the graph represents the percentage of errors located when checking the percentage of executable statements.

TABLE 4: Comparison of error location costs of different methods

Program name	Error location cost	Tarantula method	Jaccard method	Wong1 method	Wong2 methods	Naish2 methods	FCW method	
							QWo1 method	QWo2 method
Next day	Find the average number of check statements for fault 1	15.5	12.5	6.5	45.5	2.5	17	17
	Find the average number of check statements for fault 2	4	3	17	3	15	3	3
	Find the average number of check statements for fault 3	4	9.5	23.5	7.5	22.5	2	2
	Positioning cost	0.178	0.189	0.356	0.424	0.303	0.167	0.167
TCAS	Find the average number of check statements for fault 1	36	64	65.5	25.5	64	2	2
	Find the average number of check statements for fault 2	4.5	50	65.5	4.5	59	4	4
	Positioning cost	0.248	0.699	0.804	0.184	0.755	0.037	0.037
Sorting	Find the average number of check statements for fault 1	42.5	42.5	39.5	56.5	42.5	4.5	4.5
	Find the average number of check statements for fault 2	22	11.5	8.5	22	11.5	41.5	41.5
	Find the average number of check statements for fault 3	42.5	42.5	39.5	56.5	42.5	30.5	30.5
	Find the average number of check statements for fault 4	8	3	8.5	3	3	4.5	4.5
	Positioning cost	0.535	0.463	0.447	0.642	0.463	0.367	0.367
Tetris	Find the average number of check statements for fault 1	5	1	5	3	1	7	7
	Find the average number of check statements for fault 2	2.5	5	12.5	3	10.5	2.5	2.5
	Positioning cost	0.313	0.25	0.729	0.25	0.479	0.396	0.396

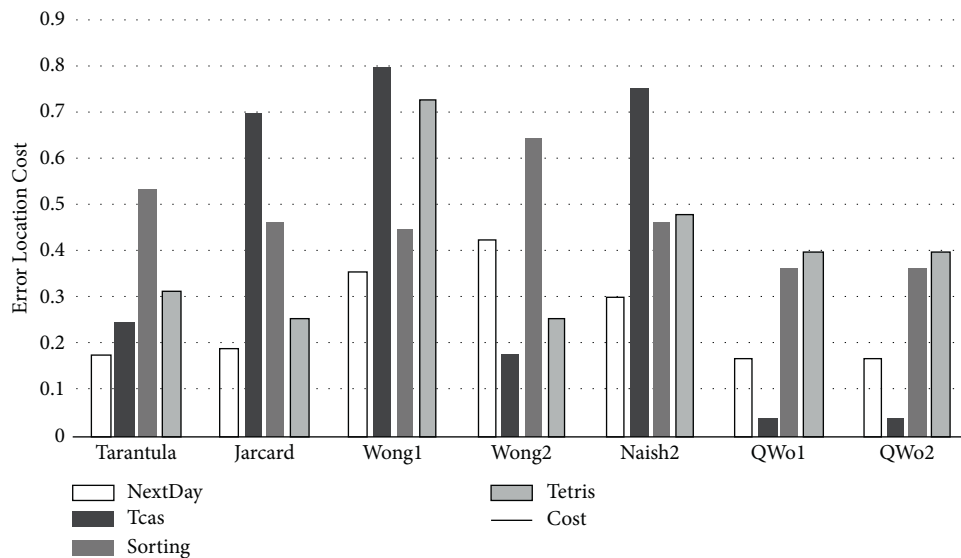


FIGURE 2: Comparison of error location costs.

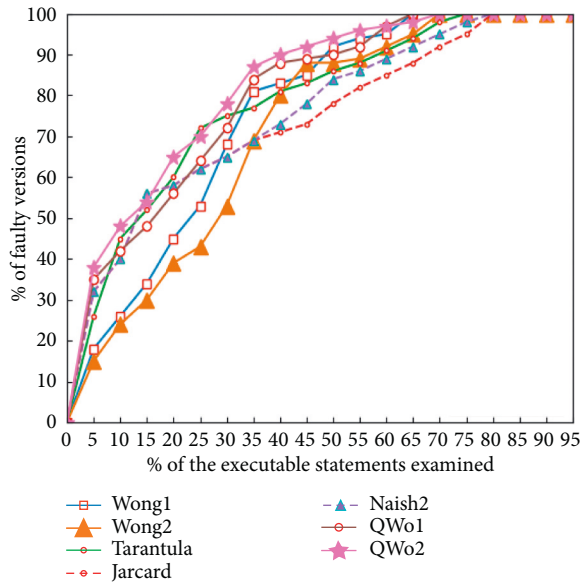


FIGURE 3: Comparison of localization efficiency.

6. Conclusion

Aiming to tackle multiple errors in software testing, this paper adopts an error location strategy combining path clustering and failure weighting. Through the performance test of the FCW method on four Java benchmark programs, and the comparison of the pros and cons of the strategy with five equivalent evaluation functions, the results show that the use of cluster analysis algorithm can cognize the difference between multiple errors, and weaken the interference between errors, and the failure weighted suspicion formula method can reduce the impact of the proportion of successful test cases after clustering. This method can improve the positioning accuracy within a certain range, reduce the complexity of the method implementation and affect the software testing cost.

Data Availability

The raw/processed data required to reproduce these findings cannot be shared as the data contains private data.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

The work is supported by the National Natural Science Foundation of China (Grant no. 61876138), the Key R&D Project of Shaanxi Province (2020GY-010), the Key Industrial Chain Core Technology Research Project of Xi'an (2022JH-RGZN-0028), and the Special Fund for Key Discipline Construction of General Institutions of Higher Learning from Shaanxi Province.

References

- [1] A. Bertolino, "Software testing research: achievements, challenges, dreams," in *Proceedings of the Future of Software Engineering*, pp. 85–103, IEEE, Minneapolis, MN, USA, May 2007.
- [2] C. Sharma, S. Sabharwal, and R. Sibal, "A survey on software testing techniques using genetic algorithm," *International Journal of Computer Science Issues*, vol. 10, no. 1, pp. 381–393, 2013.
- [3] P. Hao, Z. Zheng, Z. Y. Zhang, Y. C. Gao, C. Gong, and Y. Z. Xue, "Self-Adaptive fault localization algorithm based on predicate execution information analysis," *Chinese Journal of Computers*, vol. 37, no. 3, pp. 500–511, 2014.
- [4] Li Zheng, H. Wang, and Y. Liu, "HMER: a hybrid mutation execution reduction approach for mutation-based fault localization," *Journal of Systems and Software*, vol. 168, Article ID 110661, 2020.
- [5] Z. Peng, X. Xiao, G. Hu, A. Kumar Sangaiah, M. Atiqzaman, and S. Xia, "ABFL: an autoencoder based practical approach for software fault localization," *Information Sciences*, vol. 510, pp. 108–121, 2020.
- [6] Y. Zheng, Z. Wang, X. Fan, X. Chen, and Z. Yang, "Localizing multiple software faults based on evolution algorithm," *Journal of Systems and Software*, vol. 139, pp. 107–123, 2018.
- [7] F. Feyzi and S. Parsa, "FPA-FL: i," *Journal of Systems and Software*, vol. 136, pp. 39–58, 2018.
- [8] H. Zhu, "Software testing as a problem of machine learning: towards a foundation on computational learning theory," in *Proceedings of the 2018 IEEE/ACM 13th International Workshop on Automation of Software Test (AST)*, p. 1, Gothenburg, May 2018.
- [9] R. Gao, W. E. Wong, Z. Chen, and Y. Wang, "Effective software fault localization using predicted execution results," *Software Quality Journal*, vol. 25, no. 1, pp. 131–169, 2017.
- [10] W. E. Wong, V. Debroy, R. Gao, and Y. Li, "The DStar method for effective software fault localization," *IEEE Transactions on Reliability*, vol. 63, no. 1, pp. 290–308, 2014.
- [11] Q. Huang, Y. Mou, Z. Cui, and Z. Zhang, "Function level software error location," *Computer Engineering and Applications*, vol. 55, no. 20, pp. 1–10, 2020.
- [12] Y. Li, S. Huang, Y. Li, C. Ronghua, and D. Lang, "A software error location algorithm," *Journal of Electronics*, vol. 47, no. 01, pp. 25–32, 2019.
- [13] Q. Wang and W. Sun, "Software error location," *Computer Engineering*, vol. 43, no. 12, pp. 55–59, 2017.
- [14] Y. Jiang, W. Li, Y. Qiao, and S. Jiang, "Approach of fault localization based on Bayesian," *Computer Engineering and Design*, vol. 35, no. 11, pp. 3845–3849, 2014.
- [15] N. Diguseppe and J. A. Jones, "Fault density, fault types, and spectra-based fault localization," *Empirical Software Engineering*, vol. 20, no. 4, pp. 928–967, 2015.
- [16] T. T. Wang, C. Wang, X. H. Su, and Z. Lei, "Invariant based fault localization by analyzing error propagation," *Future Generation Computer Systems*, vol. 94, pp. 549–563, 2019.
- [17] J. Sun, J. Deng, Y. Li, and N. Han, "A BCS-GDE multi-objective optimization algorithm for combined cooling, heating and power model with decision strategies," *Applied Thermal Engineering*, vol. 213, Article ID 118685, 2022.

- [18] V. Debroy, W. E. Wong, X. Xu, and B. Choi, "A grouping-based strategy to improve the effectiveness of fault localization techniques," in *Proceedings of the 10th Int'l Conference on Quality Software. Zhangjiajie*, pp. 13–22, IEEE Computer Society, Zhangjiajie, China, July 2010.
- [19] Y. Lei, X. G. Mao, Z. Y. Dai, and C. S. Wang, "Effective statistical fault localization using program slices," in *Proceedings of the 36th Annual Int'l Computer Software and Applications Conference Izmir*, pp. 1–10, IEEE Computer Society, Izmir, Turkey, July 2013.