applied sciences

MDPI

Article

The MADAG Strategy for Fault Location Techniques

Shih-DA Wu [1] and Jung-Hua Lo [2,*]

[1] Ausenior Information Co., Ltd., Taipei City 11493, Taiwan
[2] Department of Applied Informatics, Fo-Guang University, Yilan County 26247, Taiwan
* Correspondence: jhlo@mail.fgu.edu.tw

Abstract: Spectrum-based fault localization (SBFL), which utilizes spectrum information of test cases to calculate the suspiciousness of each statement in a program, can reduce developers' effort. However, applying redundant test cases from a test suite to fault localization incurs a heavy burden, especially in a restricted resource environment, and it is expensive and infeasible to inspect the results of each test input. Prioritizing/selecting appropriate test cases is important to enable the practical application of the SBFL technique. In addition, we must ensure that applying the selected tests to SBFL can achieve approximately the effectiveness of fault localization with whole tests. This paper presents a test case prioritization/selection strategy, namely the Minimal Aggregate of the Diversity of All Groups (MADAG). The MADAG strategy prioritizes/selects test cases using information on the diversity of the execution trace of each test case. We implemented and applied the MADAG strategy to 233 faulty versions of the Siemens and UNIX programs from the Software-artifact Infrastructure Repository. The experiments show that (1) the MADAG strategy uses only 8.99 and 14.27 test cases, with an average of 18, from the Siemens and UNIX test suites, respectively, and the SBFL technique has approximate effectiveness for fault localization on all test cases and outperforms the previous best test case prioritization method; (2) we verify that applying whole tests from the test suite may not achieve the better effectiveness in fault localization compared with the tests selected by MADAG strategy.

Keywords: debugging; spectrum-based fault localization; test case prioritization; test suite reduction

Citation: Wu, S.-D.; Lo, J.-H. The MADAG Strategy for Fault Location Techniques. *Appl. Sci.* **2023**, *13*, 819. https://doi.org/10.3390/app13020819

Academic Editors: Zhenyu Chen, Chunrong Fang and Song Huang

Received: 18 November 2022
Revised: 27 December 2022
Accepted: 4 January 2023
Published: 6 January 2023

1. Introduction

1.1. Research Background and Motivation

The automatic software fault localization techniques are proposed to reduce the burden for a programmer to debug the program, where it identifies the statements (in a program) that need to be inspected by the programmer for debugging. One of the most well-known automatic software fault localization techniques is the spectrum-based fault localization (SBFL) technique. SBFL is a statistical technique that has relatively small overhead with respect to CPU time and memory requirement [1–4]. SBFL aims to locate the most suspiciously faulty statements of a program by the program spectrum information and testing results. A program spectrum records the execution information of program elements, such as how many statements, branches, or functions are executed in a specific test suite [5]. In this paper, we consider the elements of a program to be analyzed are statements. Our study could also be applied to different elements such as branches, functions, blocks, etc.

Although SBFL has superiority in locating faults, these techniques need to obtain information about the testing results (failed or passed) for each test case and require a test suite with high statement coverage composed of a large number of test cases. However, a high statement coverage test suite with expected testing outputs is difficult to build in practice. The inspection of each testing result as a failure or a success needs the human label and is time consuming. Even though currently used test case generation techniques [6–8] can produce a high statement coverage test suite, the testing results (failed/passed) are still unknown. Moreover, test case generation techniques may produce a large number of

redundant test cases that increase the cost of fault localization. Thus, the above-mentioned conditions would influence the feasibility of the existing SBFL techniques. We face the problem that considering a sufficient number of test cases with testing results for the SBFL technique is impractical; and even we are unable to determine whether the test cases are suitable for a program to locate the fault.

From a test suite, applying all the test cases to locate the fault may not be the most efficient approach. Therefore, we attempt to prioritize test cases or select only a subset of these test cases to retain "key" ones (unknown testing results) from the test suite in a program. Developers merely inspect the testing results of these selected test cases. Test case prioritization and test case reduction are popular issues in the field of regression testing. In previous studies [9–13], several researchers have performed prioritization or reduction on test cases, and selected the subset of test cases from the test suite to form a sub-test suite. The test cases of the sub-test suite are then applied to SBFL technique, thereby reducing the effort required for verifying the results of each test case and saving the cost of test cases to locate the fault. The effectiveness of the software fault localization is dependent on the test suite used; and the quality of the test cases determines the effectiveness of fault localization. Consequently, the supply of pivotal test cases to software fault localization not only facilitates the identification of program bug, but also substantially improves the efficiency of the fault localization. In other words, key test cases affect the effectiveness of the fault localization; thus, picking up key test cases is an important subject.

1.2. Research Purpose

The key research issues in this paper are as follows:

1. *How many test cases does MADGA need to achieve the same level of effectiveness of fault localization?*
2. *Is it still efficient to apply the SBFL technique with the whole test cases?*

In this paper, we propose a novel test case prioritization strategy, Minimal Aggregate of the Diversity of All Groups (MADAG), to select test cases from an existing test suite. Our strategy aims at minimizing the number of test cases to save the developer's effort in locating the fault. The selected test cases can be applied to SBFL techniques to locate the fault, while keeping the fault localization effectiveness as when whole test cases are run. In addition, the MADAG strategy does not obtain the test results in advance (besides the initial failed test case). We use the information about the execution trace of each test case, further computing the diversity of traces and selecting test cases. Moreover, in order to investigate the effectiveness of the MADAG approach, we performed an experimental study on 7 Siemens programs and 5 UNIX programs that are particularly widely used benchmarks for fault localization [1,2,14,15]. The experimental results showed that the MADAG strategy selected a substantially smaller subset of test cases from the test suite and keep approximate fault localization effectiveness.

The main contribution of this paper is twofold: (1) We propose a MADAG strategy to select the least number of test cases using SBLF (e.g., Jaccard, Tarantula, and Ochiai) to locate the fault, such that even if the number of test cases is reduced, the difference in the fault localization effectiveness will be not significant with checking more than 20% codes, compared with whole test cases. (2) We present that applying whole tests from the test suite is not likely to obtain the best effectiveness of fault localization in comparison with the selected test cases by the MADAG strategy.

The rest of this paper is organized as follows: Section 2 revisits test case prioritization techniques and fault localization techniques to be used in the experiment. In Section 3, the MADAG strategy is proposed in detail. Section 4 describes the empirical study, followed by its results. Section 5 reviews related work. Section 6 concludes our paper.

2. Preliminary

In this section, we summarize the technologies related to test case prioritization and software fault localization and discuss the application of the above-mentioned techniques to our experimental evaluation.

2.1. Test Case Prioritization

Suppose that a program P consists of n statements, s_1, s_2, s_3, . . . , s_n, and there are m test cases, t_1, t_2, t_3, . . . , t_m. A test case t_i is a set that consists of the statements that would be executed when ti serves as the input to run P. For each ti, there is a corresponding test result denoted by ri whose value can be 1 (implying test case i results in failure) or 0 (implying test case i results in success). Then, it can create a binary m × n matrix M to represent the program spectra and a vector R to record all the testing results, which are shown in Figure 1. SBFL can use the spectrum information and testing results to compute the suspiciousness value of each statement as being responsible for failures and sorting them in a ranking list. For SBFL techniques, several different similarity coefficients can be used [16], well known as Jaccard [17], Tarantula [18], and Ochiai [1] coefficient. Based on the spectra, a ranking metric will be used for calculating the suspicious program statements; faulty statements would have high suspiciousness scores, and non-faulty ones would have low suspiciousness scores. However, the above is not guaranteed.

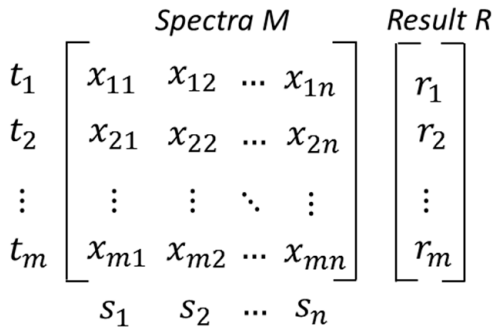


Figure 1. Program spectra *M* and testing result vector *R*.

Test case prioritization finds the ideal permutation of the test cases in a test suite to enhance a special testing criterion, such as the rate of fault detection, code coverage, or some other units of measurement [19,20]. This concept was first mentioned by Wong et al. [15]. They combined minimization and prioritization approaches to erase the omitted test cases and sort the selected ones from a test suite. Subsequently, Rothermel et al. [20] defined the test case prioritization (TCP) problem as follows:

Given: (1) *T*, a test suite, (2) *PT*, the set of all permutations of *T*, and (3) *f*, a function from *PT* to real numbers, $f{:}PT \rightarrow R$.

Problem: To find a permutation $p \in PT$ such that $\forall p \in PT.\ f(p) \geq f(p')$.

In this definition, the key for TCP techniques is to find a mapping function f that assigns an appropriate priority to the test cases. The following are the previous studies for TCP techniques that we compare with our approach.

Total-Statement [20] strategy assigns higher priorities to a test case that executes more statements in a program. Add-Statement [20] strategy extends the total-statement strategy by selecting the next test case that covers more statements that have not been covered by the previously selected test cases.

Adaptive Random Testing (ART) [21] strategy is a hybrid random/coverage-based approach that selects the next test case in two steps. First, ART selects the test cases randomly until one of them does not add additional coverage (e.g., until maximal coverage is achieved). Second, it selects test cases that maximized the Jaccard distance function with the already selected test cases. Experimental results show that of the above, ART-MIN is the best test case prioritization strategy. However, ART may be not effective when the failure

rate is low and the calculated Jaccard distance cost is higher than the cost of reducing the test execution times [22].

Fault-Exposing Potential (FEP): FEP is a coverage-based prioritization algorithm that guides the test case order. Rothermel et al. [20] used a mutation analysis to obtain an approximation of the fault-exposing potential of a test case to reduce the test oracle. Such an approximation was obtained based on the PIE analysis (propagation, infection, and execution) [23]. Suppose we are given a program P and the test suite T. First, they considered each test case t_i and each statement s_j in P, and calculated the fault-exposing potential $FEP(s, t)$ of t_i on s_j as the ratio of mutations of s_j detected by t_i to the total number of mutations of s_j. If t_i does not cover s_j, this ratio is zero. Second, the approach calculates for test case t_i an award value by summing the $FEP(s_j, t_i)$ values for all statements s_j in $P(\sum_n FEP_{ij})$. According to the calculated award value of the test, all test cases can be prioritized.

González-Sanchez et al. [11] developed two tools for test case prioritization. One is called "SEQUOIA", which uses the concept of diagnostic distribution, and estimates the probability of a program element to be faulty by making a Bayesian inference based on the already selected test cases. The other is called "RAPTOR" [24,25], which considers program elements that have the same execution traces as part of the same ambiguity group (AG). The principle of PAPTOR is to select the next test case by maximizing the ambiguity group value while trying to minimize the deviation of the sizes of the ambiguity groups.

Xia et al. [12] proposed the Diversity Maximization Speedup (DMS) strategy to prioritize test cases. DMS uses a linear regression analysis to identify the trend of each program element with a high change potential (W_T). Then, it assigns change-potential scores to the suspicious groups (program elements that are the same suspicious scores are grouped together). The rule of DMS for selecting the next test case is to divide a group into two sub-groups where the overall change potential values are maximized. Experiment results show that the current DMS strategy selects the least number of test cases.

2.2. Fault Localization Technique

Spectrum-based fault localization (also known as coverage-based fault localization) is a family of approaches to identify the faults in a program. Well-known similarity coefficients of the SBFL technique include Jaccard [17], Tarantula [18], and Ochiai [9]. These approaches collect the spectrum information by executing a subject program with its test suite. The program spectrum often includes information about whether a program element (e.g., a branch, a function, or a statement) is hit in an execution. A test case corresponds to a testing result (passed or failed). All the spectrum of the test cases forms a matrix mentioned in Figure 1. On the basis of the matrix, the above-mentioned approaches compute the suspiciousness score for each element and rank them according to their suspiciousness for the programmer to inspect. Jiang et al. [26] indicated that strategy and time cost of test case prioritization are factors to affect the effectiveness of SBFL techniques, not coverage granularity. So, we consider program elements to be statements in this study. Program elements could just as easily have been switched to other elements such as functions, branches, or predicates.

The key idea is for SBFL techniques to create a ranking of the most probable faulty elements such that these elements can then be inspected by programmers in the order of their suspiciousness until a fault is found. Table 1 lists the formulas of three similarity coefficients of the SBFL technique: Jaccard, Tarantula, and Ochiai. N_{ef} refers to the number of failed test cases that execute the statement. N_{uf} refers to the number of failed test cases that do not execute the statement. N_{es} refers to the number of successful test cases that execute the statement. N_{us} refers to the number of successful test cases that do not execute the statement. N_s refers to the total number of successful test cases. N_f refers to the total number of failed test cases.

Table 1. Spectrum-based fault localization.

Time	Name	Authors	Title 3
2002	Jaccard	Chen et al. [17]	$\frac{N_{ef}}{N_{ef}+N_{uf}+N_{es}}$
2005	Tarantula	Jones et al. [18]	$\frac{\left(N_{ef}/\ N_f\right)}{N_{es}/\ N_s+N_{ef}/\ N_f}$
2007	Ochiai	Abreu et al. [1]	$\frac{N_{ef}}{\sqrt{N_f\times\left(N_{ef}+N_{es}\right)}}$

SBFL is a statistical fault localization approach as compared to a probabilistic one with ultra-low computational complexity. Therefore, we attempt to use SBFL techniques after the test case prioritization (or test case reduction) techniques have selected the test cases.

3. The Proposed MADAG Strategy

In this section, we describe the proposed strategy in detail by presenting an example. Our strategy, Minimal Aggregate of the Diversity of All Groups' Elements (MADAG), prioritizes test cases and selects the appropriate ones, thereby applying them to the SBFL technique to locate the fault.

3.1. Overview

In order to maintain the superiority of SBFL, which can rapidly locate the fault, we need to keep "critical" test cases that can provide the information required for SBFL techniques to locate the fault and reduce the test input cost. Before applying our strategy, we must obtain the spectrum information by using a tool such as the "gcov" command of the GNC C compiler. Intuitively, the failed test cases can provide more information for SBFL to locate the fault. The execution traces of the failed test cases are more likely to be faulty. Such assumptions were adapted by Jones et al. [18] and Wong et al. [27] for the SBFL technique. Therefore, we set a failed test case as the starting point of our strategy. Before encountering a failed test, we may have to check some test cases. For a simple test case selection process, we adopt the same assumption as that adopted by Hao et al. [10]. That is, before checking the results of the test cases, the first test case encountered was a failed test. Moreover, we set the covered maximum number of statements of a failed test as the starting point of our strategy because there is a relatively small possibility of the covered maximum number of statements of a failed test missing locating faults.

When obtaining the spectrum information of a program and the initial failed test case, we set the execution trace of the initial failed test as the initial group. For the SBFL technique, an appropriate test suite can discriminatively divide the statements of a program. Therefore, it is possible that the suspected number of the same top-ranking program statements' can be minimized, e.g., to narrow down the inspection of the located fault. On the basis of this concept, our strategy selects test cases that have a stronger division capability for the number of statements in a program, such that each group composed of statements can be divided into subgroups, and the subtraction of the number of subgroup statements is the smallest. In Section 3.2, we illustrate the MADAG strategy in detail.

In this study, we implement the MADAG strategy based on statement coverage type. However, this approach can be easily shifted to other program elements, such as those related to function or branch coverage types, which we will evaluate and then use in an analysis of the effectiveness of different program elements in our future work.

3.2. MADAG Strategy

The statements covered by the initial failed test are divided into two coverage types: G_{cf} and G_{uf}. G_{cf} includes statements covered by the initial failed test, and Guf includes statements not covered by the initial failed test. Intuitively, the opportunity for faults in Guf is very low [18,27]. Therefore, we do not consider that the G_{uf} of the initial failed test in our study saves the cost of locating the fault. Furthermore, the statements of G_{cf} can set the initial group in our strategy. We define the division operations as $D_{o(1,1)}$, $D_{o(1,0)}$, $D_{o(0,1)}$,

and $D_{o(0,0)}$ for the statement coverage condition. $D_{o(1,1)}$ refers to both the current test case and the next selected test case that covers the statement. $D_{o(1,0)}$ refers to the current test case that covers the statement and the next selected test case that does not. $D_{o(0,1)}$ refers to the next selected test case that covers the statement and the current test case that does not. $D_{o(0,0)}$ refers to both the current test case and the next selected test case, which do not cover the statement. We describe the division operation using an example in Figure 2.

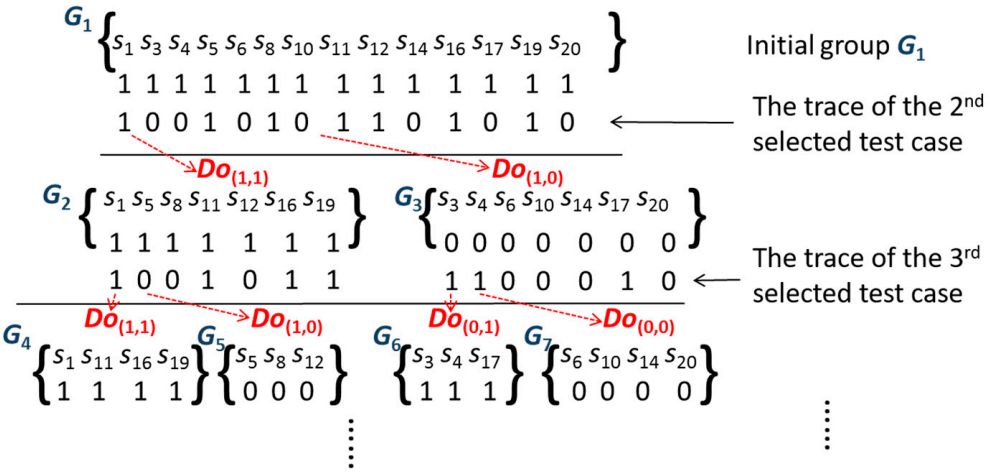


Figure 2. An example of the group division.

In this example, statements of the initial group G_1 (G_j, j denotes the group id) are covered by a failed test case, implying that the statements {s_1, s_3, s_4, s_5, s_6, s_8, s_{10}, s_{11}, s_{12}, s_{14}, s_{16}, s_{17}, s_{19}, s_{20}} may contain a fault. Further, we set the statements {s_1, s_3, s_4, s_5, s_6, s_8, s_{10}, s_{11}, s_{12}, s_{14}, s_{16}, s_{17}, s_{19}, s_{20}} as initial group G_1. The trace of the 2nd selected test case is different from the initial G_1. The 2nd selected test case would divide the initial group G_1 into two subgroups, G_2 {s_1, s_5, s_8, s_{11}, s_{12}, s_{16}, s_{19}} and G_3 {s_3, s_4, s_6, s_{10}, s_{14}, s_{17}, s_{20}}, which correspond to division operations $Do_{(1,1)}$ and $Do_{(1,0)}$, respectively. Further, the 3rd selected test case may divide G_2 and G_3 into G_4, G_5, G_6, and G_7, which correspond to $Do_{(1,1)}$, $Do_{(1,0)}$, $Do_{(0,1)}$, and $Do_{(1,0)}$, respectively. According to the previous coverage condition, we can learn of the next situation, in which $Do_{(1,-)}$ can be divided into $Do_{(1,1)}$, and $Do_{(1,0)}$, $Do_{(0,-)}$ can be divided into $Do_{(0,1)}$ and $Do_{(0,0)}$, and so on.

Starting from group G_1, the group can be divided into two subgroups: G_2 and G_3 when the 2nd selected test case is input. Then, when the 3rd selected test case is input, G_2 may be divided into two subgroups: G_4 and G_5; G_3 may be divided into two subgroups: G_6 and G_7, and so on. The procedure of the division group would generate a binary tree (called a MADAG tree), as shown in Figure 3. To facilitate the presentation of our strategy, we assigned each group a unique group id by level-order in the MADAG tree. In order to not undermine the sequential numbering in this tree, a group id would be represented as G_{2j} and G_{2j+1} from the parent node G_j (j starts from 1). Even if one of the groups (leaf node) cannot be divided into any subgroups (the number of statements in this group is less than two or it cannot be divided anymore by any test case), we still keep the group number to show which group does not exist. Therefore, the kth level has 2^{k-1} groups at most in the MADAG tree. On the kth level, the numbering of group id is $G_{2^{k-1}} \leq G_j < G_{2^k-1}$. As an example of $k = 4$, the number of groups is 2^3, and the numbering ranges from G_8 to G_{15}.

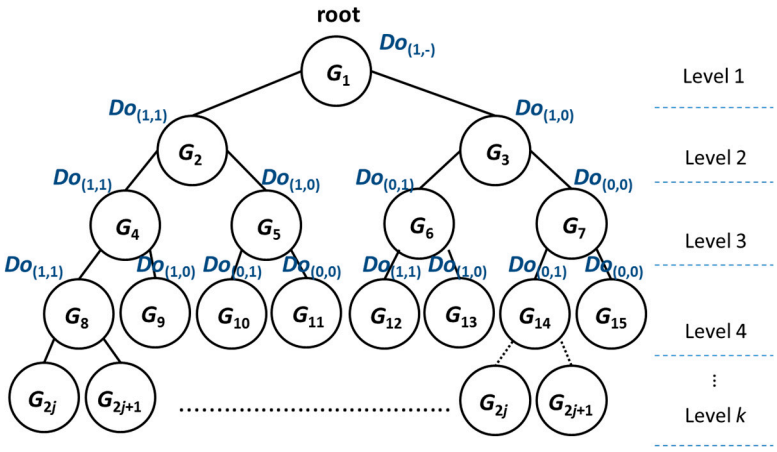


Figure 3. MADAG tree.

The principle in the proposed strategy is that the number of statements of each group can be evenly divided by the selected test case on each level in the MADAG tree. That is, when selecting the next test case, our strategy is the minimum value of the difference between the sum of the node numbers of the two subtrees of the MADAG tree. This test case prioritization strategy is called Minimal Aggregate of the Diversity of All Groups' Elements (MADAG), and the Algorithm 1 is presented as follows:

Algorithm 1. MADAG

function MADAG(T)
define
G_j denotes group id, and the group is a set that consists of statements
$\left|G_j\right|$ denotes the number of statements of this group
set
the selected test case set $ST = \{\ \}$
the statements covered by the initial failed test case ($t_{1^{st}}$) as initial group G_1
push $t_{1^{st}}$ into ST on level 1 (k = 1) such that $ST = \{t_{1^{st}}\}$
$T = T - ST \qquad \forall t_i \in T.$
k = 2;
loop {
select test cast t_i in ST by calculating the M_i of each test case at the k level
divide G_j into G_{2j} and G_{2j+1} by division operation//$\mathrm{Do}_{(1,1)}$, $\mathrm{Do}_{(1,0)}$, $\mathrm{Do}_{(0,1)}$, and $\mathrm{Do}_{(1,0)}$
$M_i(t_i) = \left\{\sum_{j=2^{(k-1)}-1}^{2^{(k-1)}-1} \left|\left|G_{2j}\right| - \left|G_{2j+1}\right|\right|\right\}, \forall t_i \in T.$
find the target test case t_{target} that has the minimal value of $M_i(t_i)$ on each level
$ST = ST \cup t_{target}$ *s.t.* $T = T - ST$.
$k = k + 1$;
if all of the selected test cases have the same $M_i(t_i)$ value
break;}
return ST

The algorithm receives a test suite T composed of test case t_i ($\forall t_i \in T$) and encounters the initial failed test case. In this algorithm, k denotes the level in the MADAG tree, G_j denotes the group id, and the group is a set that consists of statements. Moreover, $\left|G_j\right|$ denotes the number of statements of this group. First, we fetch the statements covered by the initial failed test case as the initial group G_1 and set the set ST, which exists in the selected test cases. We push the initial failed test case $t_{1^{st}}$ into ST at level 1 (k = 1) such that $ST = \{t_{1^{st}}\}$; then, T excludes set SK such as $T = T - ST$. Hereafter, we calculate the $M_i(t_i)$ value of each test case at each level; here, the $M_i(t_i)$ value measures the capability of each

test case, which can divide the statements of the parent group by division operations (such as $Do_{(1,1)}$, $Do_{(1,0)}$, $Do_{(0,1)}$, and $Do_{(0,0)}$). The formula for calculating $M_i(t_i)$ is as follows:

$$M_i(t_i) = \left\{ \sum_{j=2^{(k-1)}-1}^{2^{(k-1)}-1} \left| |G_{2j}| - |G_{2j+1}| \right| \right\}, \forall t_i \in T. \quad (1)$$

Here, groups $|G_{2j}|$ and $|G_{2j+1}|$ are divided from G_j by a division operation.

Subsequently, we find the target test case t_{target}, which has the minimal value of $M_i(t_i)$ at each level. The $M_i(t_i)$ value is the smaller mean used to divide the statements of a group more evenly. To attain t_{target}, it is pushed into the ST such that $ST = \{t_{1^{st}}, t_{target}\}$ and T exclude the set ST at this level. Then, we continue to the calculation of the next level. The procedure continues until all of the selected test cases have the same $M_i(t_i)$ value, thereby implying a convergence condition. Finally, the procedure returns the set ST, which consists of the selected test cases. Further, we apply the SBFL technique to the set ST to calculate the suspiciousness of each statement in a program.

3.3. An Example

To understand how the MADAG strategy works, we use an example program *Mid*() with the spectrum of test cases and suspiciousness metrics as shown in Figure 4. This program (adapted from Jones et al. [18] and Santelices [28]) takes three input values, namely x, y, and z, and then provides the median value as its output. Suppose that the program has 6 test cases t_1, t_2 . . . , t_6 and 13 statements s_1, s_2 . . . , s_{13}. For the program *Mid*(), a faulty statement is at s_7 ($m = y$;). A black dot for a statement in a test case implies that the corresponding statement is executed in the corresponding test case. The program spectra refer to obtaining such dots for the execution traces of each test case.

	Mid(){ int x,y,z,m	Test Collection t1 3,3,5	t2 1,2,3	t3 3,2,1	t4 5,5,5	t5 5,3,4	t6 2,1,3	Suspiciousness Metrics N_{cf}	N_{uf}	N_{cs}	N_{us}	Jaccard	Tarantula	Ochiai
1	scanf("%d %d %d" ,&x,&y,&z);	●	●	●	●	●	●	1	0	5	0	0.17	0.50	0.41
2	m=z;	●	●	●	●	●	●	1	0	5	0	0.17	0.50	0.41
3	if(y<z)	●	●	●	●	●	●	1	0	5	0	0.17	0.50	0.41
4	if(x<y)	●	●			●	●	1	0	3	2	0.25	0.63	0.50
5	m=y;		●					0	1	1	4	0.00	0.00	0.00
6	else if(x<z)	●				●	●	1	0	2	3	0.33	0.71	0.58
7	m=y;	●					●	1	0	1	4	**0.50**	**0.83**	**0.71**
8	else			●	●			0	1	2	3	0.00	0.00	0.00
9	if (x>y)			●	●			0	1	2	3	0.00	0.00	0.00
10	m=y;			●				0	1	1	4	0.00	0.00	0.00
11	else if (x>z)				●			0	1	1	4	0.00	0.00	0.00
12	m=x;							0	1	0	5	0.00	0.00	0.00
13	printf("middle number is =%d\n",m);	●	●	●	●	●	●	1	0	5	0	0.17	0.50	0.41
	Test Case Outcomes	P	P	P	P	P	F							

The maximum value of suspiciousness metrics

Figure 4. An example program *Mid*() with suspiciousness metrics.

The test result of each case is P (passed) or F (failed). Based on the spectrum and testing results, the SBFL technique can calculate the suspiciousness scores for each statement and further rank them at the top or bottom. In this example, the three well-known similarity coefficients of the SBFL technique, namely Jaccard [17], Tarantula [18], and Ochiai [9], rank the s_7 statement as the most suspicious statement. However, when our strategy is applied, the above-mentioned similarity coefficients of the SBFL technique can achieve the same effectiveness with fewer test cases (t_6, t_3, t_2, and t_5).

The process of the MADAG strategy for selecting test cases is shown in Table 2. The first column shows the k level in the MADAG tree. The second column refers to the set

ST that consists of test cases that have been selected. The third and fourth column shows the groups and each group that consists of the statements. The fifth column lists select test cases. The sixth column lists the $M_i(t_i)$ value of each test case. First, we select an initial test case t_6, which is a failed test with the covered maximum number of statements. Then, we define the set of execution trace of t_6 as the initial group G_1, which covers 7 statements $\{s_1, s_2, s_3, s_4, s_6, s_7, s_{13}\}$ and put the t_6 into the set *ST* on level 1. In our strategy, the principle of selecting the test case can divide the number of statements of the group evenly on each level. We calculate $M_i(t_i)$ value to find the t_{target} test case with the minimal value of $M_i(t_i)$ in this round, and then put the t_{target} into the set *SK*. Examples of $k = 2$, the $M_i(t_i)$ value for t_1, t_2, t_3, t_4, and t_5 is 7, 3, 1, 1, and 5, respectively. Therefore, we can choose t_3 or t_4 for the *ST* according to the $M_i(t_i)$ value on level 2. To ease the description, we show only the process of choosing the test input encountered earlier when more than one test has the same $M_i(t_i)$ value, as shown in Table 2. So, we select t_3 on level 2. The procedure is continued until *ST* contains test cases $\{t_6, t_3, t_2, t_5\}$. As all of the selected test cases have the same $M_i(t_i)$ value on level 5, the procedure would not select any test case immediately, thereby implying a convergence condition.

Table 2. Process of MADAG strategy.

***k* Level**	***ST***	**Groups**	**Groups Elements**	**To Select Test t_i**	**$M_i(t_i)$**
1	$\{t_6\}$	G_1	$\{s_1, s_2, s_3, s_4, s_6, s_7, s_{13}\}$		
2	$\{t_6, t_3\}$	G_2	$\{s_1, s_2, s_3, s_{13}\}$	t_1	7
				t_2	3
		G_3	$\{s_4, s_6, s_7\}$	t_3	1
				t_4	1
				t_5	5
3	$\{t_6, t_3, t_2\}$	G_4	$\{s_1, s_2, s_3, s_{13}\}$	t_1	7
		G_5	$\{\}$	t_2	5
		G_6	$\{s_4\}$	t_4	7
		G_7	$\{s_6, s_7\}$	t_5	5
4	$\{t_6, t_3, t_2, t_5\}$	G_8	$\{s_1, s_2, s_3, s_{13}\}$	t_1	7
		G_9	$\{\}$	t_4	7
		G_{14}	$\{s_6\}$	t_5	5
		G_{15}	$\{s_6\}$		
5	$\{t_6, t_3, t_2, t_5\}$	G_{16}	$\{s_1, s_2, s_3, s_{13}\}$	t_1	6
		G_{17}	$\{\}$	t_4	6

The procedure of the MADAG strategy generates the binary tree shown in Figure 5a. As the number of statements in the group was less than 2 (the left of colon denotes group id, and the right of the colon denotes the number of the group in Figure 5), the group cannot be divided into any subgroups (e.g., G_5 and G_6), but we still keep the numbered for the subgroups. After applying our strategy, we collect the program spectra and test results from the selected test case $\{t_6, t_3, t_2, t_5\}$. Jaccard, Tarantula, and Ochiai all assigned the s_7 statement a high suspiciousness score by using only four test cases and the suspiciousness scores of them are shown in Figure 5b. Even if we selected t_4 on level 2, the selection result of our strategy would be $\{t_6, t_4, t_2, t_5\}$ and the effectiveness of fault localization would be the same: $\{t_6, t_3, t_2, t_5\}$.

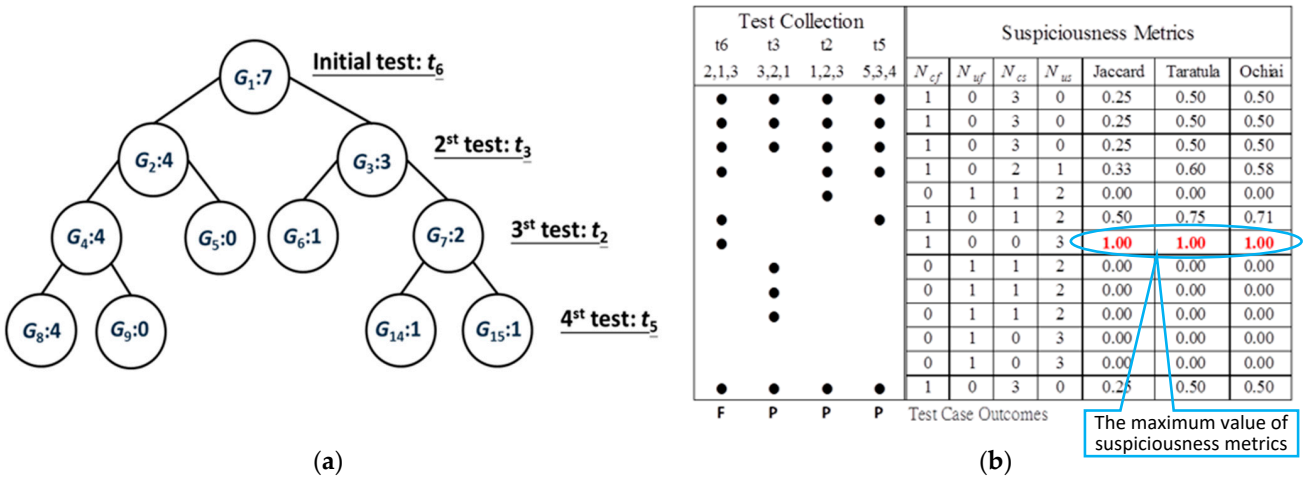


Figure 5. An example program *Mid*() with (**a**) MADAG tree and (**b**) suspiciousness metrics.

4. Empirical Study

This section presents an empirical evaluation that shows the effectiveness of the MADAG strategy compared with the other previous test case prioritization methods. We describe our experimental setup and measurements in Section 4.1, and we use the program subjects from SIR in Section 4.2. Then, we apply the MADAG strategy to the SBFL technique in Section 4.3. Finally, we explain the experimental results and analysis in Section 4.4.

4.1. Experimental Setups and Metrics

In our experiment, the MADAG strategy starts from a failed test input with the maximum number of statements. The reason why we set the maximum number of statements of a failed test as a starting point is to reduce the possibility of missing the fault. To reduce the influence of the selection regarding the initial failed test on the validity of the experiment, Hao et al. [10] and Xia et al. [12] used many random inputs (20 or 30 tests) as initial failed tests and computed the average experimental results when using these initial failed tests. Although randomly selecting failed test cases can alleviate the sensitivity of the technique to the choice of the starting statements, faulty locations not covered by the statements of the chosen failed tests may be missed. Moreover, each faulty version may not have more than 20 failed tests in its subject programs. Thus, we chose an initial failed test with the maximum number of statements as the starting point. Further, we experimentally investigated the effectiveness of the MADAG strategy compared with the other test case prioritization mentioned in Section 2.1.

Test case prioritization aim at prioritizing tests, reducing redundant test cases, and further narrowing the results to inspect the test outputs for developers. Simultaneously, the tests selected by the test case prioritization methods are applied to the SBFL technique to identify the fault compared with the application of all test cases to the SBFL technique, which achieves the approximate effectiveness of fault localization. A simple way to measure the effectiveness of test case prioritization methods is to observe how many test cases can be reduced. In other words, we will calculate the ratio of the selected test cases to the overall test cases, which is defined as follows:

$$\textit{Reduction ratio} = 1 - \frac{\text{selected test cases}}{\text{overall test cases}} \times 100 \quad (2)$$

where a larger *reduction ratio* indicates a better result.

In addition to considering the reduction ratio, we use expense [9,18,29] as the metric to measure fault localization effectiveness. We apply the selected test cases to the SBFL technique to observe the effectiveness between the prioritized (or selected) and non-prioritized test cases. Given a ranked list produced by the SBFL technique, expense measures the mini-

mum percentage of statements in a subject program that must be examined in descending order of the ranks to locate the fault. The formula is defined as follows:

$$Expense = \frac{\text{rank of the faulty statement}}{\text{number of executable statements}} \times 100\% \quad (3)$$

where a smaller expense indicates a better result.

We apply all test cases to the SBFL technique in order to calculate the expense, which we call the baseline diagnostic cost c. Test case prioritization and SBFL technique, then lead to a diagnostic cost c′ as same as in Xia et al. [12]. Evaluating the effectiveness of the SBFL technique involves discriminating the degree of cost c and c′, so the technique achieves x% of the baseline cost as follows:

$$x = \frac{c\prime}{c} \times 100\% \quad (4)$$

To compare each prioritization method mentioned in Section 2.1 fairly, the method can achieve 100% baseline effectiveness calculated by the number of selected tests as in Xia et al. [15]. Moreover, in practice, it is difficult to control the cost to be exactly 100% of the baseline in our strategy. Thus, we evaluate the number of selected test cases when the baseline effectiveness is 100% or greater, and we do not select any more test cases. A major goal of the MADAG strategy is to minimize the number of test cases for which the SFBL technique must be used to calculate the suspicious statements and the test results must be manually inspected, but still retain the approximate effectiveness of the SBFL technique. To verify our strategy, we evaluate the result of the above-mentioned metrics to prove that it has better effectiveness than the others.

We carried out an empirical study on a general laptop computer running Ubuntu with the gcc compiler [30] (gcc version 4.8.4). The laptop had an Intel Core i5-4210U (2.4 GHz, 4 cores) processor with 8 GB of physical memory.

4.2. Subject Programs

We use the Siemens test suite with 7 subject programs and the UNIX test suite with 5 real C programs (sed, flex, grep, gzip, and space) from the Software-artifact Infrastructure Repository (SIR) [12,31,32]. We instrument the programs and obtain their spectrum information with the "gcov" tool, but we exclude some faults that lead to a crash (e.g., core dump) and some faulty versions that do not cause any test cases to fail even when a fault exists. Furthermore, we exclude identical versions and faults that produce identical results to the correct versions. Overall, we study the 233 faults in total. Table 3 shows the details of subject programs, including their descriptions, code sizes (LOC), available test cases (Test case), and faulty versions.

Table 3. Subject programs.

Program	Description	LOC	Test Case	Faculty Versions
printtokens	Lexical analyzer	563–564	4130	5
printtokens2	Lexical analyzer	504–510	4115	9
schedule	Priority scheduler	307–308	2650	5
schedule2	priority scheduler	173–179	2710	9
tcas	Altitude separator	563–566	1608	40
totinfo	Info measurer	406	1052	23
replace	Pattern replacer	412–414	5542	29
space	ADL compiler	9126	13,585	29
grep	Text processor	12,653–13,372	470	13
gzip	Data compressor	6576–7996	214	15
flex	Lexical parser	12,424–14,245	525	47
sed	Text processor	10,055–11,990	360	9

4.3. Well-Known Similarity Coefficients of SBFL Technique

To measure the fault localization effectiveness of the selected tests, we use the MADAG strategy to select test cases, and then apply the selected test cases to the three similarity coefficients of the SBFL technique, including Jaccard (or Pinpoint) (Chen et al. [17]), Tarantula [18], and Ochiai [1,2]. In prior work, Abreu et al. [1,2] used the Jaccard similarity coefficient, which is used by the Pinpoint tool (Chen et al. [17]). The Jaccard coefficient can be defined as follows:

$$suspiciousnessJac(s) = \frac{N_{cf}}{N_{cf} + N_{uf} + N_{cs}} \tag{5}$$

where N_{cf}, N_{uf}, N_{cs}, N_{cf}, N_s, and N_f are defined as the same as in Section 2.2. Given the *suspiciousnessJac* values calculated for each statement, *s*, the suspiciousness is computed as shown in Figure 4.

Jones et al. [33] proposed the Tarantula technique, which was used initially for the visualization of testing information. The Tarantula technique [18] assigns two metrics to each program element—*suspiciousness* and *confidence*—according to the coverage information on the passed and failed test cases. The Tarantula coefficient can be defined as follows:

$$suspiciousnessTar(s) = \frac{\left(N_{cf} / N_f\right)}{N_{cs} / N_s + N_{cf} / N_f} \tag{6}$$

where N_{cf}, N_{uf}, N_{cs}, N_{cf}, N_s, and N_f are the same as in Equation (5).

The *confidence* metric is meant to evaluate the degree of confidence for a suspiciousness value. The Tarantula technique assigns greater confidence to statements that are covered by more test cases. The *confidence* of a statement is calculated by the formula:

$$confidence(s) = max\left(N_{cf} / N_f \, , \, N_{cs} / N_s \right) \tag{7}$$

The Tarantula technique ranks all statements at the top or bottom by suspiciousness values in the subject and uses the confidence values to resolve ties. Abreu et al. [1,2] also proposed the Ochiai similarity coefficient (originates from the molecular biology domain) to compare Tarantula one. The Ochiai coefficient can be defined as follows:

$$suspiciousnessOch(s) = \frac{N_{cf}}{\sqrt{N_f \times \left(N_{cf} + N_{cs}\right)}} \tag{8}$$

where N_{cf}, N_{uf}, N_{cs}, N_{cf}, N_s, and N_f are the same as in Equation (5). This technique also ranks the similarity of the statements to the Jaccard and Tarantula coefficients. Abreu et al. [1,2] indicated that the Ochiai coefficient can yield a better diagnosis than the other coefficients, including Pinpoint (Chen et al. [17]) and Tarantula (Jones and Harrold [18]).

4.4. Experimental Results and Analysis

In this section, we evaluate the MADAG strategy based on the reduction ratio and expense cost with three similarity coefficients of the SBFL technique. Our experimental results answer RQ1 and RQ2.

RQ1. How many test cases applying the SBLF technique compared to whole test cases can achieve the same level of effectiveness of fault localization?

Table 4 shows the number of selected test cases and reduction ratio for each program by our strategy to achieve 100% effectiveness or greater of the baseline mentioned in Section 4.1. For the Siemens test suite (including 7 programs), we apply 9.96, 9.99, and 7.01 test cases on average to the Ochiai, Tarantula, and Jaccard, respectively. Moreover, the reduction ratio of the three similarity coefficients of the SBFL technique is more than 99.6%.

The Jaccard yields the highest value (99.744%), Tarantula has the worst one (99.622%), and Ochiai has the middle value (99.639%). In the Siemens test suite, the program with fewer number of test cases (e.g., totinfo has 1052 test cases in total) yields the worst performance in terms of reduction ratio; the larger one (e.g., replace has 5542 test cases in total) yields the best result in terms of reduction ratio, but the diversity of the statements for the replace program is greater than that of printtokens program, so the reduction ratio of printtokens program is better. The principle of our strategy is to select test cases from the information on the diversity of the statements; larger test cases have better results in terms of reduction ratio relative to smaller ones, but it is not guaranteed that the largest one has the best result.

Table 4. Test cases and reduction ratio on subject programs.

Program	Total TS	Ochiai		Tarantula		Jaccard	
		Test Cases	Ratio	Test Cases	Ratio	Test Cases	Ratio
printtokens	4130	8.40	99.797%	6.60	99.840%	6.20	99.850%
printtokens2	4115	11.78	99.714%	13.11	99.681%	10.89	99.735%
schedule	2650	10.80	99.592%	9.60	99.638%	9.00	99.660%
schedule2	2710	9.22	99.660%	9.00	99.668%	3.89	99.856%
tcas	1608	4.78	99.703%	4.08	99.747%	3.38	99.790%
totinfo	1052	7.13	99.322%	9.35	99.111%	5.17	99.508%
replace	5542	17.62	99.682%	18.17	99.672%	10.55	99.810%
Siemens Avg.		**9.96**	**99.639%**	**9.99**	**99.622%**	**7.01**	**99.744%**
grep	470	9.69	97.938%	10.85	97.692%	10.38	97.791%
gzip	214	12.47	94.174%	12.07	94.361%	12.20	94.299%
flex	525	13.09	97.508%	10.57	97.986%	13.85	97.362%
sed	360	13.56	96.235%	18.44	94.877%	13.56	97.235%
space	13,585	22.93	99.831%	17.62	99.870%	22.76	99.832%
UNIX Avg.		**14.35**	**97.137%**	**13.91**	**96.957%**	**14.55**	**97.104%**

For the UNIX test suite (including 5 programs), we merely use 14.35, 13.91, and 14.55 test cases on the Ochiai, Tarantula, and Jaccard on average, respectively, and the reduction ratio of these three similarity coefficients of SBFL technique are more than 96.9%. In the UNIX test suite, the program with fewer number of test cases (e.g., gzip has 214 test cases in total) yields the worst performance in terms of reduction ratio; larger test one (e.g., space has 13,585 test cases in total) yields the best result in terms of reduction ratio. However, there is no guarantee that a larger one has a better reduction ratio, as with the Siemens suite.

Furthermore, we compare the MADAG strategy with previous experimental results. Xia et al. [12] indicated how many test cases to run with the SBFL technique to achieve 100% baseline effectiveness or greater. Table 5 shows how many test cases are needed on average according to the baseline for each test case prioritization method. For example, the DMS approach requires 18 test cases on average for each faulty version of the 7 Siemens programs, whereas MADAG needs only 10 (Ochiai, Tarantula, and Jaccard techniques require 10, 10, and 7 (the numerical results rounded off to the 2nd decimal place) test cases to calculate suspicious statements, respectively, to achieve 100% baseline effectiveness or greater. The Tarantula technique needs the maximum number of test cases (9.99), and the Jaccard technique needs the minimum (7.01)) (maximum) and 7 (minimum) test cases for Tarantula and Jaccard, respectively. From the 5 UNIX programs, the DMS approach needs 16 test cases, whereas the MADAG strategy merely needs 14.6 (maximum) and 13.9 (minimum) test cases for Jaccard and Tarantula, respectively. Even if we apply only the Ochiai technique to calculate how many test cases we need, compared with DMS, MADAG needs 10 and 14.4 tests of the Siemens and UNIX programs, respectively, which are both less than for DMS. Overall, based on the Ochiai technique, the MADAG strategy achieves a 44.4% reduction to inspect the test outputs of the Siemens program and a 10% reduction in the UNIX programs compared with the existing best approach (DMS).

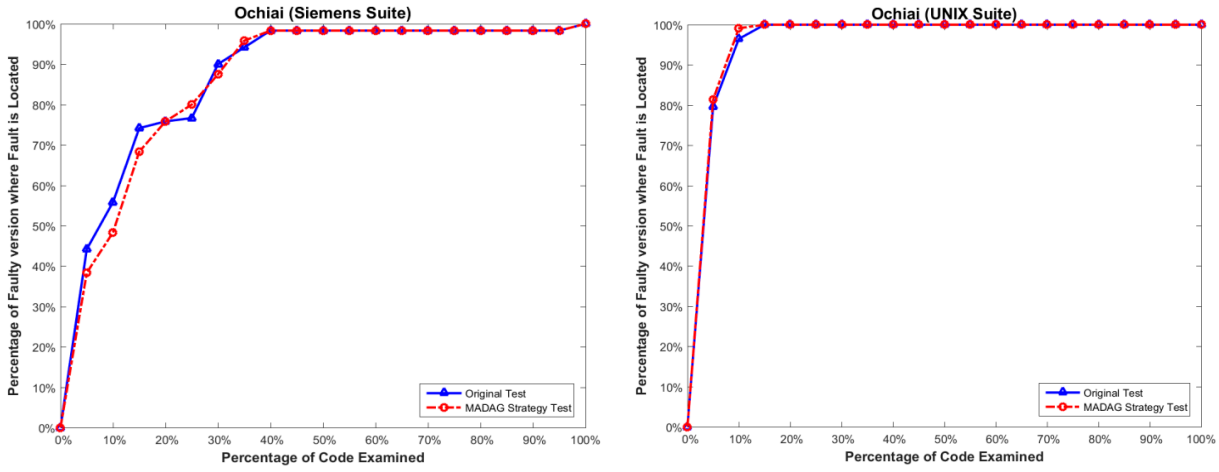
Table 5. Needed the number of test cases to SBFL techniques.

Subject Programs	MADAG	DMS	RAPTOR	FEP	ART-MIN	SEQUOIA	Total-Statement	Add-Statement
Siemens	Och:10.0 * Tar:10.0 * Jac:7.0 *	18	20	97	150	500_+	500_+	500_+
UNIX	Och:14.4 * Tar:13.9 * Jac:14.6 *	16	48	98	56	176	500_+	150

* The numerical results round off to the 2nd decimal place.

RQ2. Is it still efficient to apply the SBFL technique with the whole test cases?

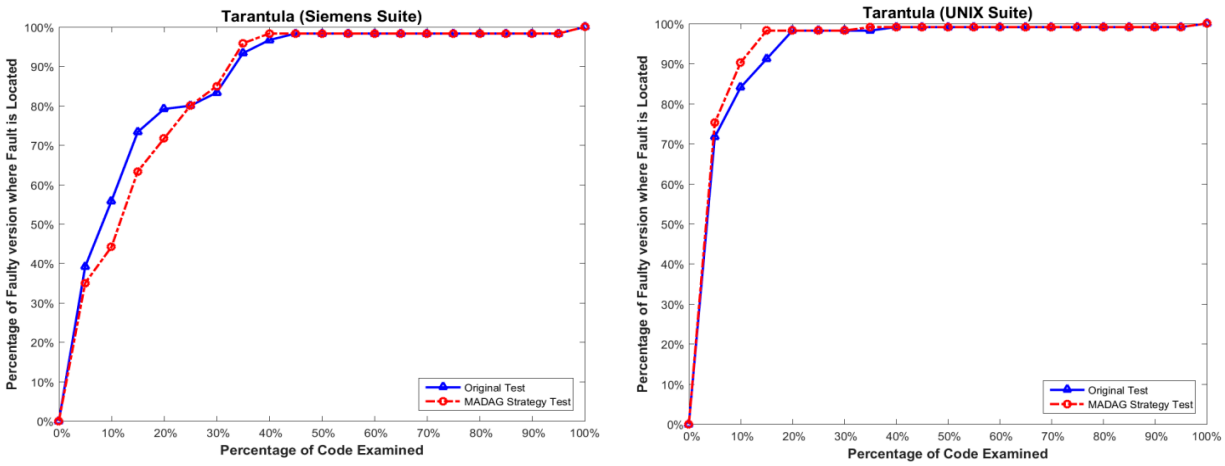
In addition to considering the reduction ratio, we must also determine whether the test cases selected by the MADAG strategy are suitable for the SBFL technique. To measure the effectiveness of the selected test cases by the MADAG strategy for the SBFL technique, we apply whole test cases (called original test) to Ochiai, Tarantula, and Jaccard to calculate the cost of fault localization for comparison with the test cases selected by the MADAG strategy (called MADAG test). The experimental results shown in Figures 6a, 7a and 8a are the results of the Ochiai, Tarantula, and Jaccard techniques respectively, for the Siemens suite, and Figures 6b, 7b and 8b are the same for the UNIX suite. In this figure, the x-axis represents the percentage of statements examined, and the y-axis represents the number of faulty versions where the fault is located by the examination of a number of statements less than or equal to the corresponding value on the x-axis.



(**a**) (**b**)

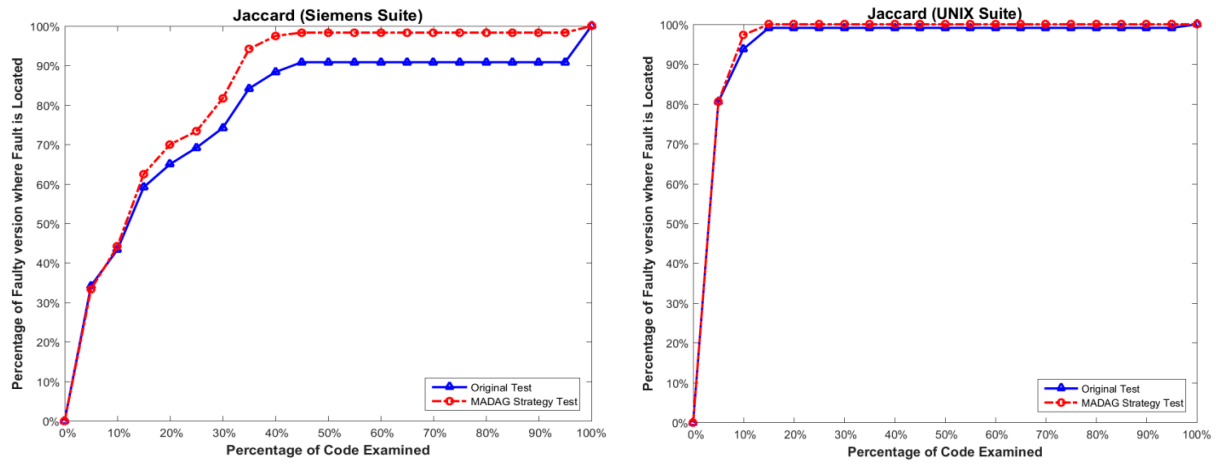
Figure 6. (**a**) Effectiveness comparison on the Siemens suite for Ochiai; and (**b**) on the UNIX suite for Ochiai.

In Figure 6a, we observe that the MADAG test and original test can find 38.3% and 44.1% faulty versions, respectively, in the Siemens suite by the Ochiai when the x-axis is 5%. Obviously, the gap between the MADAG test and the original test is 5.8%. However, examining more statements until the x-axis reaches 20%; both the MADAG test and the original test have quite similar effectiveness in fault localization. The curve of the MADAG test is equal to that of the original test when examining 30% of the statements, and they overlap when examining 40% of the statements. Here, we merely apply approximately 10 test cases on average of the Siemens suite with 7 programs, whereas the effectiveness of fault localization can achieve or equal that of the original test. This proves that the test cases selected by the MADAG strategy are fruitful for spectrum-based fault localization techniques.



(**a**) (**b**)

Figure 7. (**a**) Effectiveness comparison on the Siemens suite for Tarantula; and (**b**) on the UNIX suite for Tarantula.



(**a**) (**b**)

Figure 8. (**a**) Effectiveness comparison on the Siemens suite for Jaccard; and (**b**) on the UNIX suite for Ochiai.

In Figure 6b, the MADAG test and original test are similar in the shapes of the curves for effectiveness. However, the curve of the MADAG test is always higher than that of the original test until 15%, where they overlap. Moreover, we apply the selected test cases to the Tarantula and Jaccard to measure the effectiveness of fault localization. The Tarantula has similar effectiveness to the Ochiai for the Siemens and UNIX suites (Figure 7a,b). Among them, the effectiveness of the Ochiai is still superior to Tarantula. Jaccard has the worst efficiency compared with Tarantula and Ochiai [1,2].

In Figure 8a,b, we observe that the curve of the MADAG test is always higher than the original test because the test suite contains many redundant tests, which causes the effectiveness of the Jaccard to be poor. The selected test cases by the MADAG strategy improve the effectiveness of fault localization with the Siemens and UNIX suites for the Jaccard. This result implies that the MADAG strategy can pick up "key test cases" that have been applied to the Jaccard and have better effectiveness in terms of fault localization than the original test. Furthermore, by selecting test cases via the MADAG strategy and applying them to Ochiai, Tarantula, and Jaccard, their curves are always higher than those of the original test as shown in Figures 6b, 7b and 8b. Therefore, more redundant test cases

have an essentially inverse effect on the effectiveness of fault localization. In other words, the effectiveness of SBFL is likely not improved by applying whole test cases from the test suite. A similar experimental study by Zhang et al. [34] applied only FOnly tests to the SBFL technique, in which the effectiveness of fault localization is better than applying whole test cases.

5. Discussion

In this section, we discuss related work on test prioritization and selection, because the purpose of our proposed MADAG strategy is to select test cases as inputs to SBFL techniques to locate the fault. Furthermore, a suitable fault localization is an effective method to shorten the debugging process. Both topics have received considerable attention in the literature and in practice

5.1. Test Case Prioritization and Selection

Test case prioritization is a regression testing technique concerned with the ordering of test cases for early maximization of some properties, such as the rate of fault detection [32,35] whereas test selection aims at selecting some test cases from a given test suite for a special purpose. The test suite minimization aims at identifying redundant test cases and removing them to reduce the cost of test cases from the test suite. Test case prioritization and selection approaches often rely on structural coverage, such as statements and branches [20,36,37]. Rothermel et al. [20] summarized some criteria of test case prioritization methods based on coverage-based and fault-exposing based that can improve the rate of fault detection for the test suite. Elbaum et al. [36] showed a family of empirical studies to investigate the effectiveness of coarse granularity techniques and fine granularity techniques. Li et al. [37] suggested that the additional greedy algorithm should be used for regression test case prioritization. The other test suite reduction approach such as Dynamic Basic Block (DBB) [38] focuses on the number of DBBs and González-Sanchez et al. [24] further applied ambiguity groups for test suite reduction. Although González-Sanchez et al. both consider the testing effort (C_t) and inspection cost (C_d), the goal of this approach is to minimize overall cost ($C_t + C_d$), they do not separately consider the test case when a fault is actually detected during the execution of the prioritized test suite. However, this article considers the reduction ratio of the test case and subsequently applies selected test cases by the MADAG strategy to SBFL to discriminate the effectiveness of fault localization compared with all test cases.

5.2. Fault Localization

We applied the test cases selected by the MADAG strategy to three typical similarity coefficients of the SBFL technique (i.e., Jaccard [17]; Tarantula [18,33]; and Ochiai [1,2] in our experiment. Similar techniques, which calculate the suspiciousness of program elements based on program spectrum, have been proposed, such as Sober [39,40], Dstar [5], a novel model for fault localization [41], and others (e.g., Xie et al. [42]; Lucia et al. [43]). Renieris and Reiss [14] proposed a nearest-neighbor fault localization method that contrasts a failed test with another successful test and reports the most ambiguous locations in the program. Cleve and Zeller [44] applied Delta Debugging to identify the portion of one failed test that eventually causes failure. Some other researchers focused on generating tests to locate faults, such as Wang and Roychoudhury [45], who proposed a technique that generates a bug report by comparing passed execution and a failed execution. Campos et al. [46] applied probability theory concepts to guide test case generation using entropy. Our strategy, MADAG, selects test cases without knowing whether they have passed or failed, other than the initial failed test case. Furthermore, when picking up new test cases applied to the SBFL technique in order to calculate each suspicious statement, we merely inspect the result (passed or failed) of the selected tests. This strategy could reduce the tedious manual effort and the cost of test cases.

6. Conclusions

In this paper, we propose a new concept and strategy named Minimal Aggregate of the Diversity of All Groups aiming to select a small subset of test cases from a test suite while still permitting effective fault localization. We implement the MADAG strategy and apply it to 12 C programs with three well-known SBFL technology, such as Ochiai, Tarantula, and Jaccard. The research results show that the proposed MADAG strategy can make the test case reduction rate higher than 96%. Especially the Jaccard error location technology is the most obvious. The experimental result also shows that the MADAG strategy requires fewer test cases on average to achieve the effectiveness of subsequent fault localization techniques compared with seven other existing prioritization approaches. Furthermore, we also present that applying test cases selected by the MADAG strategy is likely to obtain better effectiveness in fault localization compared with applying whole test cases.

In future work, we will evaluate the MADAG strategy for more testing programs to investigate factors and characteristics of defects that most significantly impact SBFL performance. For the newly added dataset, we reduce the number of test cases and discuss whether there are significant differences between the experimental results and existing findings, while minimizing the impact on the effectiveness of software fault localization. Furthermore, we will combine different SBFL techniques and explore the possibility of locating the fault by the traversal of the MADAG tree (such as pre-order, in-order, and post-order) to improve overall performance.

Author Contributions: Conceptualization, S.-D.W. and J.-H.L.; methodology, S.-D.W. and J.-H.L.; software, S.-D.W.; validation, S.-D.W. and J.-H.L.; investigation, S.-D.W.; data curation, S.-D.W.; writing—original draft preparation, S.-D.W. and J.-H.L.; writing—review and editing, S.-D.W. and J.-H.L.; visualization, S.-D.W. and J.-H.L.; supervision, J.-H.L.; project administration, J.-H.L.; funding acquisition, J.-H.L. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by National Science and Technology Council (NSTC) of Taiwan, under grant no. [MOST 109-2221-E-431-003, and [MOST 111-2221-E-431-001].

Data Availability Statement: The data presented in this study are available on request from the corresponding author.

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

References

1. Abreu, R.; Zoeteweij, P.; van Gemund, A.J.C. Spectrum-Based Multiple Fault Localization. In Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE), Auckland, New Zealand, 16–20 November 2009; pp. 88–99. [CrossRef]
2. Abreu, R.; Zoeteweij, P.; Golsteijn, R.; Van Gemund, A.J. A practical evaluation of spectrum-based fault localization. *J. Syst. Softw.* **2009**, *82*, 1780–1792. [CrossRef]
3. Zakari, A.; Abdullahi, S.; Shagari, N.M.; Tambawal, A.B.; Shanono, N.M.; Maitama, J.Z.; Abdulrahman, S.M. Spectrum-based fault localization techniques application on multiple-fault programs: A review. *Glob. J. Comput. Sci. Technol.* **2020**, *20*, G2. [CrossRef]
4. Wen, M.; Chen, J.; Tian, Y.; Wu, R.; Hao, D.; Han, S.; Cheung, S.C. Historical spectrum based fault localization. *IEEE Trans. Softw. Eng.* **2019**, *47*, 2348–2368. [CrossRef]
5. Wong, W.E.; Debroy, V.; Ruizhi, G.; Yihao, L. The DStar method for effective software fault localization. *IEEE Trans. Reliab.* **2013**, *63*, 290–308. [CrossRef]
6. Godefroid, P.; Klarlund, N.; Sen, K. Dart: Directed automated random testing. In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, Chicago, IL, USA, 12–15 June 2005; pp. 213–223. [CrossRef]
7. Ma, E.; Fu, X.; Wang, X. Scalable path search for automated test case generation. *Electronics* **2022**, *11*, 727. [CrossRef]
8. Sen, K.; Marinov, D.; Agha, G. Cute: A concolic unit testing engine for c. In Proceedings of the ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), Lisbon, Portugal, 5–9 September 2005; pp. 263–272.
9. Yu, Y.; Jones, J.A.; Harrold, M.J. An empirical study of the effects of test-suite reduction on fault localization. In Proceedings of the 30th International Conference on Software Engineering (ICSE), Leipzig, Germany, 10–18 May 2008; pp. 201–210. [CrossRef]
10. Hao, D.; Xie, T.; Zhang, L.; Wang, X.; Sun, J.; Mei, H. Test input reduction for result inspection to facilitate fault localization. *J. Autom. Softw. Eng.* **2010**, *17*, 5–31. Available online: https://link.springer.com/article/10.1007/s10515-009-0056-x (accessed on 17 November 2022).

11. González-Sanchez, A.; Piel, É.; Abreu, R.; Groß, H.G.; van Gemund, A.J.C. Prioritizing tests for software fault diagnosis. *Softw. Pract. Exper.* **2011**, *41*, 1105–1129. [CrossRef]
12. Xia, X.; Gong, L.; Le, T.-B.; Lo, D.; Jiang, L.; Zhang, H. Diversity maximization speedup for localizing faults in single-fault and multi-fault programs. *J. Autom. Softw. Eng.* **2016**, *23*, 43–75. [CrossRef]
13. Wu, Y.; Liu, Y.; Wang, W.; Li, Z.; Chen, X.; Doyle, P. Theoretical Analysis and Empirical Study on the Impact of Coincidental Correct Test Cases in Multiple Fault Localization. *IEEE Trans. Reliab.* **2022**, *71*, 830–849. [CrossRef]
14. Renieris, M.; Reiss, S. Fault localization with nearest neighbor queries. In Proceedings of the IEEE/ACM 18th International Conference on Automated Software Engineering (ASE), Montreal, QC, Canada, 6–10 October 2003; pp. 141–154. [CrossRef]
15. Wong, W.E.; Horgan, J.R.; London, S.; Agrawal, H. A study of effective regression testing in practice. In Proceedings of the 8th International Symposium on Software Reliability Engineering (ISSRE), Albuquerque, NM, USA, 2–5 November 1997; pp. 264–274. [CrossRef]
16. Jain, A.K.; Dubes, R.C. *Algorithms for Clustering Data*; Prentice-Hall, Inc.: Upper Saddle River, NJ, USA, 1988.
17. Chen, M.Y.; Kiciman, E.; Fratkin, E.; Fox, A.; Brewer, E. Pinpoint: Problem determination in large, dynamic Internet services. In Proceedings of the International Conference on Dependable Systems and Networks (DSN), Washington, DC, USA, 23–26 June 2002; pp. 595–604. [CrossRef]
18. Jones, J.A.; Harrold, M.J. Empirical Evaluation of the Tarantula Automatic Fault-Localization Technique. In Proceedings of the IEEE/ACM 20th International Conference on Automated Software Engineering (ASE), Long Beach, CA, USA, 7–11 November 2005; pp. 273–282. [CrossRef]
19. Bajaj, A.; Sangwan, O.P. A systematic literature review of test case prioritization using genetic algorithms. *IEEE Access* **2019**, *7*, 126355–126375. [CrossRef]
20. Rothermel, G.; Untch, R.H.; Chu, C.; Harrold, M.J. Prioritizing test cases for regression testing. *IEEE Trans. Softw. Eng.* **2001**, *27*, 929–948. [CrossRef]
21. Jiang, B.; Zhang, Z.; Chan, W.K.; Tse, T.H. Adaptive Random Test Case Prioritization. In Proceedings of the IEEE/ACM 24th International Conference on Automated Software Engineering (ASE), Auckland, New Zealand, 16–20 November 2009; pp. 233–244. [CrossRef]
22. Arcuri, A.; Briand, L.C. Adaptive random testing: An illusion of effectiveness? In Proceedings of the International Symposium on Software Testing and Analysis (ISSTA), Toronto, ON, Canana, 17–21 July 2011; pp. 265–275.
23. Voas, J. PIE: A dynamic failure-based techniques. *IEEE Trans. Softw. Eng.* **1992**, *18*, 717–727. [CrossRef]
24. Gonzalez-Sanchez, A.; Abreu, R.; Gross, H.-G.; van Gemund, A.J.C. Prioritizing tests for fault localization through ambiguity group reduction. In Proceedings of the IEEE/ACM 26th International Conference on Automated Software Engineering (ASE), Lawrence, KS, USA, 6–10 November 2011; pp. 83–92. [CrossRef]
25. Zakari, A.; Lee, S.P.; Abreu, R.; Ahmed, B.H.; Rasheed, R.A. Multiple fault localization of software programs: A systematic literature review. *Inf. Softw. Technol.* **2020**, *124*, 106312. [CrossRef]
26. Jiang, B.; Zhang, Z.; Chan, W.K.; Tse, T.H.; Chen, T.Y. How well does test case prioritization integrate with statistical fault localization? *J. Inf. Softw. Technol.* **2012**, *54*, 739–758. [CrossRef]
27. Wong, W.E.; Debroy, V.; Choi, B. A family of code coverage-based heuristics for effective fault localization. *J. Syst. Softw.* **2010**, *83*, 188–208. [CrossRef]
28. Santelices, R.; Jones, J.A.; Yu, Y.; Harrold, M.J. Lightweight Fault-localization Using Multiple Coverage Types. In Proceedings of the 31th International Conference on Software Engineering (ICSE), Vancouver, BC, Canada, 16–24 May 2009; pp. 56–66. [CrossRef]
29. Jeffrey, D.; Gupta, N.; Gupta, R. Fault localization using value replacement. In Proceedings of the International Symposium on Software Testing and Analysis (ISSTA), Seattle, WA, USA, 20–24 July 2008; pp. 167–178. [CrossRef]
30. GCC, the GNU Compiler Collection. Available online: https://gcc.gnu.org/ (accessed on 3 November 2022).
31. Do, H.; Elbaum, S.G.; Rothermel, G. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empir. Softw. Eng.* **2005**, *10*, 405–435. [CrossRef]
32. Yoo, S.; Harman, M. Regression testing minimization, selection and prioritization: A survey. *Softw. Test. Verif.* **2010**, *22*, 67–120. [CrossRef]
33. Jones, J.A.; Harrold, M.J.; Stasko, J. Visualization of test information to assist fault localization. In Proceedings of the 24th International Conference on Software Engineering (ICSE), Orlando, FL, USA, 19–25 May 2002; pp. 467–477. [CrossRef]
34. Zhang, Z.; Chan, W.K.; Tse, T.H. Fault Localization Based Only on Failed Runs. *IEEE J. Comput.* **2012**, *45*, 64–71. [CrossRef]
35. Nayak, S.; Kumar, C.; Tripathi, S. Analytic hierarchy process-based regression test case prioritization technique enhancing the fault detection rate. *Soft Comput.* **2022**, *26*, 6953–6968. [CrossRef]
36. Elbaum, S.; Malishevsky, A.G.; Rothermel, G. Test case prioritization: A family of empirical studies. *IEEE Trans. Softw. Eng.* **2002**, *28*, 159–182. [CrossRef]
37. Li, Z.; Harman, M.; Hierons, R. Search algorithms for regression test case prioritization. *IEEE Trans. Softw. Eng.* **2007**, *33*, 225–237. [CrossRef]
38. Baudry, B.; Fleurey, F.; Traon, Y.L. Improving test suites for efficient fault localization. In Proceedings of the ICSE, Shanghai, China, 20–28 May 2006; pp. 82–91. [CrossRef]

39. Liu, C.; Yan, X.; Fei, L.; Han, J.; Midkiff, S.P. SOBER: Statistical model-based bug localization. In Proceedings of the ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Lisbon, Portugal, 5–9 September 2005; pp. 286–295.
40. Lei, Y.; Xie, H.; Zhang, T.; Yan, M.; Xu, Z.; Sun, C. Feature-FL: Feature-Based Fault Localization. *IEEE Trans. Reliab.* **2022**, *71*, 264–283. [CrossRef]
41. Naish, L.; Lee, H.J. Ramamohanarao, K. A model for spectra-based software diagnosis. *ACM Trans. Softw. Eng. Meth.* **2011**, *20*, 1–32. [CrossRef]
42. Xie, X.; Chen, T.Y.; Kuo, F.C.; Xu, B. A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization. *ACM Trans. Softw. Eng. Meth.* **2013**, *22*, 1–40. [CrossRef]
43. Lucia, L.; Lo, D.; Jiang, L.; Thung, F.; Budi, A. Extended comprehensive study of association measures for fault localization. *J. Softw. Evol. Proc.* **2014**, *26*, 172–219. [CrossRef]
44. Cleve, H.; Zeller, A. Locating causes of program failure. In Proceedings of the 27th International Conference on Software Engineering (ICSE), St. Louis, MO, USA, 15–21 May 2005; pp. 342–351. [CrossRef]
45. Wang, T.; Roychoudhury, A. Automated path generation for software fault localization. In Proceedings of the IEEE/ACM 20th International Conference on Automated Software Engineering (ASE), Long Beach, CA, USA, 7–11 November 2005; pp. 347–351. [CrossRef]
46. Campos, J.; Abreu, R.; Fraser, G.; d'Amorim, M. Entropy-based test generation for improved fault localization. In Proceedings of the IEEE/ACM 28th International Conference on Automated Software Engineering (ASE), Silicon Valley, CA, USA, 11–15 November 2013; pp. 257–267. [CrossRef]