The Journal of Systems and Software 139 (2018) 107–123

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Localizing multiple software faults based on evolution algorithm

Yan Zheng[a], Zan Wang[a,*], Xiangyu Fan[a], Xiang Chen[b], Zijiang Yang[c]

[a] *School of Computer Software, Tianjin University, China*
[b] *School of Computer Science and Technology, Nantong University, China*
[c] *Department of Computer Science, Western Michigan University, USA*

ARTICLE INFO

Article history:
Received 7 March 2017
Revised 18 January 2018
Accepted 5 February 2018
Available online 6 February 2018

Keywords:
Multi-fault localization
Program spectrum
Genetic algorithm
Search based software engineering

ABSTRACT

During software debugging, a significant amount of effort is required for programmers to identify the root cause of manifested failures. Various spectrum-based fault localization techniques have been proposed to automate the procedure. However, most of the existing fault localization approaches do not consider the fact that programs tend to have multiple faults. Considering faults in isolation results in less accurate analysis. In this paper, we propose a flexible framework called FSMFL for localizing multiple faults simultaneously based on genetic algorithms with simulated annealing. FSMFL can be easily extended by different fitness functions for the purpose of localizing multiple faults simultaneously. We have implemented a prototype and conducted extensive experiments to compare FSMFL against existing spectrum based fault localization approaches. The experimental results show that FSMFL is competitive in single-fault localization and superior in multi-fault localization.

1. Introduction

Testing and debugging are considered as the most expensive phase of the entire software development cycle (Beizer, 1990). One reason for such high cost is that the process of tracing propagation of faults and identifying the location of erroneous program statements is labor-intensive and time consuming. To reduce the cost, many fault localization techniques that automate, or partially automate, the locating faults procedure have been proposed in the past decade. Among them, Spectrum-based Fault Localization (SFL) methods (Wong et al., 2016; Jones et al., 2002; Abreu et al., 2006; Yoo, 2012; Dallmeier et al., 2005; Chen et al., 2002; Naish et al., 2011) have been shown effective and efficient. These methods assume that a test suite includes a sufficient number of passing and failing test cases. After comparing and contrasting these passing and failing executions, SFL then assign suspicious scores to all executed statements and output a descending list of these statements according to their suspicious value.

Numerous SFL techniques with a variety of suspicious score computation functions have been proposed. All of them have a common goal to assign a suspicious score as high as possible to a buggy statement, and as low as possible to a correct one. According to the study by Xie et al. (2016), fault localization techniques offer little help to debug small programs. This is understandable as examining the ranked statements may take as much effort as examining the whole source code in a small program. Therefore, fault localization techniques are likely used for debugging large programs, where multiple errors typically exist. From another perspective, according to the investigation performed by Xia et al. (2016), developers use and benefit from spectra-based fault localization techniques. Their experimental result reveals that both the accurate and mediocre spectra-based fault localization tools can help professional developers save their debugging time, and the improvements are statistically significant and substantial. Unfortunately, most SFL methods are optimized for single fault and thus the ranked statements are less accurate when there exist multiple faults. These methods assign the statements that are executed by more failed test cases with higher suspicious scores, even they are actually correct. For some faulty statements, the SFL methods may assign them low suspicious scores because executing them alone may not lead to failure. Consequently, some statements are ranked incorrectly when there are multiple faults.

To investigate this issue, the relationship between faults and the influence of multiple faults on fault localization have been examined (Debroy and Wong, 2009; DiGiuseppe and Jones, 2011; Xue and Namin, 2013). The studies imply that different faults may interfere with each other and significantly decrease localization accuracy. Based on the findings, several approaches for localizing multiple faults, or multi-fault localization (in contrast to traditional single-fault localization), have been proposed (Dean et al., 2009; Abreu et al., 2009; Steimann and Bertschler, 2009a;

* Corresponding author.
E-mail address: wangzan@tju.edu.cn (Z. Wang).

https://doi.org/10.1016/j.jss.2018.02.001

Steimann and Frenkel, 2012a; Gong et al., 2012). However, these multi-fault localization approaches have several limitations that prevent their adoption in real applications. Specifically, *linear-based* approach (Dean et al., 2009) adopts a linear programming model to solve multi-fault localization. Such approach requires that a suspicious function can be converted to a linear model. Thus, it is not general as not all suspicious functions can be converted to linear models. Instead of calculating the probability of each statement being the faulty one, the *coverage-based* approaches produce a set that consists of as many faulty statements as possible (Steimann and Bertschler, 2009a; Steimann and Frenkel, 2012a). However, the approaches do not attempt to locate all the faulty statements. That is, the last faulty statement can be ranked very low. In order to reduce the computational cost of the algorithm proposed in Steimann and Bertschler (2009a), integer linear programming is adopted to divide the coverage matrix (Steimann and Frenkel, 2012a). Nevertheless, these *coverage-based* approaches are not scalable due to relatively complex algorithms. Moreover, in Steimann and Bertschler (2009a) and Steimann and Frenkel (2012a), comparison experiments against other multi-fault localization approaches were not conducted.

In this paper, we propose FSMFL, a Fast Software Multi-Fault Localization Framework based on Genetic Algorithms. The innovation of this approach is that we transform the multi-fault localization problem to a search problem where program statements are encoded as a chromosome to indicate whether a statement is faulty. For example, "0100100000" represents ten lines of code where the second and fifth statements are faulty. Each chromosome is a candidate solution that is evaluated by a fitness function. A fitness function determines how good a candidate solution explains the failed and passed test cases that cover its statements. Then, a genetic algorithm generates a population that consists of a set of binary chromosomes. The genetic operators, such as selection, crossover, mutation, accepting and replacement, are employed to evolve the population until the algorithm reaches a predefined threshold. Finally, statements are ranked according to the last candidate population.

We have implemented a prototype of FSMFL and evaluated it on a large number of faulty programs. Our goal is to locate all the faulty statements, thus the ranking of the last faulty statement is important to us. Among the multi-fault localization approaches, *converge-based* approach and its variants (Steimann and Bertschler, 2009a; Steimann and Frenkel, 2012a) do not attempt to locate all the faulty statements and thus not in line with our purpose. Besides, the algorithm may not be scalable as no large empirical studies were conducted (Steimann and Bertschler, 2009a). Hence, besides comparing against single-fault localization approaches, we compare our approach against *linear-based* multi-fault localization approaches only. Our benchmark consists of Siemens programs, three Linux programs, space program and five Defects4j programs (Just et al., 2014; Andrews et al., 2005). Our experiments indicate that FSMFL outperforms the state-of-the-art single- and multi-fault localization approaches.

In summary, this paper makes the following contributions.

1. To the best of our knowledge, we are the first to propose a method that transforms multi-fault localization problem to a search problem so the genetic algorithm with simulated annealing can be exploited. A framework called FSMFL has been implemented. FSMFL is a flexible framework that accepts different population initialization strategies, fitness functions, and termination criteria.
2. We design a new fitness function for the purpose of multi-fault localization. The fitness function gives a candidate a higher fitness value if it covers more failed test cases and less passed test cases. Our experiments confirm that the fitness function is effective when handling both single and multi-fault programs.
3. We perform various optimizations to improve the performance of FSMFL. Meanwhile, we have built a benchmark that has both single-fault and multi-fault programs. FSMFL is evaluated against 8 other approaches on this benchmark. The evaluation shows that execution time of FSMFL is less than 23 s for large-scale programs (over 30,000 lines of code on average). Furthermore, statistical hypothesis tests (ANOVA, LSD, Benferrion, Wilcoxon Signed Rank and Cohen's d) are also conducted to verify the competitiveness of FSMFL.

The remainder of the paper is organized as follows. Section 2 describes the background and the motivation of our work. The detailed multi-fault localization approach is presented in Section 3, followed by empirical study in Section 4. After giving the related work in Section 5, Section 6 concludes the paper.

2. Motivation

In this section, we introduce the motivation with an example. Before that, we present the preliminary background for spectrum-based fault localization.

2.1. Preliminaries

Program spectrum is defined as the execution information of a program during the testing process, including coverage information, test results, and executable conditional branches. Spectrum-based fault localization requires a set of passed and failed test cases. The statistical information used in the traditional spectrum-based fault localization approaches are summarized in Table 1. Such approach requires The terms $n_{np}(s)$ and $n_{nf}(s)$ denote the number of passed and failed test cases that do not execute the program entity s. The terms n, n_p, and n_f represent the number of test cases, the number of passed test cases and the number of failed test cases, respectively. Typically a program entity is a program statement.

Spectrum-based fault localization ranks the program statements according to their suspicious scores. Intuitively, a statement has a higher suspicious score if it is frequently executed by failed test cases and seldom executed by passed ones. Note that, the suspicious score can be computed by different strategy which is referred to as the suspicious function. Table 2 gives the formula used by suspicious functions in six spectrum-based fault localization approaches that include Tarantula (Jones et al., 2002), Ochiai (Abreu et al., 2006), GP13(Yoo, 2012), Ample (Dallmeier et al., 2005), Jaccard (Chen et al., 2002), OP2 (Naish et al., 2011). A systematic comparison on the effectiveness of different suspicious functions has been conducted in a previous empirical study (Naish et al., 2011).

2.2. Motivating example

The first column in Fig. 1 gives a C program that finds the second largest variable among the four inputs. There are two faulty statements at Lines 14 and 17. In the next ten columns, we list ten test cases $T1$ to $T10$. We use 1 and 0 to denote whether a statement is covered by a test case. In the last row, we indicate whether the test case above is a passing(P) or failing(F) test case.

With the information we can easily obtain the values of $n_{ep}(s)$, $n_{np}(s)$, $n_{ef}(s)$, $n_{nf}(s)$, n_p and n_f, and then compute the suspicious scores using existing approaches. The last four columns give the ranking according to the results computed by the suspicious functions of Tarantula, Ochiai, OP2 and FSMFL (The suspicious value in FSMFL is measured by the newly proposed formula to be explained

Table 1
Statistical information in spectrum-based fault localization.

	Program entity s covered	Program entity s not covered	Program entity summary
Passed test cases	$n_{ep}(s)$	$n_{np}(s)$	n_p
Failed test cases	$n_{ef}(s)$	$n_{nf}(s)$	n_f
Test cases summary	$n_{ep}(s) + n_{ef}(s)$	$n_{np}(s) + n_{nf}(s)$	n

	Function *findSecond*	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	Tarantula	Ochiai	Op2	FSMFL
1	int *findSecond* (int *a* , int *b* , int *c* , int *d*) {														
2	int *max* , *min* , *temp* ;														
3	if(*a* <= *b*) {	1	1	1	1	1	1	1	1	1	1	6	5	5	10
4	*max* = *b* ;	1	1	1	1	0	0	0	0	0	0	1	1	1	2
5	*min* = *a* ;	1	1	1	1	0	0	0	0	0	0	2	2	2	3
6	} else {														
7	*max* = *a* ;	0	0	0	0	1	1	1	1	1	1	12	12	16	12
8	*min* = *b* ;	0	0	0	0	1	1	1	1	1	1	13	13	17	13
9	}														
10	if(*c* > *max*) {	1	1	1	1	1	1	1	1	1	1	7	6	6	11
11	*temp* = *max* ;	0	0	0	0	0	0	0	0	0	0	14	14	12	14
12	*max* = *c* ;	0	0	0	0	0	0	0	0	0	0	15	15	13	15
13	} else if(*c* < *min*) {	1	1	1	1	1	1	1	1	1	1	8	7	7	7
14	/* bug. Right is temp = min */ *temp* = *b* ;	**0**	**1**	**1**	**0**	**0**	**1**	**0**	**1**	**1**	**1**	**10**	**10**	**10**	**5**
15	*min* = c;	0	1	1	0	0	1	0	1	1	1	11	11	11	6
16	} else {														
17	/**bug* . Right is *temp* = *c* */ *temp* = *b* ;	**1**	**0**	**0**	**1**	**1**	**0**	**1**	**0**	**0**	**0**	**4**	**9**	**9**	**4**
18	}														
19	if(*d* < *max* && *d* > *temp*) {	1	1	1	1	1	1	1	1	1	1	9	8	8	9
20	return *d* ;	0	0	0	0	0	0	1	1	0	0	16	16	14	16
21	} else if(*d* > *max*) {	1	1	1	1	1	1	0	0	1	1	5	4	4	8
22	return *max* ;	1	1	0	0	1	1	0	0	0	0	17	17	15	17
23	} else {														
24	return *temp* ;	0	0	1	1	0	0	0	0	1	1	3	3	3	1
25	}														
26	}														
	Test Case Result	P	P	F	F	P	P	P	P	P	P				

Fig. 1. A program with two faults and its suspicious scores.

Table 2
Formulas used in spectrum-based fault localization.

Name	Formula	Name	Formula
Tarantula	$\frac{n_{ef}(s)/n_f}{n_{ef}(s)/n_f + n_{ep}(s)/n_p}$	Ample	$\left\lvert \frac{n_{ef}(s)}{n_f} + \frac{n_{ef}(s)}{n_p} \right\rvert$
Ochiai	$\frac{n_{ef}(s)}{\sqrt{n_f \times (n_{ef}(s) + n_{ep}(s))}}$	Jaccard	$\frac{n_{ef}(s)}{n_f + n_{ep}(s)}$
GP13	$n_{ef} \times \left(1 + \frac{1}{2n_{ep}(s) + n_{ef}(s)}\right)$	OP2	$n_{ef}(s) - n_{ep}(s)/(n_p + 1)$

in Section 3.4). It can be observed that Line 14 is ranked 10th by the three existing approaches, a very poor result considering there are only 17 statements in the program. As for Line 17, it is ranked 4th by Tarantula and 9th by Ochiai and OP2. The example may not be very representative, but it indeed shows that under multiple faults the existing approaches may give poor results. One of the reasons is that these approaches consider each statement in isolation and do not consider the effect of the combination of multiple statements. When there is only one faulty statement, it is more likely to be covered by more failing executions and less passing ones. The passing and failing executions are diluted by multiple faults so less accurate results are obtained. For example, Lines 14 and 17 are each covered once by either, but not both, of the two failing test cases *T*3 and *T*4. On the other hand, there are many statements are covered by both failing test cases.

In this paper, we propose an approach that does not consider statements in isolation. For example, if we consider Lines 14 and 17 together, as a group they are covered by both failing test cases and thus receives higher suspicious value. In our approach, we design a fitness function to evaluate the combinations of program entities effectively through a genetic algorithm integrated with a simulated annealing algorithm. As for this example, our approach ranks Lines 14 and 17 at 5th and 4th position, respectively.

3. Our approach

In this section, we present our approach on multi-fault localization after necessary definitions.

Definition 1 (Test Suite)**.** $T = \{t_1, t_2, \ldots, t_n\}$ represents a test suite with t_i being the ith test case. Moreover, We use T_F and T_P to denote the sets of failing and passing test cases, respectively.

Definition 2 (Program Entities)**.** $E = \{e_1, e_2, \ldots, e_m\}$ represents a set of program entities. Each entity can be a statement, a function or a class. For test case t_i, t_{ij} gives whether the i^{th} test case covers the jth entity.

Definition 3 (Candidate Solutions)**.** $C = \{c_1, c_2, \ldots, c_m\}$ represents a candidate solution that is a set of entities. The value of c_j indicates whether the jth entity should be assumed faulty. For example, {0,1,0,1,0} indicate the second and fourth entities are assumed faulty while others are assumed correct.

3.1. Coverage information preprocessing

Before adopting the genetic algorithm, our approach has a preprocessing step to reduce the complexity of the coverage data as large-scale programs may have a very large coverage matrix. Spectrum-based fault localization approaches depend on the execution coverage information, therefore two statements with the same coverage and two test cases with the same execution coverage paths are indistinguishable during ranking. In this case, we merge adjacent entities with the same coverage to form a single program entity.

An entity that is never covered by any failed test case has little chance to be identified faulty by a spectrum-based fault localization approach. Since these entities can increase the search space, we exclude these entities before executing our genetic algorithm. After executing the genetic algorithm, these excluded entities are added to the final ranking list in the order of their calculation. More details will be given in Section 3.4.

3.2. Framework overview

Traditional spectrum-based fault localization approaches often calculate a suspicious score for every program entity. Different from those approaches, our framework starts with potential candidates and then evaluates them by a fitness function. We assume that the entities with value 1 indicate the existence of multiple faults. Exploring all possible candidates is hard due to exponential computational complexity. To solve the problem within a limited amount of time, we propose to use the genetic algorithm to search the best candidates in the whole search space.

Genetic algorithm (GA) is an adaptive heuristic search algorithm (Mitchell, 1998) based on the ideas of natural selection and genetics. It is widely used for generating solutions to optimization problems with complex search space. In GA, a population of candidate solutions to an optimization problem is evolved toward better solutions. Optimal solutions can be found after a certain number of iterations.

GA commonly has four procedures including “Initialization”, “Selection”, “Generation” and “Termination”. In the initialization procedure, an initial candidate population is generated according to different strategies (Winter et al., 1996). In the selection procedure, a fitness function is used for evaluating the fitness value for each candidate. Those candidates which have higher fitness value will be better ones. Only the candidates whose fitness values are high enough are selected for the next procedure. In the generation procedure, crossover and mutation occur and new candidates are generated, evaluated and added to the population. The selection and generation procedures form a loop until the termination procedure terminates the iterations.

Based on the genetic algorithm,FSMFL consists of four components “population initialization”, “candidates evaluation”, “next population generation” and “termination criterion setting”. First, a strategy is chosen to initialize the population, and candidates in the population are evaluated by a fitness function. Then, new populations are generated through selection, crossover, mutation, accepting and replacement operators. Finally, a termination criterion is used to determine whether to evolve the current population. Fig. 2 shows the structure of the proposed framework. Details of every component will be discussed in sequence in the remainder of this section.

The pseudo-code of FSMFL is given in Algorithm 1, which corresponds to the four aforementioned four procedures. The parameter values used in the pseudo-code will be discussed later.

3.3. Population initialization components

In FSMFL, any strategy can be used to create the first generation of candidate solutions. In practice, a random strategy usually performs well (Winter et al., 1996). The random strategy produces an initial population that every program entity has the equal possibility to be faulty. That is, in the initial candidate every entity has the equal possibility to be set to 0 or 1. In Fig. 2, different strategies mean different implementations are used to store genetic sequence data. Therefore, we randomly generate candidates and add them to the initial population until the number of candidates reaches the predefined population size. In the following population evolution phase, the candidates are evaluated by a fitness function.

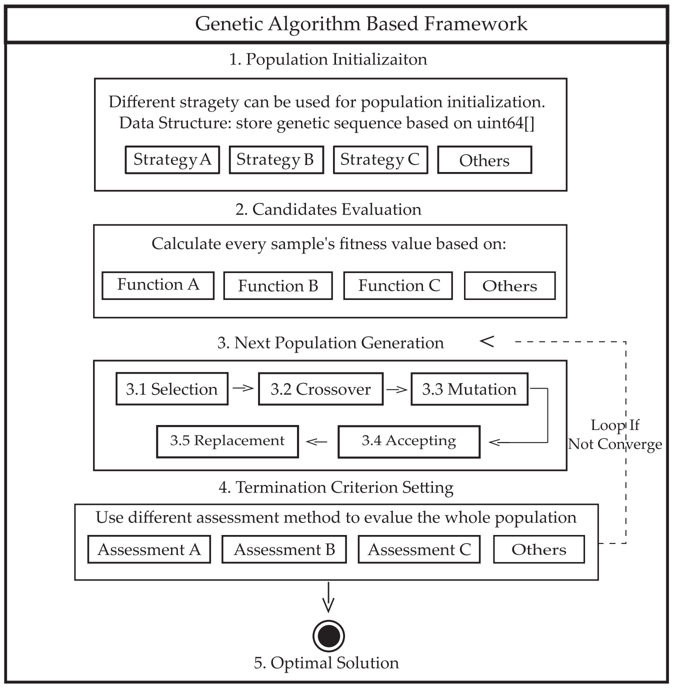

Genetic Algorithm Based Framework
1. Population Initializaiton
Different stragety can be used for population initialization.
Data Structure: store genetic sequence based on uint64[]
Strategy A
Strategy B
Strategy C
Others
2. Candidates Evaluation
Calculate every sample's fitness value based on:
Function A
Function B
Function C
Others
3. Next Population Generation
3.1 Selection
3.2 Crossover
3.3 Mutation
3.4 Accepting
3.5 Replacement
Loop If Not Converge
4. Termination Criterion Setting
Use different assessment method to evalue the whole population
Assessment A
Assessment B
Assessment C
Others
5. Optimal Solution

Fig. 2. Structure of FSMFL.

Algorithm 1 Fast Software Multi-fault Localization Framework.

```
Input: population size $\alpha$, crossover rate $\gamma$, mutation rate $q$, iteration count $\delta$
Output: optimal population consists of best candidates
1: // generate feasible solutions randomly and save them into population Pop
2: $population \leftarrow InitializeRandomly()$    ▷ Initialize feasible candidate
3: $SortByEvaluating(population)$    ▷ Evaluate and sort for selecting
4: // initial annealing probability and decrease 0.05 after every generation
5: $\varepsilon \leftarrow 0.9$    ▷ Initial simulated annealing probability to 0.9
6: for $i = 1$ to $\delta$ do    ▷ Loop after iterating $\delta$ times
7:     $new \leftarrow nextGeneration(population, \gamma, q, \varepsilon)$    ▷ update population
8:     if terminable($population, new$) then    ▷ check if terminal
9:         return $population$
10:    end if
11:    $population \leftarrow new$
12:    if $\varepsilon > 0.1$ then    ▷ decrease if larger than 0.1
13:        $\varepsilon = \varepsilon - 0.05$    ▷ decrease $\varepsilon$ every iteration
14:    end if
15: end for
16: return $population$    ▷ returning the best candidate set after evolution
```

3.4. Candidate evaluation components

Fitness function plays an important role in candidate evaluation. Our fitness function is described as follows.

When there is only one fault in a program, all failed test cases should cover that faulty entity. So, the faulty entity has the highest n_{ef} that equals to n_f. The suspicious score is proportional to n_{ef} when two program statements have the same n_{ep} for the six fault localization formulas listed in Table 2. So, the faulty entity can have a good ranking. However, it is not certain that faulty entities have the highest n_{ef} when there are multiple faults (because some correct entities may be covered by more test cases). However, subject to that any candidate must cover all failed test cases, candidates with faulty entities have higher suspicious score than the correct entities. Considering this situation, an aggregation evaluation strategy is exploited by our fitness function. We define some relevant notions before presenting the fitness function.

First, a weight value for every entity s is defined. This is similar to Ochiai because we also think that an entity is more likely to be faulty when it is covered by more failed test cases.

$$weight(s) = \frac{n_{ef}(s)}{\sqrt{n_f \times \left(n_{ef}(s) + n_{ep}(s)\right)}} \tag{1}$$

Based on these weight values, the aggregated failure ratio $f(C)$ and the aggregated passing ratio $p(C)$ of a candidate C can be calculated by the following formulas. The term $weight(s)$ indicates the importance of an entity, $n_{ef}(e_i)$ represents its coverage by failed test cases and c_i represents whether a candidate includes this entity. We multiply them and the result $f(C)$ indicates this candidate's weighed coverage on failed test cases. The same is with $p(C)$.

$$f(C) = \sum_{1\le i\le m} c_i \times n_{ef}(e_i) \times weight(e_i) \tag{2}$$

$$p(C) = \sum_{1\le i\le m} c_i \times n_{ep}(e_i) \times weight(e_i) \tag{3}$$

Finally, the suspicious score of a candidate C is defined as $S(C)$, which is proportional to $f(C)$ and inversely proportional to $p(C)$. The numerator and denominator are added by 1 at the same time to avoid the division-by-zero exception.

$$S(C) = \frac{1 + f(C)}{1 + f(C) + p(C)} \tag{4}$$

There exist many invalid candidates in the search space. To reduce these invalid candidates, we restrict that any candidate must be covered by all the failed test cases. That is, the following *Coverage*(C) equals to 1 for a candidate C. Otherwise, the fitness function will be zero. Note that our framework is capable of integrating other effective fitness functions.

$$Coverage(C) = \frac{1}{n_f} \sum_{1\le i\le n, t_i\in T_F} min\left(1, \sum_{1\le j\le m} t_{ij} \times c_j\right) \tag{5}$$

3.5. Next population generation components

This component focuses on how to generate the next population in our framework. There are 5 basic steps: Selection, Crossover, Mutation, Accepting and Replacement. Algorithm 2 gives the pseudo-code.

In the Selection step, a rank selection operator is utilized to select parents for the next crossover step, in which both a single point crossover operator and a shuffle crossover operator are applied (Mitchell, 1998).

Generally speaking, the crossover operator ensures that every bit of candidates has a chance to be mutated by a very low rate in the mutation step (Mitchell, 1998; Winter et al., 1996). The purpose of mutation, which introduces random modification, is to maintain diversity within the population and inhibit premature convergence. Different mutation operators can be used to achieve the goal. The most commonly used operator, i.e. single bit mutation, is used in the experiments. The mutation rate is empirically set to 0.05 in practice, which can achieve competitive performance in our experiments.

In the Accepting phase, candidates are evaluated using the fitness function to decide whether they are good enough to survive in the new population. In the last replacement step, we use the newly generated population for a further run of the algorithm. Meanwhile, a probabilistic technique named "Simulated Annealing" (SA) (Khachaturyan et al., 1979; 1981) is adopted for searching global optimal solution rather than a premature solution. SA makes the algorithm more robust when searching in a large search space and prevents the algorithm from becoming stuck at a local minimum. We use ε to denote the acceptance probability of SA. The higher the value of ε is, the more likely a bad solution is accepted. Through accepting bad solutions, SA keeps the diversity of the population and allows for a more extensive search for the optimal solution. Inspired by the optimization in SA (Kirkpatrick et al., 1983), the parameter ε is empirically set to 0.9 at the beginning and is decreased by a factor of 0.05 after every iteration until ε becomes 0.1. Pseudo-code is given in Algorithm 3.

Algorithm 2 nextGeneration(*population*, γ, q, ε).

```
Input: current population population, crossover rate $\gamma$, mutation rate $q$, simulated annealing possibility $\varepsilon$
Output: next generation: nextPopulation
1: $count \leftarrow 0, size \leftarrow length(population)$
2: $nextPopulation = \{p \mid p \in population\}$
3: $lowerBound = minimumFitness(Population)$
4: while $count < (2 * size)$ do    ▷ children is twice the size of parent
5:     $(Parent_A, Parent_B) \leftarrow random\_select(population)$
6:     $(Child_A, Child_B) \leftarrow crossover(Parent_A, Parent_B, \gamma)$
7:     $Child_A \leftarrow mutate(Child_A, q)$    ▷ Mutate child with probability $q$
8:     $Child_B \leftarrow mutate(Child_B, q)$
9:     $check2accept(nextPopulation, lowerBound, Child_A, Child_B, \varepsilon)$    ▷ check if accept the child
10:    $count = count + 1$
11: end while
12: return $nextPopulation$    ▷ returning the best candidate set after evolution
```

Algorithm 3 check2accept(*population, lowerBound, Child*$_A$, *Child*$_B$, ε).

```
Input: population is the current candidates set, lowerBound is the threshold used for testing whether to accept the new candidates($Child_A, Child_B$), with respect to simulated annealing probability $\varepsilon$.
1: if $fitness(Child_A) >= lowerBound$ then
2:     $population \leftarrow population \cup \{Child_A\}$
3: else if $random() < \varepsilon$ then
4:     $population \leftarrow population \cup \{Child_A\}$
5: end if
6: if $fitness(Child_B) >= lowerBound$ then
7:     $population \leftarrow population \cup \{Child_B\}$
8: else if $random() < \varepsilon$ then
9:     $population \leftarrow population \cup \{Child_B\}$
10: end if
```

3.6. Termination criterion setting

In the last process (Steimann and Bertschler, 2009b; Steimann and Frenkel, 2012b), a termination criterion is predefined to determine whether a solution (i.e., current population) is good enough or still needs to be further evolved. In each iteration, the total fitness value of the current population is compared against that in the previous generation to find out whether a better solution is found. The algorithm continues until no better offspring is generated in a specific number of continuous iterations. If the termination criterion is satisfied, algorithm stops and the current solution is treated as the best population.

The termination criterion we use is the number of continuous iterations in which no better solution is generated. In our experiments, we set the number empirically to 50 and this value achieves satisfactory results in our empirical study.

3.7. Multiple faults localization for FSMFL

After the genetic algorithm component, the last surviving population contains a set of candidates. The following process is adopted. First, these candidates are sorted by their fitness values in the descending order. Multiple entities, which are contained in a candidate, are sorted by their values of *weight*(s). Finally, the final ranking list is constructed by adding entities from the candidate list one by one according to its first appearance.

4. Empirical studies

We design three research questions for our experiments.

RQ1: Is FSMFL better than existing approaches ?

For this research question, we compare our approach with seven single-fault localization approaches, including Tarantula (Jones et al., 2002), Ochiai (Abreu et al., 2006), OP2 (Naish et al., 2011), DStar (Wong et al., 2014), GP13 (Yoo, 2012), Ample (Dallmeier et al., 2005) and Jaccard (Chen et al., 2002). A Linear multi-fault localization approach (Dean et al., 2009) is chosen because it is also designed to evaluate the set of suspicious solutions which is similar to our candidate solution. Please refer to Dean et al. (2009) for specific calculation process of their method.

RQ2: Does FSMFL perform significantly better than existing approaches by using statistical hypothesis test methods?

FSMFL is designed for solving multi-fault localization problem. The first step to compare FSMFL with other baseline approaches is to verify that there indeed exists a significant difference between them. For this research question, three statistical hypothesis test methods, including ANOVA (Analysis of variance) (Friedman, 1937), LSD (Least Significant Difference) (Wilcoxon, 1992) and Bonferroni Correction (Benjamini and Yekutieli, 2001), are adopted to measure the difference between FSMFL and the other approaches.

RQ3: Is FSMFL's Efficiency acceptable ?

Effectiveness is a significant issue when we apply FSMFL in practice, especially for large-scale programs. We have optimized our approach in several ways to improve the efficiency. For this research question, we will measure time usage of our approach on all the subjects.

4.1. Experiment setup

In this subsection, an experiment is conducted to evaluate the diagnostic capability of our approach for real programs, and compare the effectiveness and efficiency against other baseline approaches. Our experiments are conducted on a Linux Server with a 3.00GHz Intel(R) Xeon(R) E5-2623 v3 CPU and 32GB physical memory. The operating system is CentOS 7.0 and the compiler is GCC version 3.8.5 and JDK 1.7.0_79. FSMFL and other existing fault localization approaches are implemented using the Go programming language. In the experiment, we compare FSMFL against Tarantula, Ochiai, OP2, DStar, GP13, Ample, Jaccard and Linear (Dean et al., 2009).

4.1.1. Subject programs

Our benchmarks consist of Siemens programs, Linux programs, Space program, JFreeChart program, Joda-Time program, Apache Commons Lang program, Apache Commons Math and Google's Closure program. The first three programs are downloaded from SIR (Do et al., 2005), the rest are downloaded from defects4j (Just et al., 2014). The Siemens programs have 174 to 539 lines of code (LOC), with 1052 to 5542 test cases in the test suite. The three programs for Linux are gzip (6576 LOC), grep (12635 LOC) and sed (7125 LOC). The Space program (9126 LOC) has 3814 test cases. JFreeChart program (52104 LOC) has 2193 test cases. Joda-Time program (13630 LOC) has 4041 test cases. Apache Commons Lang (11844 LOC) has 2291 test cases and Apache Commons Math (42684 LOC) has 4378 test cases and Google Closure program (47446 LOC) has 7911 test cases.

4.1.2. Fault injection and construction of real multi-fault versions

Empirical studies in software testing are usually difficult and unrealistic because real bugs are rarely used in software testing research (Just et al., 2014). Extracting and reproducing real bugs is challenging. As a result, manual faults or mutants generated by mutation testing are commonly used as a substitute.

In order to build manual test suites. Versions with single fault are downloaded from SIR (Do et al., 2005), while versions with multiple faults are constructed by manually injecting faults which are based on original faults.

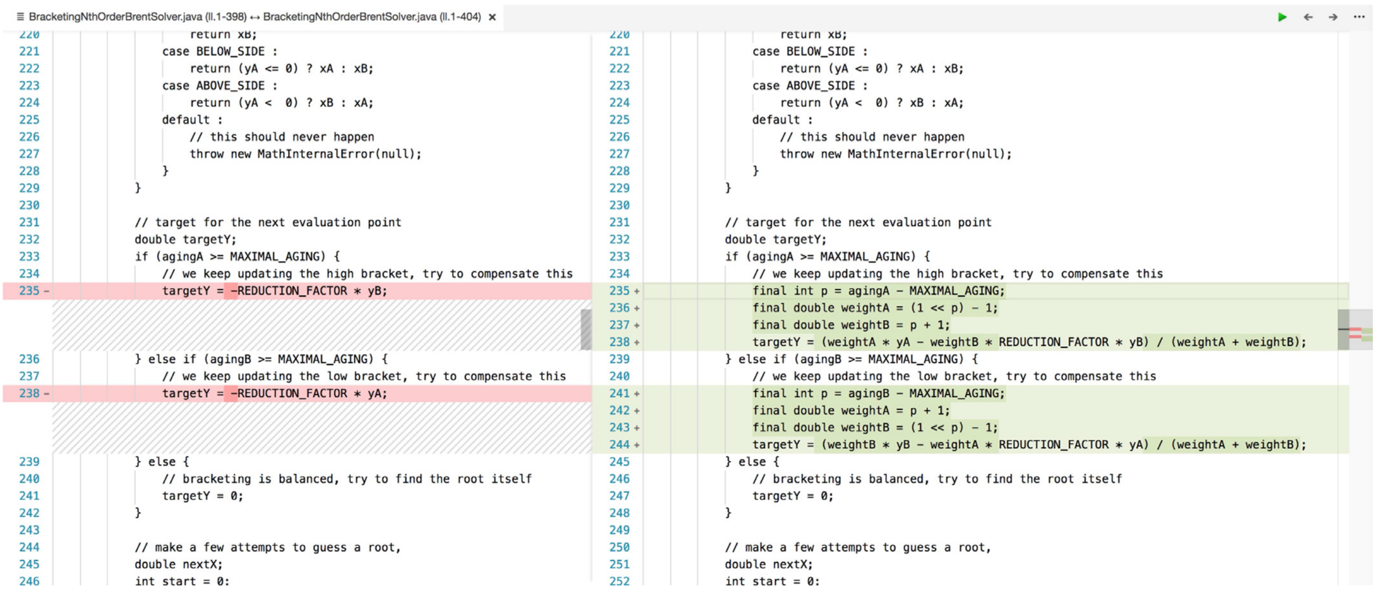
We also downloaded 5 real large-scale programs from Defects4j (Andrews et al., 2005). On average there are more than 30,000 LOCs. The programs are JFreechart, Closure compiler, Apache commons-lang, Apache commons-math and Joda-Time that are maintained by Google Company, Apache Foundation, etc. All have a number of real faults.

Defects4j collects both buggy and fixed program revisions for every fault. Figs. 3 and 4 show the detailed information of No.5 and No.40 bugs in Apache commons-math program. In both figures, the buggy version and faulty lines are shown on the left, and the fixed version is listed on the right. As shown in the figure, a new combined multi-buggy version can be obtained by concatenating the faulty lines of two buggy versions manually. Real multi-fault programs collected from defects4j are used for verifying the effectiveness of FSMFL.

We evaluate the effectiveness of approaches with single-fault on all programs, and with multi-fault on four Siemens programs (i.e., print_tokens, print_tokens2, replace and tot_info), three Linux programs Space program and five real programs in Defects4j. The number of bugs in multi-fault versions is two, three or five. The reason to exclude the other three Siemens programs in the multi-fault experiment is that they have too few executable lines to generate enough multi-fault versions. Table 3 shows the characteristics of the subject programs in terms of program name, source, the number of single-fault and multi-fault versions, LOC of the whole program, LOC covered by the test cases and the number of test case.

4.2. Optimization and implementation

We consider the following aspects to improve the performance of FSMFL: memory usage, parallel computation, and algorithmic improvement. Firstly, every candidate is represented by a bit string. We choose raw uint64 array as the basic data structure, by which

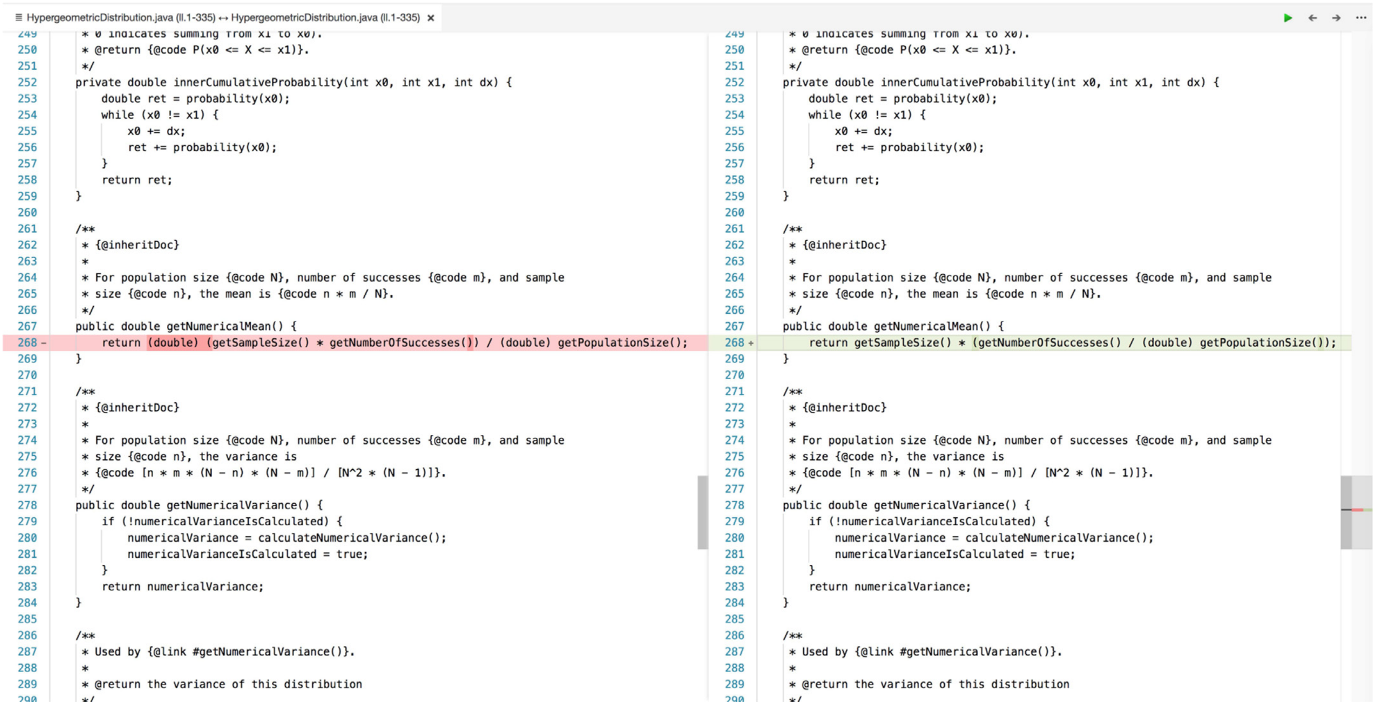


BracketingNthOrderBrentSolver.java (ll.1-398) ↔ BracketingNthOrderBrentSolver.java (ll.1-404)

```java
                return xB;
            case BELOW_SIDE :
                return (yA <= 0) ? xA : xB;
            case ABOVE_SIDE :
                return (yA <  0) ? xB : xA;
            default :
                // this should never happen
                throw new MathInternalError(null);
            }
        }

        // target for the next evaluation point
        double targetY;
        if (agingA >= MAXIMAL_AGING) {
            // we keep updating the high bracket, try to compensate this
-           targetY = -REDUCTION_FACTOR * yB;
        } else if (agingB >= MAXIMAL_AGING) {
            // we keep updating the low bracket, try to compensate this
-           targetY = -REDUCTION_FACTOR * yA;
        } else {
            // bracketing is balanced, try to find the root itself
            targetY = 0;
        }

        // make a few attempts to guess a root,
        double nextX;
        int start = 0:
```

```java
                return xB;
            case BELOW_SIDE :
                return (yA <= 0) ? xA : xB;
            case ABOVE_SIDE :
                return (yA <  0) ? xB : xA;
            default :
                // this should never happen
                throw new MathInternalError(null);
            }
        }

        // target for the next evaluation point
        double targetY;
        if (agingA >= MAXIMAL_AGING) {
            // we keep updating the high bracket, try to compensate this
+           final int p = agingA - MAXIMAL_AGING;
+           final double weightA = (1 << p) - 1;
+           final double weightB = p + 1;
+           targetY = (weightA * yA - weightB * REDUCTION_FACTOR * yB) / (weightA + weightB);
        } else if (agingB >= MAXIMAL_AGING) {
            // we keep updating the low bracket, try to compensate this
+           final int p = agingB - MAXIMAL_AGING;
+           final double weightA = p + 1;
+           final double weightB = (1 << p) - 1;
+           targetY = (weightB * yB - weightA * REDUCTION_FACTOR * yA) / (weightA + weightB);
        } else {
            // bracketing is balanced, try to find the root itself
            targetY = 0;
        }

        // make a few attempts to guess a root,
        double nextX;
        int start = 0:
```

Fig. 3. No.5 bug of Apache commons-math Program.



HypergeometricDistribution.java (ll.1-335) ↔ HypergeometricDistribution.java (ll.1-335)

```java
     * 0 indicates summing from x1 to x0).
     * @return {@code P(x0 <= X <= x1)}.
     */
    private double innerCumulativeProbability(int x0, int x1, int dx) {
        double ret = probability(x0);
        while (x0 != x1) {
            x0 += dx;
            ret += probability(x0);
        }
        return ret;
    }

    /**
     * {@inheritDoc}
     *
     * For population size {@code N}, number of successes {@code m}, and sample
     * size {@code n}, the mean is {@code n * m / N}.
     */
    public double getNumericalMean() {
-       return (double) (getSampleSize() * getNumberOfSuccesses()) / (double) getPopulationSize();
    }

    /**
     * {@inheritDoc}
     *
     * For population size {@code N}, number of successes {@code m}, and sample
     * size {@code n}, the variance is
     * {@code [n * m * (N - n) * (N - m)] / [N^2 * (N - 1)]}.
     */
    public double getNumericalVariance() {
        if (!numericalVarianceIsCalculated) {
            numericalVariance = calculateNumericalVariance();
            numericalVarianceIsCalculated = true;
        }
        return numericalVariance;
    }

    /**
     * Used by {@link #getNumericalVariance()}.
     *
     * @return the variance of this distribution
     */
```

```java
     * 0 indicates summing from x1 to x0).
     * @return {@code P(x0 <= X <= x1)}.
     */
    private double innerCumulativeProbability(int x0, int x1, int dx) {
        double ret = probability(x0);
        while (x0 != x1) {
            x0 += dx;
            ret += probability(x0);
        }
        return ret;
    }

    /**
     * {@inheritDoc}
     *
     * For population size {@code N}, number of successes {@code m}, and sample
     * size {@code n}, the mean is {@code n * m / N}.
     */
    public double getNumericalMean() {
+       return getSampleSize() * (getNumberOfSuccesses() / (double) getPopulationSize());
    }

    /**
     * {@inheritDoc}
     *
     * For population size {@code N}, number of successes {@code m}, and sample
     * size {@code n}, the variance is
     * {@code [n * m * (N - n) * (N - m)] / [N^2 * (N - 1)]}.
     */
    public double getNumericalVariance() {
        if (!numericalVarianceIsCalculated) {
            numericalVariance = calculateNumericalVariance();
            numericalVarianceIsCalculated = true;
        }
        return numericalVariance;
    }

    /**
     * Used by {@link #getNumericalVariance()}.
     *
     * @return the variance of this distribution
     */
```

Fig. 4. No.40 bug of Apache commons-math Program.

any length of chromosomes can be easily represented and the storage space can be tremendously reduced. Secondly, using Go language and parallel computing technique, we significantly improve computational efficiency and speed up the crossover and mutation steps. As for the selection process, a combination of the heap and quick sort is adopted to reduce time complexity of the algorithm to $O(\log n)$.

With the optimized genetic algorithm implementation, the framework can support large-scale search space. As for implementation, the population size is configured to 500 and every new generation generates 1000 children. A rank selection operator is used in the selection step. Both a single point crossover operator and a shuffle crossover operator are used in the crossover step. Single bit string mutation operator is adopted, which means each position has the same probability to be chosen to mutate. The crossover rate is set to 0.99 and mutation rate is set to 0.05. All the algorithm's parameters such as population size, crossover rate, mutation rate and etc. are empirically chosen according to the standard genetic algorithm (Winter et al., 1996). The value of these parameters can achieve competitive results in our empirical study.

4.3. Performance metrics for evaluation

In single-fault localization problem, the *EXAM* (Wong et al., 2008) metric is usually used to evaluate the effectiveness of the result. The metric measures the percentage of the entities one need to examine before finding the faulty entity. For a specific program, the localization result is better when the *EXAM* value is

Table 3
Program used in experiment.

Program	Source	#Single-Fault Version (#Multi-fault Version)	All LOC	Executable LOC	#Test Case
print_tokens	Siemens	20(33)	539	203	4130
print_tokens2	Siemens	20(36)	489	201	4115
replace	Siemens	21(45)	507	273	5542
schedule	Siemens	13	397	166	2650
schedule2	Siemens	14	299	146	2710
tcas	Siemens	18	174	73	1608
tot_info	Siemens	31(48)	398	138	1052
gzip	Linux	7(9)	6576	1744	213
grep	Linux	2(4)	12635	3197	470
sed	Linux	6(3)	7125	2027	360
space	space	37(39)	9126	3814	13585
lang	Defects4J	17(30)	11844	1391	2291
chart	Defects4J	9(5)	52104	1193	2193
time	Defects4J	6(4)	13630	1131	4041
math	Defects4J	13(6)	42684	936	4378
closure	Defects4J	3(1)	47446	5631	7911

smaller. However, *EXAM* metric cannot be applied to multi-fault localization problem that considers all fault locations at the same time. There are no general metrics to estimate the performance of multi-fault localization algorithms. Abreu et al. (2009) proposed a wasted effort metric to evaluate their multi-fault localization method. But the metric only considers the statements in the result set. Dean et al. (2009) only compared their *linear-based* algorithm against single fault localization algorithms rather than multi-fault localization algorithms. *Coverage-based* algorithms (Steimann and Bertschler, 2009b; Steimann and Frenkel, 2012b) do not attempt to locate all the faulty statements as FSMFL does. Hence, we propose two metrics ($EXAM_F$ and $EXAM_L$) based on *EXAM* to measure the performance of multi-fault localization algorithm in finding the first and the last faulty statements. Both metrics are used in empirical studies to verify the effectiveness of FSMFL compared with single-fault as well as multi-fault localization algorithms.

Definition 4 ($EXAM_F$). It represents the percentage of entities that have to be examined until the first faulty entity is found. For example, Tarantula's $EXAM_F$ can be calculated by 4/17 in Fig. 1.

Definition 5 ($EXAM_L$). It represents the percentage of entities that have to be examined until the last faulty entity is found. For example, Tarantula's $EXAM_L$ can be calculated by 10/17 in Fig. 1.

$EXAM_F$ is more useful in the scenario of finding a fault one at a time. On the other hand, $EXAM_L$ is more suitable in the scenario of finding all faults at the same time, such as finding faults for compiler programs. A smaller $EXAM_L$ means that the locations of all the faults have good rankings in the suspicious ranking list.

Even though $EXAM_L$ is able to indicate the percentile of the last faulty statement in the ranked list, it is insensitive to the faulty position in ranked list. To remedy the problem, we introduce a position based metric called *Top-N* to further evaluate the effectiveness of our approach.

Definition 6 (Top-N)**.** It represents the number of programs that an algorithm ranks all faulty statements among the top *N* positions in the ranked list.

Intuitively, the smaller the value of *N* in *Top-N*, the stricter the metric. For example, Top-3 metric requires that all faults are ranked within top 3 positions in the ranked list. Specifically, for the program shown in Fig. 1, Tarantula locates the last fault at the 10th position while FSMFL locates the last one at the 5th position. While both Tarantula and FSMFL pass the *Top-10* metric, Tarantula fails *Top-9*. Since *Top-N* metric is more sensitive to the fault positions in a ranked list, it is more effective than others when verifying the accuracy of fault localization. $EXAM_F$, $EXAM_L$ and *Top-N* are all used to compare the performance of our approach against others in our experiments.

4.4. Hypothesis testing methods

A group of experiments between our approach and existing approaches are conducted on programs with both single fault and multiple faults. Several statistical methods, including ANOVA, LSD, Bonferroni Correction, Wilcoxon Signed Rank test and Cohen's d test, are adopted to analyze the experimental results.

1. Analysis of variance (ANOVA) (Friedman, 1937) is used to analyze the difference among group means and their variance. ANOVA provides a statistical test on whether the means of several groups are equal. ANOVAs are useful for comparing three or more means for statistical significance.
2. Wilcoxon Signed Rank (WSR) (Wilcoxon, 1992) is a nonparametric test for comparing samples. It is used as an alternative to ANOVA when the normal assumption is not satisfied. It is useful for determining whether two dependent samples selected from populations have the same distribution.
3. Least Significant Difference (LSD) (Wilcoxon, 1992) is a measure of how much of a difference between means must be observed before one can draw a conclusion that the means are significantly different. However, this technique can be used only when ANOVA result is relatively significant.
4. Bonferroni Correction (Benjamini and Yekutieli, 2001) is used to counteract the problem of multiple comparisons.
5. Cohen's d (Sawilowsky, 2009) is adopted to measure the effective size of different datasets.

Experimental process includes the following:

1. Calculate the average *EXAM* value of FSMFL and other baseline approaches by executing each program 30 times.
2. Use ANOVA test method to analyze the difference between the means of all the approaches and compute *EXAM* values and variance. Meantime, LSD is used for measuring the degree of the difference between two specific approaches and decide which approach is better. Moreover, Bonferroni Correction is used to counteract the problem of multiple comparisons. Bonferroni is another multiple comparison method like LSD. It also can only be used when the ANOVA result is significant. The usage is similar to LSD, but Bonferroni cares more about the false discovery rate.
3. WSR is adopted in $EXAM_L$ to identify whether samples of different approaches come from different distributions. Cohen's d (Sawilowsky, 2009) is also used to measure the effectiveness

Table 4
ANOVA test result on single-fault version.

Source	Sum of Squares	Degrees of Freedom	Mean Squares	Chi-Square	P-Value
Columns	2702.28	8	337.785	451.83	$1.49622e^{-99}$
Error	8445.72	1856	4.55		
Total	11,148	2096			

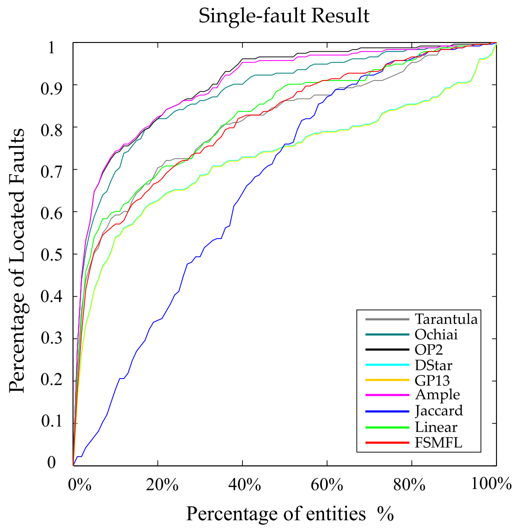


Fig. 5. Visualization of single-fault localization result.

between FSMFL and other baseline approaches in multi-fault versions (when considering $EXAM_L$).

4. Verifying the effectiveness of FSMFL against other baseline approaches by using *Top-N* metric and visualizing the $EXAM_F$, $EXAM_L$ and *TOP-N* experimental results.

4.5. Experiments for single-fault versions

Though FSMFL aims at solving the multi-fault localization problem, it can also be applied to programs with only a single fault. In this subsection, we compare the effectiveness of FSMFL algorithm with Tarantula, Ochiai and OP2. As shown in Table 3 we choose 189 single-fault versions of C programs from Siemens, Linux and space suite, 48 single-fault versions of Java programs from Defects4j. We execute 30 times for each program and calculate the average *EXAM*. Based on these *EXAM* values, following hypothesis tests are conducted to verify the effectiveness between FSMFL and other approaches.

4.5.1. Result analysis with single-fault versions

The results on single-fault versions are shown in Fig. 5. They give the percentage of faulty versions whose scores are within the specified segment. Each segment indicates one percentage, For example, 0%–1% or 43%–44%.

It can be observed from Fig. 5 that OP2, Ochiai, DStar, Ample have a better performance. FSMFL and Linear are close to Tarantula that can localize nearly same percentage of faults within 20%, 40%, 60% entities. Jaccard and GP13 have poor performance when handling single-fault problems.

Statistical hypothesis test results indicate that FSMFL does not have a worse performance than other single-fault localization approaches.

4.5.2. Hypothesis test with single-fault versions

ANOVA analysis is conducted among 9 approaches on *EXAM* values. Detailed results are shown in Table 4. The *p-value* is $1.49622e^{-99}$, far smaller than 0.05. According to hypothesis testing, we reject the original hypothesis based on 95% probability. Such a small *p-value* usually means the difference among 9 approaches is significant. More statistical hypothesis test experiments such as LSD and Bonferroni can be conducted for confirmation.

The LSD comparison experiments with 0.05 significant level are conducted and the results are shown in Fig. 6. The LSD value indicates the approach performance, the approach with low LSD value is better than the one with a high LSD value. In Fig. 6, 9 approaches (Tarantula, Ochiai, OP2, DStar, GP13, Ample, Jaccard, Linear, FSMFL) are represented in the y-axis. A smaller value in x-axis means the corresponding approach is better. Fig. 6 also shows the Bonferroni test result that has the similar criteria with LSD.

In Fig. 6, we can get the same result that Ochiai, OP2, DStar, GP13 and Ample belong to the first echelon, FSMFL, Linear and Tarantula belong to the second echelon and Jaccard gives the worst result. As the LSD result shows, although FSMFL does not have the best result among 9 approaches, it is still an acceptable and feasible approach even for single-fault localization.

Table 5 further shows the effectiveness on single-fault localization among these approaches. In the table, a check mark means that the approach above is better than the approach on the left. If an approach has more check marks in its column, it performs better in this metric. Similarly, if an approach has more check marks in its row, it performs worse in this metric. We get the same result that FSMFL, Linear and Tarantula belong to the second echelon. Thus, FSMFL can also be used to solve single-fault localization problem.

4.6. Experiments for multiple-fault versions

The main goal of FSMFL is for multi-fault localization. A set of experiments are conducted in this section. Specifically, $EXAM_F$, $EXAM_L$ and *Top-N* are used for evaluating and comparing the effectiveness between FSMFL and other baseline approaches. Thorough analysis of the experimental results is discussed to support the conclusion that FSMFL performs well in multi-fault localization, especially under $EXAM_L$ metric.

As shown in Table 3, there are totally 189 single-fault versions and 217 multi-fault versions of C programs (from Siemens, Linux and space), 48 single-fault versions and 46 multi-fault versions of Java programs (from Defects4j). We execute each program for 30 times and calculate the average $EXAM_F$ and $EXAM_L$ to compare the approaches. Based on the $EXAM_F$ and $EXAM_L$ values, the hypothesis test is conducted to verify the effectiveness of FSMFL. Moreover, *Top-N* metric is used to verify the accuracy and effectiveness of FSMFL.

4.6.1. Result analysis with multi-fault versions

Experiments using $EXAM_F$ and $EXAM_L$ metrics

As shown in Table 3, we evaluate the approaches by considering both $EXAM_F$ and $EXAM_L$. We execute each program 30 times by each of the 9 approaches and calculate the average of $EXAM_F$ and $EXAM_L$ values. Fig. 7 depicts the results of 9 different fault localization approaches on multi-fault versions. It can be observed that OP2 does not perform as well as in single-fault localization experiments.

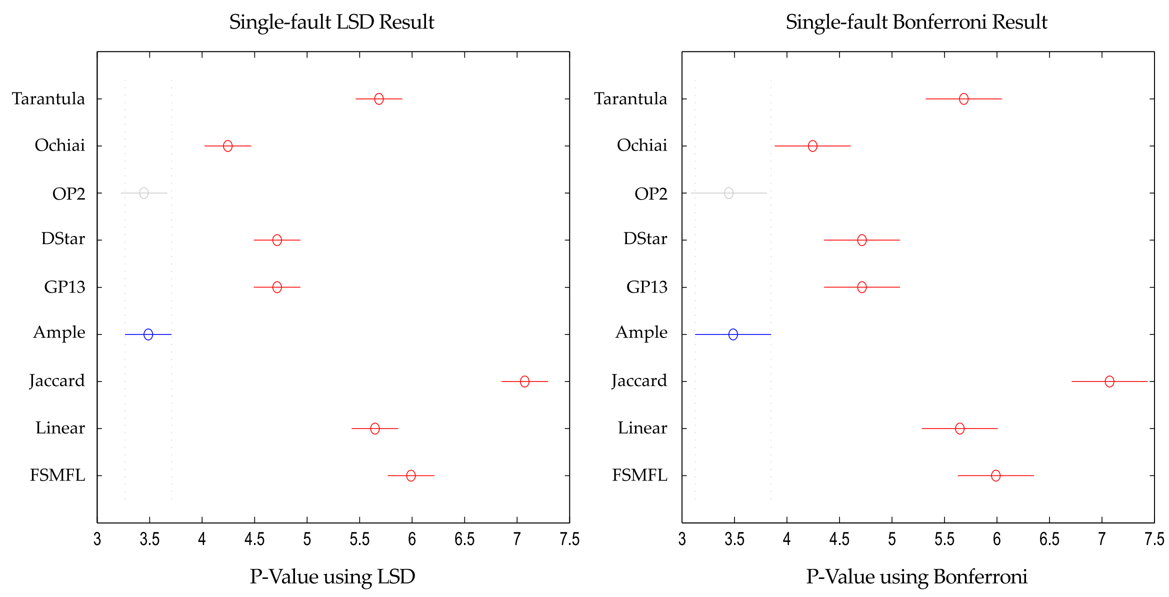


Fig. 6. LSD and Bonferroni result on single-fault versions.

Table 5
Multiple comparison for single-fault versions.

	Tarantula	Ochiai	OP2	Dstar	GP13	Ample	Jaccard	Linear	FSMFL
Tarantula		√	√	√	√	√			
Ochiai			√			√			
OP2									
Dstar			√			√			
GP13			√			√			
Ample									
Jaccard	√	√	√	√	√	√		√	√
Linear		√	√	√	√	√			
FSMFL		√	√	√	√	√			

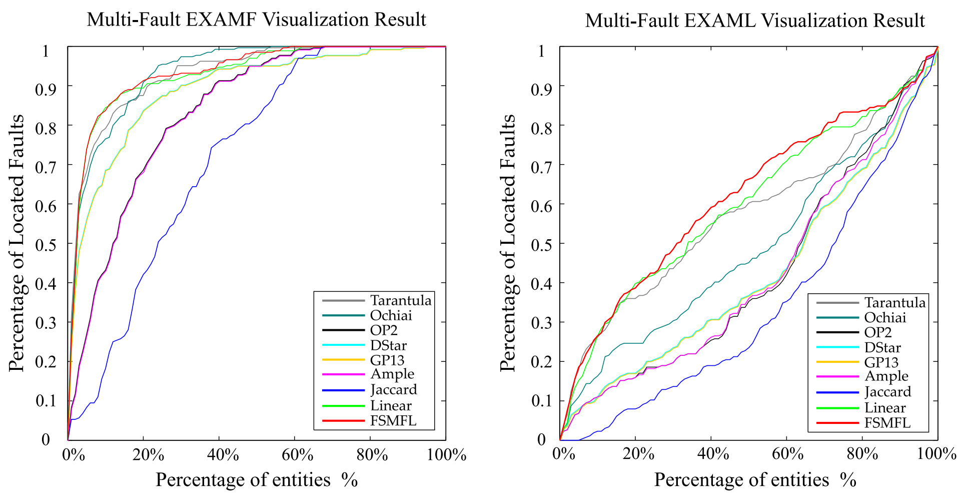


Fig. 7. Visualization of multi-fault localization result.

Sub-figures in the left of Fig. 7 shows that FSMFL works as well as other approaches (Tarantula, Ochiai, GP13, Ample, Linear) in finding the first fault in multi-fault problems using $EXAM_F$ metric. It indicates that FSMFL is a feasible approach for finding the first fault under multi-fault problems. As for the $EXAM_F$ experiments, it can be observed from the right of the Fig. 7 that FSMFL and Linear algorithm are better than other approaches.

For a further verification, statistical hypothesis test methods (ANOVA, LSD and Bonferroni) are adopted to check if the difference in performance is significant in the following section.

4.6.2. Hypothesis test with multiple-fault versions

ANOVA is conducted for hypothesis test of 9 approaches. The *p-value* of two *EXAM* metrics ($EXAM_F$, $EXAM_L$) are listed in Table 6 and 7. Both *p-values* ($4.28756 * e^{-99}$ and $6.09209 * e^{-89}$) are much less than 0.05, which indicates 9 approaches have significant difference no matter using $EXAM_F$ or $EXAM_L$.

Table 6
ANOVA test result on multi-fault versions using $EXAM_F$.

Source	Sum of Squares	Degrees of Freedom	Mean Squares	Chi-Square	P-Value
Columns	3123.9	8	390.482	482.36	$4.28756e^{-99}$
Error	10554.1	2104	5.016		
Total	13,678	2375			

Table 7
ANOVA test result on multi-fault versions using $EXAM_L$.

Source	Sum of Squares	Degrees of Freedom	Mean Squares	Chi-Square	P-Value
Columns	2681.98	8	335.248	434.98	$6.09209e^{-89}$
Error	10340.02	2104	4.914		
Total	13,022	2375			

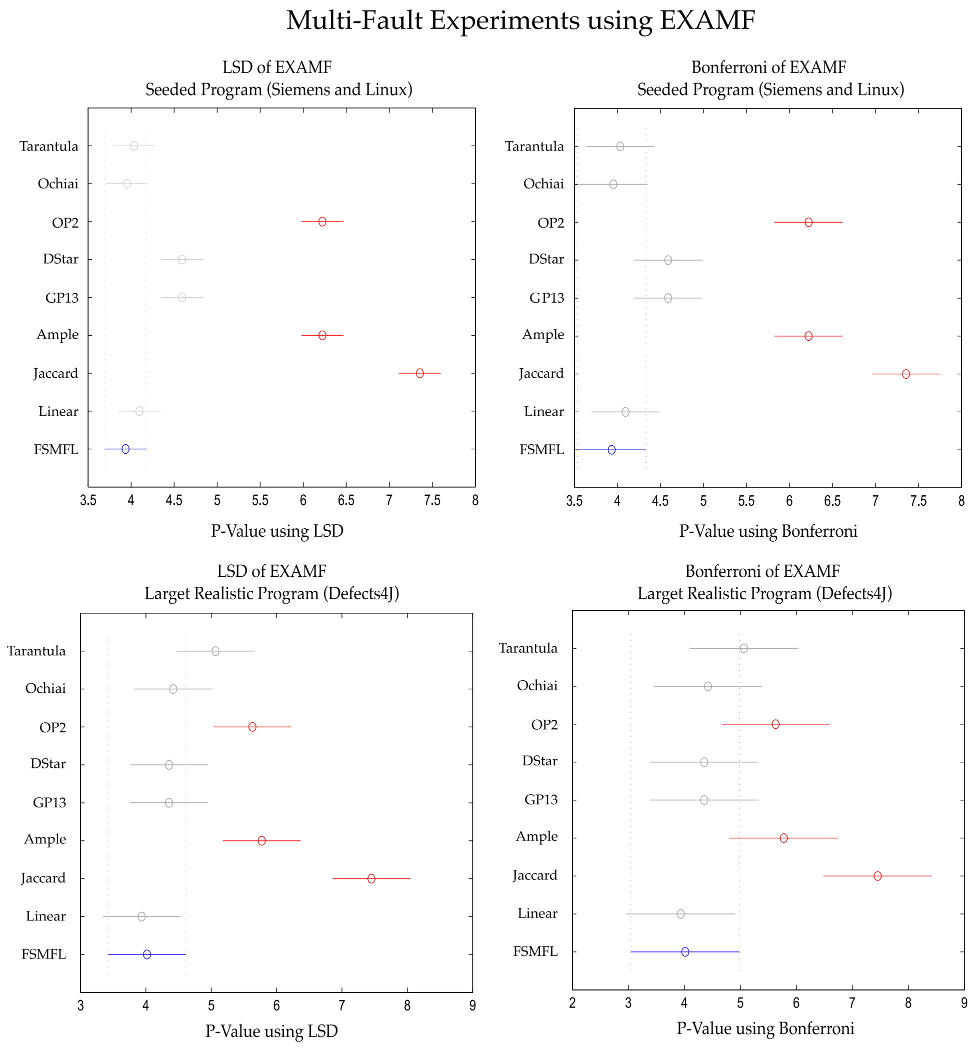


Fig. 8. LSD and Bonferroni result on multi-fault versions using $EXAM_F$.

LSD and Bonferroni test results are shown in Fig. 8 using $EXAM_F$ and Fig. 9 using $EXAM_L$. Both seeded programs (Siemens and Linux) and large-scale real programs(Defects4j) are used to verify FSMFL and other baseline approaches separately. We analyze the result in details in the following section.

Hypothesis test using $EXAM_F$ metric.

In Fig. 8, two sub-figures on the top give experimental results on seeded programs while two sub-figures at the bottom show the experimental results on large-scale real programs. Two sub-figures on the left are LSD test while two on the right are Bonferroni test. As previously explained, Tarantula, Ochiai, OP2, DStar, GP13, Ample, Jaccard, Linear, FSMFL are listed in the y-axis. The smaller value in x-axis means the better the corresponding approach. In Fig. 8, OP2, Ample and Jaccard have high mean values and are significantly different from the other 6 approaches.

The LSD and Bonferroni results indicate that OP2, Ample and Jaccard have worse performance in finding the first fault ($EXAM_F$) than the other no matter on seeded or real large-scale programs. FSMFL is similar to other 5 approaches. Although FSMFL is not the

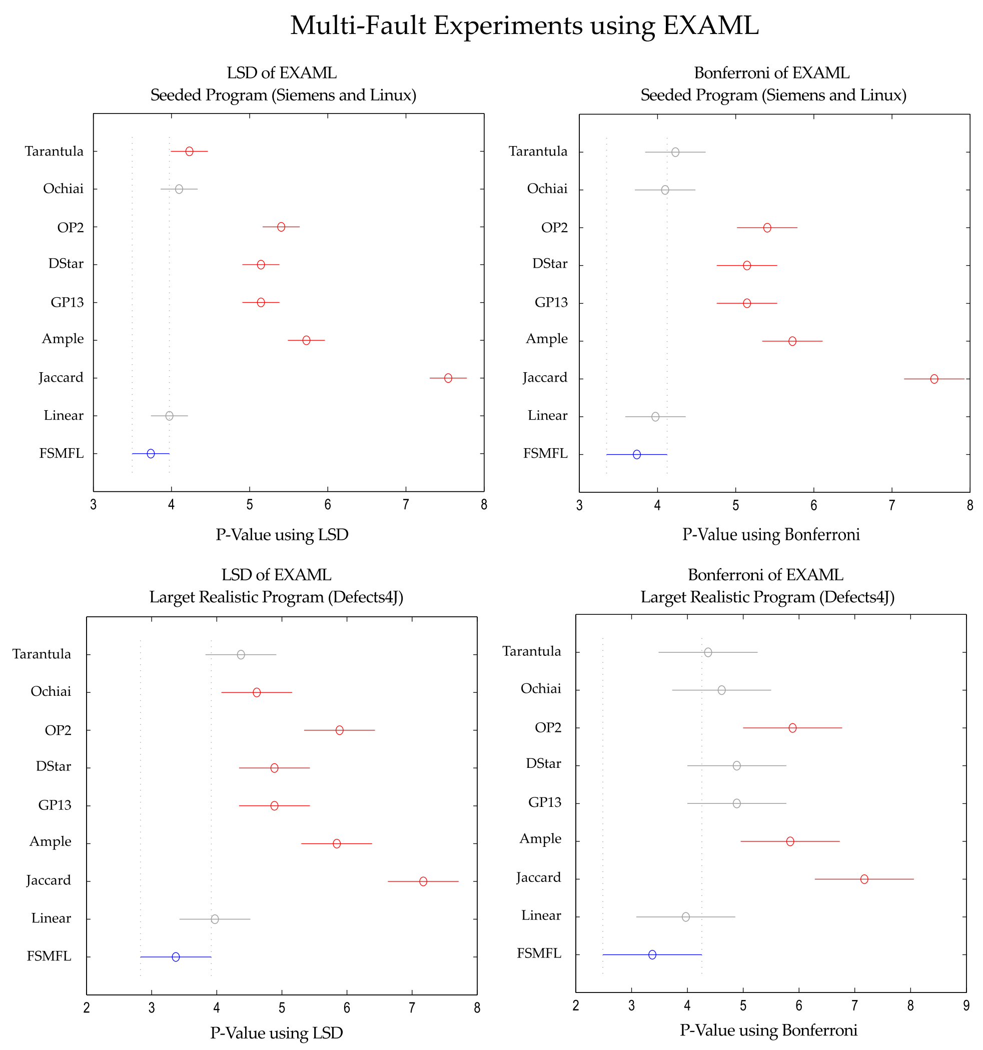


Fig. 9. LSD and Bonferroni result on multi-fault versions using $EXAM_L$.

Table 8
Multiple comparison for multi-fault versions using $EXAM_F$.

	Tarantula	Ochiai	OP2	Dstar	GP13	Ample	Jaccard	Linear	FSMFL
Tarantula									
Ochiai									
OP2	√	√		√	√			√	√
Dstar	√	√						√	√
GP13	√	√						√	√
Ample	√	√		√	√			√	√
Jaccard	√	√	√	√	√	√		√	√
Linear									
FSMFL									

best approach in $EXAM_F$, we can still find the first fault as effectively as the other 5 approaches in practice.

Table 8 shows the effectiveness measured by $EXAM_F$ among these approaches. It can be observed that FSMFL, Tarantula, Linear and Ochiai perform well in seeded programs while FSMFL and Linear perform best in both seeded and large-scale realistic programs.

Hypothesis test using $EXAM_L$ Metric

Two sub-figures on the top in Fig. 9 give experimental results on seeded programs using $EXAM_L$, and two at the bottom on large-scale real programs. Two sub-figures on the left show LSD test while two on the right Bonferroni test.

Fig. 9 indicates that both FSMFL and Linear approaches have good results and perform significantly better than other 7 approaches in terms of $EXAM_L$ metric. Jaccard, OP2 and Dstar have high mean value show in Fig. 9 and are significantly different from the other 6 approaches. The experimental results indicate FSMFL usually outperforms other approaches in terms of $EXAM_F$.

Table 9 gives a clear picture of comparison. The notations in the table have been described in Section 4.5. Hypothesis test indicates that FSMFL has similar performance with that of other approaches (Tarantula, Ochiai, GP13, Ample, Linear) in finding the first fault in multi-fault localization in terms of $EXAM_F$. Meanwhile, FSMFL and Linear outperform the rest in terms of $EXAM_L$.

Table 9
Multiple comparison for multi-fault versions using $EXAM_L$.

	Tarantula	Ochiai	OP2	Dstar	GP13	Ample	Jaccard	Linear	FSMFL
Tarantula		✓						✓	✓
Ochiai								✓	✓
OP2	✓	✓		✓	✓			✓	✓
Dstar	✓	✓						✓	✓
GP13	✓	✓						✓	✓
Ample	✓	✓	✓	✓	✓			✓	✓
Jaccard	✓	✓	✓	✓	✓	✓		✓	✓
Linear									✓
FSMFL									

Table 10
Wilcoxon signed rank test (p-value).

Tarantula	Ochiai	OP2	Dstar	GP13	Ample	Jaccard	Linear
$4.1548e^{-05}$	$2.7060e^{-07}$	$8.7571e^{-17}$	$1.1531e^{-15}$	$1.1531e^{-15}$	$2.2436e^{-19}$	$5.2730e^{-29}$	0.2178

Table 11
Effect size test (Cohen's d between FSMFL and other 8 approaches).

Tarantula	Ochiai	OP2	Dstar	GP13	Ample	Jaccard	Linear
0.131514	0.393969	0.637708	0.613346	0.613346	0.638572	0.953559	0.05974

For further verifying the effectiveness of FSMFL, a nonparametric test WSR is adopted. WSR does not assume the data are normal distributed. It can be used to determine whether two dependent samples are selected from populations having the same distribution. The p-value between FSMFL and other 8 approaches are listed in Table 10.

All the p-values are much smaller than 0.05 in Table 10 except the one with Linear. According to the empirical result, FSMFL is obviously different from other approaches except for Linear. As the p-value for Linear is 0.2178, the hypothesis that FSMFL and Linear are from different distribution does not hold. The similar result can be found from the effect size experiment shown in Table 11 where all the Cohen's d values between FSMFL and other 8 approaches are shown (Only the d-value between FSMFL and Linear is less than 0.1, which means only the difference between FSMFL and Linear is relatively small). Hence more experiments using various *Top-N* metrics are performed in the next section to verify whether FSMFL can locate all the faults faster than other approaches by considering only top *N* positions in the ranked list.

In summary, based on the analysis on the experimental results, we can draw a conclusion that FSMFL is more effective than other approaches (i.e., Tarantula, Ochiai, OP2, Dstar, GP13, Ample and Jaccard) in finding both the first and the last faulty statements at the same time. But the advantage of our approach over Linear is not significant. Hence, we conduct analysis by considering the *Top-N* metric to further verify the advantage of our approach over Linear.

4.6.3. *Experiments using* Top-N *metric*

Nowadays, a large-scale software development tends to be modularized, which encourages each function or class to follow the single responsibility principle (SRP) (Martin, 2002). SRP suggests that each function has only a single responsibility. Thus the functions that follow the principle usually have small LOC. As a result, most test cases usually cover only a small fraction of the source code. Fig. 10 depicts the test coverage information of the benchmarks used in our empirical studies. The results show that most test cases cover a small number of LOC (e.g., 0–200). Therefore the search space for fault localization is not very large if these programs follow SRP.

Table 12 shows the average length of the ranked lists that are produced by FSMFL. In Siemens programs, the average lengths range from 18 to 104. In Linux programs, the average lengths are about 150. In Defects4J, the average lengths range from 106 to 339. In space, the average is 624. The experiments reveal that the average length of the ranked lists is not very large.

Five metrics (*Top-3, Top-5, Top-8, Top-15, Top-25*) are used to verify the accuracy and effectiveness of the nine approaches. Table 13 summarizes the experimental results. Specifically, there are 19 programs that FSMFL succeeds in locating all faults by considering only the top 3 positions in ranked lists. Moreover, the number of programs that all faults are successfully located by FSMFL is always more than the others when using other 4 *Top-N* metrics. Fig. 11 shows that FSMFL outperforms other baseline

Table 12
Average length of ranked list that FSMFL suggested.

Program	Source	Average length of ranked list	ALL LOC
print_tokens	Siemens	94	539
replace	Siemens	104	507
schedule2	Siemens	53	299
tcas	Siemens	18	174
tot_info	Siemens	38	398
gzip	Linux	153	6578
grep	Linux	179	12,635
sed	Linux	127	7125
space	Space	624	9126
lang	Defects4J	339	11,844
chart	Defects4J	106	52,104
time	Defects4J	287	13,630
math	Defects4J	226	42,684
closure	Defects4J	328	47,446

Table 13
Number of programs that all faults are successfully located.

Methods	*Top-3*	*Top-5*	*Top-8*	*Top-15*	*Top-25*
Jaccard	0	0	0	1	2
OP2	0	0	0	1	6
DStar	4	6	7	13	18
Ample	0	0	0	1	3
Ochiai	7	10	11	18	21
DP13	4	6	7	13	18
FSMFL	**19**	**29**	**33**	**46**	**52**
Linear	7	11	12	14	20
Tarantula	14	20	25	31	34

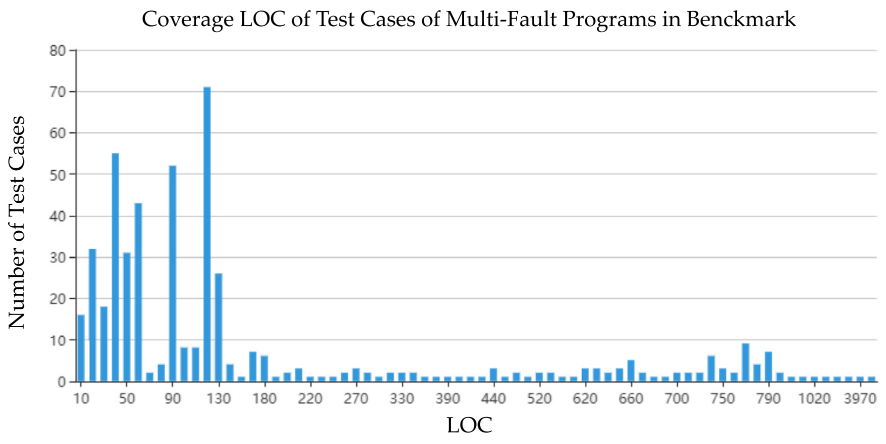


Fig. 10. LOC of failed test cases of the multi-fault programs in the benchmark.

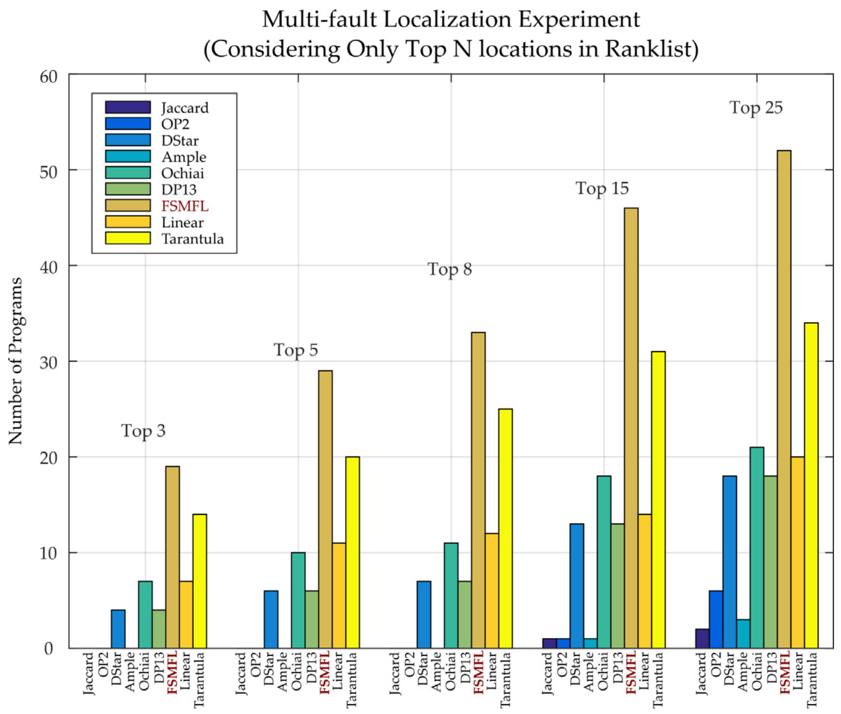


Fig. 11. Experiments of locating all faults. Five metrics (*Top-3*, *Top-5*, *Top-8*, *Top-15*, *Top-25*) are used to test the 9 approaches. Each bar indicates the number of programs with all faults located among top *N* positions.

approaches in locating all faults when using 5 *Top-N* metrics in a more intuitive way.

Fig. 12 depicts more experiments based on 200 metrics from *Top-1* to *Top-200*. The X-axis indicates the value of *N* in top-N, and the Y-axis represents the number of programs that all faults are located within top *N* positions. For instance, there are nearly 50 programs that FSMFL can locate all faults by considering only top 20 positions in its ranked lists. As expected, the experimental results show that the number of programs with all faults found grows if more positions in the ranked lists are considered. Among all the approaches, FSMFL outperforms all other 8 baseline approaches regardless the value of *N*. Based on the results, we can conclude that FSMFL is superior to other 8 approaches in finding all faults.

4.7. *Efficiency*

The time required for a fault localization approach includes two parts. Part I involves data collection. Part II uses the data collected in Part I to locate faults. All spectrum-based fault localization approaches require the same set of data, so they have same cost in Part I. Part II contains two processes. In the first process, time is spent on loading coverage information and collecting execution results. In the second process, time is spent on the computation and evolution of the genetic algorithm. Table 14 gives the computational time by Part II of FSMFL on Siemens, Linux and space and Defects4j suites. The computational time is the average of 100 executions for every faulty version of the program. It can be observed that the computational time is less than 23 s even for large-scale programs, which is acceptable in the real development environment.

It can also observe that the computational time increases with executable LOC (i.e. *LOC*) and the number of test cases (i.e. *TestCaseNumber*). We use the least-square method (Stigler, 1981) to perform a correlation test on all the executions and get a fit function as following:

$$Time = 6.12 \times LOC + 0.04 \times TestCaseNumber \qquad (6)$$

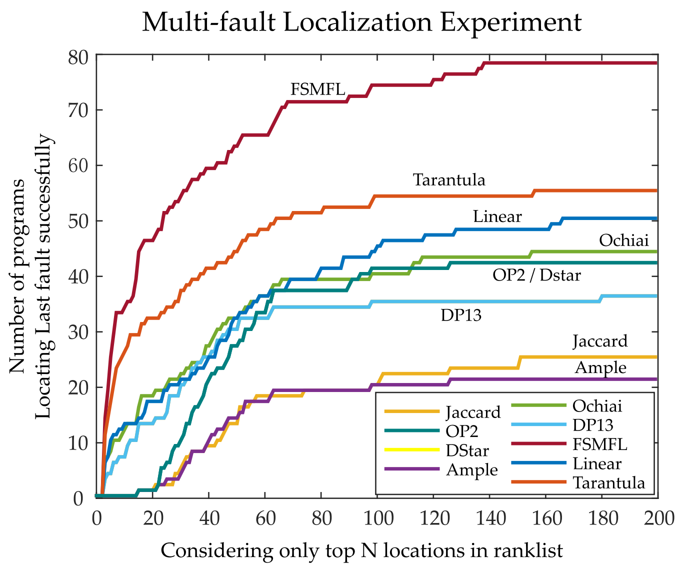


Fig. 12. Multi-fault localization experiments using 200 metrics.

Table 14
The computational time of locating multi-fault by Part II of FSMFL.

Program	Executable LOC	Test cases number	FSMFL using Go
tot_info	138	1052	1.15 s
print_tokens2	201	4115	1.76 s
print_tokens	203	4130	2.48 s
gzip	1744	213	2.02 s
sed	2027	360	1.97 s
replace	273	5542	3.47 s
grep	3197	470	2.03 s
space	3814	13585	22.97 s
lang	1391	2291	0.9 s
chart	1193	2193	0.6 s
time	1131	4041	1.1 s
math	936	4378	0.9 s
closure	5631	7911	12.9 s

4.8. Summary

For the RQ1, our approach is compared against Tarantula, Ochiai, OP2, DStar, GP13, Ample, Jaccard and Linear. The result shows that FSMFL has better performance on subjects with multiple faults and has similar performance on subjects with one fault. For RQ2, statistical hypothesis methods like ANOVA, LSD and Bonferroni Correction are used for measuring the difference between FSMFL and other baseline approaches. The results indicate that FSMFL is indeed significantly different from the others. For RQ3, we evaluate time usage of FSMFL and confirm that the cost is acceptable.

4.9. Threats to validity

In this subsection, we discuss the potential threats to validity in our empirical study.

Threats to external validity are about whether the observed experimental results can be generalized to other subjects. To guarantee the representativeness, we choose a large number of programs from the widely used Simens, Linux and Defects4J (Just et al., 2014; Andrews et al., 2005) suites. These subjects include large-scale programs with over 50,000 LOC, and industrial programs such as Apache Commons-Lang Apache Commons-Math, FreeChart and Closure Compiler. Our experiments consider both C and Java programs and include both artificial and real bugs. We realize that there is no perfect empirical study and there must be some tricky programs and bugs not considered by our experiments. We plan to enlarge our subjects in our future work.

Threats to internal validity are mainly concerned with the uncontrolled internal factors that might have an influence on the experimental results. The main internal threat is the potential errors in our implementation. To reduce this threat, pair programming is used and experimental results are carefully examined. Secondly, parameters are empirically chosen according to the standard GA algorithm (Mitchell, 1998). After testing different sets of parameters, we come to the conclusion that different parameter values have very little influence on the experimental results.

Threats to construct validity are about whether the performance metrics used in the empirical studies reflect the real-world situation. In real cases, programs commonly have more than one faults. We believe $EXAM_F$ is a reasonable metric to measure the effectiveness of locating the first fault. Similarly, $EXAM_L$ is a reasonable metric to measure the effectiveness of locating all the faults. In order to verify FSMFL is indeed statistically different from other baseline approaches, ANOVA, LSD (Wilcoxon, 1992) and Bonferroni Correction (Benjamini and Yekutieli, 2001) are adopted to conduct thorough experiments. We realize that design of effective metric to evaluate the effectiveness of multi-fault localization problem is still in its infancy. Therefore, designing more reasonable metric is our future work.

5. Related work

There is a large body of work on automatically localizing faulty statements in a piece of software. With the increase of software complexity, software faults become more difficult to be identified, which increases debugging cost (Wong et al., 2016). Dynamic fault localization (Jones et al., 2002; Xie et al., 2013) localizes faulty statements by leveraging the execution information from test case execution when the abnormal behavior was detected during testing. Among them, spectrum-based fault localization approaches have shown their effectiveness and efficiency in locating faults automatically (Wong et al., 2016). Spectrum-based fault localization works by ranking the program statements by their suspicious scores that can be calculated based on the spectrum information. Numerous spectrum-based fault localization approaches have been proposed to localize multiple faults and widely used.

Jones et al. (2007) introduced an approach to cluster failed test cases into groups, and localize each fault by different developers using single-fault localization approach. It is semi-automatic and its effectiveness is affected by the accuracy of clustering. In addition, Abreu and his colleagues (Abreu et al., 2009) proposed BARINEL that combines spectrum-based fault localization and model-based diagnosis. BARINEL uses faulty candidates and Bayesian Reasoning to deduce entity probabilities. Zhang et al. (2017) reported a comprehensive study that investigates the impact of cloning failed test cases on the effectiveness of SFL techniques. Perez et al. (2017) proposed a new metric, called DDU, for spectrum-based fault localization approaches with high accuracy. Moreover, Dean et al. (2009) presented a multi-fault localization approach based on a linear programming model. However, the suspicious function they used must be converted to a linear model. Thus, it cannot be used if a conversion to linear model is infeasible. Compared with their approach, ours uses a better suspicious function and both linear or non-linear models can be plugged into FSMFL. Steimann and Bertschler (2009a) and Steimann and Frenkel (2012a) proposed a coverage-based locator for multiple faults by assuming a probability distribution of the number of faults and then they apply optimization techniques to reduce algorithm complexity. However, the algorithm is exponential and the experiments confirm that it is not scalable. Gong et al. (2012) developed an indicator that can adapt existing single-fault localization approach to multi-fault localization. But in the process, it needs to interact with a programmer to identify the faults. The approach we proposed is a new multi-fault localization approach based on genetic algorithm. We design a weighted fitness function and re-implement the genetic algorithm using a different strategy.

Most existing fault localization approaches are optimized for programs with only one fault (Hamill and Goseva-Popstojanova, 2009; Thung et al., 2012). To address this limitation, some researchers have started to investigate the influence of multiple faults on the effectiveness of fault localization approaches (Debroy and Wong, 2009; DiGiuseppe and Jones, 2011; Xue and Namin, 2013). Their results imply that different faults may interfere with each other and significantly decrease the accuracy of the existing single-fault oriented localization approaches. However, the recent multi-fault localization approaches still focus on revising existing algorithms rather than redesigning new algorithms for multi-fault localization. For instance, Gong et al. proposed an indicator to determine whether to continue searching when one fault has been found (Gong et al., 2012). This mechanism tries to simplify the search space and stop the search procedure as soon as possible. Cheman et al. revealed that *Clean Program Assumption* does not always hold, which may raise a critical threat to the validity of existing fault localization approaches (Chekam et al., 2017). Song et al. believed that SFL is not suitable for all programs and the effectiveness of SFL should be evaluated before its adoption in practice (Song, 2014). Xuan et al. proposed a concept of spectrum driven test case purification based on existing approaches (like Tarantula) for improving fault localization (Xuan and Monperrus, 2014). Wu et al. suggested to locate fault by using the crash stack information in crash reports (Wu et al., 2014). With the help of specifications, Gopinath et al. improved the effectiveness of SFL (Gopinath et al., 2012). Structured Information Retrieval is adopted to improve the localization performance (Saha et al., 2013). Some other researches (Thung et al., 2012; Yoo et al., 2014; Kim and Lee, 2014) show that fault localization can facilitate debugging activities to some extent, but can still be improved in many aspects (Steimann et al., 2013). Tang et al. (2014, 2016) proposed some more practical fault localization approaches based on both coverage and version information. Sun and Podgurski (2016) investigated coverage-based statistical fault localization metrics and found the common properties of most effective metrics. Moreover, Tang et al. (2017) proposed an empirical framework of accuracy graphs to reveal the relative accuracy of formulas. This framework makes it possible to reveal accuracy relationships among the formulas which have not been discovered by theoretical analysis.

The discussion so far reveals that most of the recent work focus either on the effectiveness of applying SFL to real programs (Song, 2014; Pearson, 2016; Pearson et al., 2017; Xia et al., 2016) or on performance enhancement by various techniques (e.g. finish searching as soon as possible (Gong et al., 2012), preprocessing test cases (Chekam et al., 2017; Song, 2014; Xuan and Monperrus, 2014), utilizing additional information (Wu et al., 2014; Gopinath et al., 2012; Saha et al., 2013)) rather than developing new multi-fault localization approaches.

In the meantime, using meta-heuristic algorithms to solve software engineering problems is a popular idea. It has been successfully applied to solve problems throughout the software engineering lifestyle, especially in testing and debugging (Harman et al., 2012). Shin (Yoo, 2012) first used genetic programming to design risk evaluation formula for spectrum-based fault localization. Utilizing heuristic algorithm can give us a wider vision to solve the fault localization problem.

There is also a large body of work on automated program repair. Fault localization result is the basis of automatic program repair (Gong et al., 2012; Debroy and Wong, 2014), which aims at generating patches automatically to fix faults in software. As most automated program repair (APR) tools apply fault localization (FL) techniques to identify the locations of likely faults to be repaired. Assiri and Bieman (2017) conducted a controlled experiment to evaluate the impact of ten FL techniques on APR effectiveness, performance and repair correctness. According to their experimental results, The effectiveness, performance and correctness of APR depends on the FL method used. If FL does not identify the location of a fault, the application of an APR tool will not be effective and will fail to repair the fault. Moreover, automatic program repair result can evaluate the effectiveness of fault localization techniques (Qi et al., 2013). Recently, a new repair synthesis engine named S3 tackles patches involving multiple lines by grouping multiple buggy locations and synthesizing repairs for several locations at once (Le et al., 2017). The intuition is that it clusters buggy locations into groups by either locality or suspiciousness scores produced by fault localization. All the above studies indicate that FL is a vital pre-process in APR and it is crucial to locate all potential faults.

6. Conclusion

This paper presents FSMFL, a genetic algorithm based framework for multi-fault localization. A fitness function is designed to evaluate the combinations of program entities. Statistical hypothesis tests are conducted to compare FSMFL against other spectrum-based fault localization approaches. Experiments show that FSMFL is competitive in single-fault localization and superior in multi-fault localization. We have also developed optimization techniques to improve the efficiency of FSMFL. Our extensive experiments show that FSMFL is effective and efficient in multi-fault localization.

Acknowledgment

This work was partially supported by the National Natural Science Foundation of China (NO. 61202030,61202006,71502125).

References

Abreu, R., Zoeteweij, P., Van Gemund, A.J., 2006. An evaluation of similarity coefficients for software fault localization. In: Proceedings of International Conference on 12th Pacific Rim International Symposium on Dependable Computing (PRDC'06), pp. 39–46. doi:10.1109/PRDC.2006.18.

Abreu, R., Zoeteweij, P., Van Gemund, A.J., 2009. Spectrum-based multiple fault localization. In: Proceedings of 24th IEEE/ACM International Conference on Automated Software Engineering (ASE'09), pp. 88–99. doi:10.1109/ASE.2009.25.

Andrews, J.H., Briand, L.C., Labiche, Y., 2005. Is mutation an appropriate tool for testing experiments? [software testing]. In: Proceedings of the 27th international conference on Software engineering, pp. 402–411. doi:10.1109/ICSE.2005.1553583.

Assiri, F.Y., Bieman, J.M., 2017. Fault localization for automated program repair: effectiveness, performance, repair correctness. Software Q. J. 25 (1), 171–199. doi:10.1007/s11219-016-9312-z.

Beizer, B., 1990. Software Testing Techniques (2Nd Ed.). Van Nostrand Reinhold Co., New York, NY, USA.

Benjamini, Y., Yekutieli, D., 2001. The control of the false discovery rate in multiple testing under dependency. Ann. Stat. 29 (4), 1165–1188.

Chekam, T.T., Papadakis, M., Traon, Y.L., Harman, M., 2017. An empirical study on mutation, statement and branch coverage fault revelation that avoids the unreliable clean program assumption. In: Proceedings of the 39th International Conference on Software Engineering. IEEE Press, Piscataway, NJ, USA, pp. 597–608. doi:10.1109/ICSE.2017.61.

Chen, M.Y., Kiciman, E., Fratkin, E., Fox, A., Brewer, E., 2002. Pinpoint: problem determination in large, dynamic internet services. In: Proceedings of International Conference on Dependable Systems and Networks., pp. 595–604. doi:10.1109/DSN.2002.1029005.

Dallmeier, V., Lindig, C., Zeller, A., 2005. Lightweight bug localization with ample. In: Proceedings of the sixth international symposium on Automated analysis-driven debugging. ACM, New York, NY, USA, pp. 99–104. doi:10.1145/1085130.1085143.

Dean, B.C., Pressly, W.B., Malloy, B.A., Whitley, A.A., 2009. A linear programming approach for automated localization of multiple faults. In: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE'09), pp. 640–644. doi:10.1109/ASE.2009.54.

Debroy, V., Wong, W.E., 2009. Insights on fault interference for programs with multiple bugs. In: Proceedings of 20th International Symposium on Software Reliability Engineering (ISSRE'09), pp. 165–174. doi:10.1109/ISSRE.2009.14.

Debroy, V., Wong, W.E., 2014. Combining mutation and fault localization for automated program debugging. J. Syst. Software 90, 45–60. doi:10.1016/j.jss.2013.10.042.

DiGiuseppe, N., Jones, J.A., 2011. On the influence of multiple faults on coverage-based fault localization. In: Proceedings of the international symposium on software testing and analysis. ACM, New York, NY, USA, pp. 210–220. doi:10.1145/2001420.2001446.

Do, H., Elbaum, S., Rothermel, G., 2005. Supporting controlled experimentation with testing techniques: an infrastructure and its potential impact. Emp. Software Eng. 10 (4), 405–435. doi:10.1007/s10664-005-3861-2.

Friedman, M., 1937. The use of ranks to avoid the assumption of normality implicit in the analysis of variance. J. Am. Stat. Assoc. 32 (200), 675–701.

Gong, C., Zheng, Z., Zhang, Y., Zhang, Z., Xue, Y., 2012. Factorising the multiple fault localization problem: adapting single-fault localizer to multi-fault programs. In: Proceedings of 19th Asia-Pacific Conference on Software Engineering Conference (APSEC'12), 1. IEEE, pp. 729–732. doi:10.1109/APSEC.2012.22.

Gopinath, D., Zaeem, R.N., Khurshid, S., 2012. Improving the effectiveness of spectra-based fault localization using specifications. In: Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering. ACM, New York, NY, USA, pp. 40–49. doi:10.1145/2351676.2351683.

Hamill, M., Goseva-Popstojanova, K., 2009. Common trends in software fault and failure data. IEEE Trans. Software Eng. 35 (4), 484–496. doi:10.1109/TSE.2009.3.

Harman, M., Mansouri, S.A., Zhang, Y., 2012. Search-based software engineering: trends, techniques and applications. ACM Comput. Surv. (CSUR) 45 (1), 11:1–11:61. doi:10.1145/2379776.2379787.

Jones, J.A., Bowring, J.F., Harrold, M.J., 2007. Debugging in parallel. In: Proceedings of the international symposium on Software testing and analysis (ISSTA'07). ACM, New York, NY, USA, pp. 16–26. doi:10.1145/1273463.1273468.

Jones, J.A., Harrold, M.J., Stasko, J., 2002. Visualization of test information to assist fault localization. In: Proceedings of the 24th international conference on Software engineering. ACM, New York, NY, USA, pp. 467–477. doi:10.1145/581339.581397.

Just, R., Jalali, D., Ernst, M.D., 2014. Defects4j: a database of existing faults to enable controlled testing studies for java programs. In: Proceedings of the International Symposium on Software Testing and Analysis. ACM, New York, NY, USA, pp. 437–440. doi:10.1145/2610384.2628055.

Khachaturyan, A., Semenovskaya, S., Vainstein, B., 1979. A statistical-thermodynamic approach to determination of structure amplitude phases. Sov. Phys. Crystallogr 24, 519–524.

Khachaturyan, A., Semenovsovskaya, S., Vainshtein, B., 1981. The thermodynamic approach to the structure analysis of crystals. Acta Crystallograph. Section A 37 (5), 742–754. doi:10.1107/S0567739481001630.

Kim, J., Lee, E., 2014. Empirical evaluation of existing algorithms of spectrum based fault localization. In: Proceedings of International Conference on Information Networking (ICOIN), pp. 346–351. doi:10.1109/ICOIN.2014.6799702.

Kirkpatrick, S., Gelatt, C.D., Vecchi, M.P., 1983. Optimization by simulated annealing. Science 220 (4598), 671–680. doi:10.1126/science.220.4598.671.

Le, X.-B.D., Chu, D.-H., Lo, D., Le Goues, C., Visser, W., 2017. S3: Syntax- and semantic-guided repair synthesis via programming by examples. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering. ACM, New York, NY, USA, pp. 593–604. doi:10.1145/3106237.3106309.

Martin, R.C., 2002. Agile Software Development: Principles, Patterns, and Practices. Prentice Hall.

Mitchell, M., 1998. An Introduction to Genetic Algorithms. MIT Press, Cambridge, MA, USA.

Naish, L., Lee, H.J., Ramamohanarao, K., 2011. A model for spectra-based software diagnosis. ACM Trans. Software Eng. Methodol. (TOSEM) 20 (3), 11:1–11:32. doi:10.1145/2000791.2000795.

Pearson, S., 2016. Evaluation of fault localization techniques. In: Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. ACM, New York, NY, USA, pp. 1115–1117. doi:10.1145/2950290.2983967.

Pearson, S., Campos, J., Just, R., Fraser, G., Abreu, R., Ernst, M.D., Pang, D., Keller, B., 2017. Evaluating and improving fault localization. In: Proceedings of the 39th International Conference on Software Engineering. IEEE Press, Piscataway, NJ, USA, pp. 609–620. doi:10.1109/ICSE.2017.62.

Perez, A., Abreu, R., van Deursen, A., 2017. A test-suite diagnosability metric for spectrum-based fault localization approaches. In: Proceedings of the 39th International Conference on Software Engineering. IEEE Press, Piscataway, NJ, USA, pp. 654–664. doi:10.1109/ICSE.2017.66.

Qi, Y., Mao, X., Lei, Y., Wang, C., 2013. Using automated program repair for evaluating the effectiveness of fault localization techniques. In: Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'13). ACM, New York, NY, USA, pp. 191–201. doi:10.1145/2483760.2483785.

Saha, R.K., Lease, M., Khurshid, S., Perry, D.E., 2013. Improving bug localization using structured information retrieval. In: Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on. IEEE, pp. 345–355.

Sawilowsky, S.S., 2009. New effect size rules of thumb. J. Modern Appl. Stat. Methods 8 (2), 467–474. doi:10.22237/jmasm/1257035100.

Song, S., 2014. Estimating the effectiveness of spectrum-based fault localization. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. ACM, New York, NY, USA, pp. 814–816. doi:10.1145/2635868.2661680.

Steimann, F., Bertschler, M., 2009. A simple coverage-based locator for multiple faults. In: Proceedings of International Conference on Software Testing Verification and Validation (ICST'09), pp. 366–375. doi:10.1109/ICST.2009.24.

Steimann, F., Bertschler, M., 2009. A simple coverage-based locator for multiple faults. In: Software Testing Verification and Validation, 2009. ICST'09. International Conference on. IEEE, pp. 366–375.

Steimann, F., Frenkel, M., 2012. Improving coverage-based localization of multiple faults using algorithms from integer linear programming. In: Proceedings of IEEE 23rd International Symposium on Software Reliability Engineering (ISSRE'12), pp. 121–130. doi:10.1109/ISSRE.2012.28.

Steimann, F., Frenkel, M., 2012b. Improving coverage-based localization of multiple faults using algorithms from integer linear programming. In: Software Reliability Engineering (ISSRE), 2012 IEEE 23rd International Symposium on. IEEE, pp. 121–130.

Steimann, F., Frenkel, M., Abreu, R., 2013. Threats to the validity and value of empirical assessments of the accuracy of coverage-based fault locators. In: Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'13). ACM, New York, NY, USA, pp. 314–324. doi:10.1145/2483760.2483767.

Stigler, S.M., 1981. Gauss and the invention of least squares. Ann. Stat. 9 (3), 465–474.

Sun, S.-F., Podgurski, A., 2016. Properties of effective metrics for coverage-based statistical fault localization. In: Proceedings of IEEE International Conference on Software Testing, Verification and Validation (ICST), pp. 124–134. doi:10.1109/ICST.2016.31.

Tang, C.M., Chan, W., Yu, Y.-T., 2014. Extending the theoretical fault localization effectiveness hierarchy with empirical results at different code abstraction levels. In: Proceedings of IEEE 38th Annual Conference on Computer Software and Applications Conference (COMPSAC), pp. 161–170. doi:10.1109/COMPSAC.2014.24.

Tang, C.M., Chan, W., Yu, Y.T., Zhang, Z., 2017. Accuracy graphs of spectrum-based fault localization formulas. IEEE Trans. Reliab. 66 (2), 403–424. doi:10.1109/TR.2017.2688487.

Tang, C.M., Keung, J., Yu, Y.-T., Chan, W., 2016. Dfl: dual-service fault localization. In: Proceedings of IEEE International Conference on Software Quality, Reliability and Security (QRS), pp. 412–422. doi:10.1109/QRS.2016.53.

Thung, F., Lo, D., Jiang, L., et al., 2012. Are faults localizable? In: Proceedings of the 9th IEEE Working Conference on Mining Software Repositories, pp. 74–77. doi:10.1109/MSR.2012.6224302.

Wilcoxon, F., 1992. Individual Comparisons by Ranking Methods. Springer New York, New York, NY, pp. 196–202.

Winter, G., Periaux, J., Galan, M., Cuesta, P., 1996. Genetic Algorithms in Engineering and Computer Science, 1st John Wiley & Sons, Inc., New York, NY, USA.

Wong, E., Wei, T., Qi, Y., Zhao, L., 2008. A crosstab-based statistical method for effective fault localization. In: Proceedings of 1st International Conference on Software Testing, Verification, and Validation, pp. 42–51. doi:10.1109/ICST.2008.65.

Wong, W.E., Debroy, V., Gao, R., Li, Y., 2014. The dstar method for effective software fault localization. IEEE Trans. Reliab. 63 (1), 290–308. doi:10.1109/TR.2013.2285319.

Wong, W.E., Gao, R., Li, Y., Abreu, R., Wotawa, F., 2016. A survey on software fault localization. IEEE Trans. Software Eng. 42 (8), 707–740. doi:10.1109/TSE.2016.2521368.

Wu, R., Zhang, H., Cheung, S.-C., Kim, S., 2014. Crashlocator: locating crashing faults based on crash stacks. In: Proceedings of the 2014 International Symposium on Software Testing and Analysis. ACM, New York, NY, USA, pp. 204–214. doi:10.1145/2610384.2610386.

Xia, X., Bao, L., Lo, D., Li, S., 2016. "automated debugging considered harmful" considered harmful: a user study revisiting the usefulness of spectra-based fault localization techniques with professionals using real bugs from large systems. In: Proceedings of IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 267–278. doi:10.1109/ICSME.2016.67.

Xie, X., Chen, T.Y., Kuo, F.-C., Xu, B., 2013. A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization. ACM Trans. Software Eng. Methodol. (TOSEM) 22 (4), 31:1–31:40. doi:10.1145/2522920.2522924.

Xie, X., Liu, Z., Song, S., Chen, Z., Xuan, J., Xu, B., 2016. Revisit of automatic debugging via human focus-tracking analysis. In: Proceedings of IEEE/ACM 38th International Conference on Software Engineering (ICSE), pp. 808–819. doi:10.1145/2884781.2884834.

Xuan, J., Monperrus, M., 2014. Test case purification for improving fault localization. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. ACM, pp. 52–63.

Xue, X., Namin, A.S., 2013. How significant is the effect of fault interactions on coverage-based fault localizations? In: Proceedings of ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, pp. 113–122. doi:10.1109/ESEM.2013.22.

Yoo, S., 2012. Evolving human competitive spectra-based fault localisation techniques. In: Proceedings of the 4th International Conference on Search Based Software Engineering. Springer-Verlag, Berlin, Heidelberg, pp. 244–258. doi:10.1007/978-3-642-33119-0_18.

Yoo, S., Xie, X., Kuo, F.-C., Chen, T. Y., Harman, M., 2014. No pot of gold at the end of program spectrum rainbow: greatest risk evaluation formula does not exist. UCL Research Notes RN/14/14.

Zhang, L., Yan, L., Zhang, Z., Zhang, J., Chan, W., Zheng, Z., 2017. A theoretical analysis on cloning the failed test cases to improve spectrum-based fault localization. J. Syst. Software 129, 35–57. https://doi.org/10.1016/j.jss.2017.04.017.