Journal of Software Engineering and Applications, 2016, 9, 412-423
Published Online August 2016 in SciRes. http://www.scirp.org/journal/jsea
http://dx.doi.org/10.4236/jsea.2016.98027

Statistical Debugging Effectiveness as a Fault Localization Approach: Comparative Study

Ishaq Sandoqa[1], Fawaz Alzghoul[1], Hamad Alsawalqah[1], Isra Alzghoul[1], Loai Alnemer[1], Mohammad Akour[2]

[1]Department of Computer Information Systems, The University of Jordan, Amman, Jordan
[2]Department of Computer Information System, Yarmouk University, Irbid, Jordan
Email: isandoqa@yahoo.com, Fawaz@ju.edu.jo, h.sawalqah@ju.edu.jo, e.zghoul@ju.edu.jo, l.nemer@ju.edu.jo, mohammed.akour@yu.edu.jo

Received 4 June 2016; accepted 16 August 2016; published 19 August 2016

Abstract

Fault localization is an important topic in software testing, as it enables the developer to specify fault location in their code. One of the dynamic fault localization techniques is statistical debugging. In this study, two statistical debugging algorithms are implemented, SOBER and Cause Isolation, and then the experimental works are conducted on five programs coded using Python as an example of well-known dynamic programming language. Results showed that in programs that contain only single bug, the two studied statistical debugging algorithms are very effective to localize a bug. In programs that have more than one bug, SOBER algorithm has limitations related to nested predicates, rarely observed predicates and complement predicates. The Cause Isolation has limitations related to sorting predicates based on importance and detecting bugs in predicate condition. The accuracy of both SOBER and Cause Isolation is affected by the program size. Quality comparison showed that SOBER algorithm requires more code examination than Cause Isolation to discover the bugs.

Keywords

Testing and Debugging, Dynamic Language, Statistical Debugging, Fault Localization

1. Introduction

Dynamic programming languages can be considered as programming languages that have a strong expressive power. The main characteristic of such languages is that they have no type declaration. On contrary, when using static languages, the programmer has to define variable types in the declaration and these types are checked

How to cite this paper: Sandoqa, I., Alzghoul, F., Alsawalqah, H., Alzghoul, I., Alnemer, L. and Akour, M. (2016) Statistical Debugging Effectiveness as a Fault Localization Approach: Comparative Study. *Journal of Software Engineering and Applications*, **9**, 412-423. http://dx.doi.org/10.4236/jsea.2016.98027

during compilation time. In dynamic languages, type checking of objects and variables is performed during runtime [1].

Dynamic programming, like Python, refers to languages that handle compilation problems at runtime. In general, dynamic languages are more flexible, the programs written using dynamic language are easier to modify and maintain. Moreover, it helps developers to be more productive. Another advantage of dynamic languages is that it provides memory dynamically according to the requirements of the variables. Variables and objects that not reached during the program execution will never allocate memory [2]-[5]. The main drawback of the dynamic languages is the performance. It is less efficient compared to static languages, because the high processes perform at runtime [2] [3]. Python programming language has a dynamic typing system. It provides both imperative and object oriented programming. Code written in Python is executed by a managed runtime execution environment. Finally, it is one of the most famous dynamic programming languages.

Software debugging is a continuous process in the coding phase. Fault localization is the main activity in software debugging. It is a diagnostic process that performs after a bug or failure occurs during program execution. Traditional fault localization approaches, such as printing a trace of a bug and then following the trace to localize the fault, are less effective in localizing the correct cause of bug. Moreover, it consumes much effort and resources in order to locate the bug [6]. Therefore, automated fault localization techniques were introduced in order to enhance fault localization process and reduce resources consumption.

Typically, two steps can be followed in order to localize bugs in the source code in dynamic programming languages [7]: first, analyzing the trace of program execution to track down the cause of bug; then, observing the unexpected program behaviors. Software debugging approaches can be classified into two categories:

- Static analysis in which the debugger analyses the source code of the program. The main drawback of static analysis approach is that it produces a high rate of false positive [8].
- Dynamic analysis in which bugs tracking is performed by analyzing the program behaviors during the runtime. Dynamic analysis involves comparing program behaviors in both of correct runs and incorrect runs.

Statistical debugging is a debugging technique based on dynamic analysis approach. It involves using statistical models to analyze the program behaviors during the execution. In this technique, locating bugs and covering the source code can be tracked down in more accurate way [9]. There are different dynamic analysis algorithms:

1) Tarantula algorithm which produced by [10]. It is based on coverage information obtained from test suite to discover likely faulty code.

2) Cause transition algorithm was produced by [11]. It is based on binary search process for memory state, they use both passing and failing test cases to find likely faulty code.

3) Cause Isolation algorithm and SOBER algorithm are based on predicate instrumentation in the source code, and analyzing the predicates evaluations during both of correct and incorrect program executions [8] [12]. The major difference between these two algorithms is that Cause Isolation measures only the predicates that evaluate to true during both of passed and failed test cases, while SOBER measures the differences between both true and false evaluations for each predicate.

In this paper we addressed the following main questions:

1) How to adapt SOBER and Cause Isolation Scheme algorithms in Python programming language?

2) What is the effectiveness in term of fault localization of two studied algorithms on programs written in dynamic programming language?

3) What are the limitations of using SOBER and Cause Isolation algorithms on programs written in dynamic programming language?

The reminder of this paper is organized as follows: Section 2 presents the research work related to statistical debugging. Section 3 presents the studied statistical debugging algorithms. Section 4 presents the experimental setup. Section 5 reports and presents a discussion of the experiment's results. Section 6 presents a comparison of the studied algorithms. Section 7 concludes the paper with a summary and an outlook on future research direction.

2. Related Work

First Cause Isolation algorithm is one of statistical debugging algorithms. In [12], Liblit *et al*. applied Cause Isolation algorithm on a Siemens suite, which is a set of programs written in C programming language. They showed that Cause Isolation algorithm was able to detect 52 faults out of 130 by examining 10% of the source

code. Another statistical debugging algorithm is SOBER. In [8], C. Liu *et al*. applied SOBER on Siemens suite. They showed that SOBER was able to detect 68 faults out of 130 by examining 10% of the source code.

Jiang and Su in [7] presented their statistical debugging algorithm, Context Aware Debugging. They described it as Context aware debugging is a combination of CBI infrastructure, feature selection, clustering in machine learning and control flow graph analysis. CBI (Cooperative Bug Isolation) [13] is an infrastructure based on analyzing the predicates records received from users after using a predicates instrumented program. Context Aware debugging mixed a CBI with machine learning algorithms (Support Vector Machine, Random Forests and K-Means) in order to localize bugs in a program. Context Aware Debugging was applied on Siemens suite and rhythm box (a large music management application written using C programming language). The results showed that Context Aware debugging algorithm was able to detect 67 faults out of 130 by examining 10% of the source code.

Bayesian debug algorithm was produced in [14]. The main idea behind Bayesian debug algorithm is that it does not need many program executions to run; it only needs one failed execution and one passed execution. Bayesian debug algorithm was applied on grep 2.2 program (pattern matcher program written in C programming language). It compared with SOBER and Cause Isolation algorithms. Results showed that in most cases Bayesian debug algorithm ranked the bug related predicates much higher than SOBER and Cause Isolation and processing through it was faster than other two.

In the dynamic programming languages domain, Akhter A. and Azha H. applied both of SOBER and Cause Isolation algorithm on a set of programs written using RUBY programming language. They showed that SOBER algorithm is more efficient and accurate in fault localization than Cause Isolation algorithm [15].

3. Prepare Studied and Implemented Statistical Debugging Algorithms

In this section, we will present the Cause Isolation algorithm and the SOBER algorithm in details.

3.1. Cause Isolation Algorithm

In [12], Liblit *et al*. proposed their statistical debugging algorithm Cause Isolation, and they present it as an algorithm able to isolate causes of bugs in a program containing more than one bug. In this algorithm, the program under test records its behavior during many executions, and gives a feedback report for each run. This report denotes as **R** consist of 1 bit indicating whether the execution is failed or passed in addition to a bit vector contains one bit for every predicate p in the program. If predicate p observed to be true at least one time in the execution then $\mathbf{R}(p) = 1$, otherwise $\mathbf{R}(p) = 0$. Cause Isolation algorithm considers three types of predicates to be recorded in feedback report:

• Branches: At each branching point in the program, either explicit branching such as if else blocks, or implicit branching such as loops and short-circuiting logical operators, two predicates are tracked to indicates whether true or false branch was executed.

• Returns: Function returned values are tracked, once a scalar-return function called, six predicates are tracked to indicate whether the returned value is <0, ≤ 0, >0, ≥ 0, $=0$, or $=0$.

• Scalar Pairs: At each scalar assignment statement $x=$, for each same typed variable y or constant c in the scope, six predicates are tracked indicating the relationship between new value of x and y or c. They indicates whether $x <, \leq, >, \geq, =$, or $=y$ or c.

After recording predicates during many runs, Cause Isolation algorithm ranks them in order to determine, for each bug, the most correlated predicate. **Algorithm 1** shows a pseudo code for Cause Isolation algorithm. Ranking process performed as the following: For every predicate p over the set of all runs, let $\mathbf{S}(p)$ refers to a count of successful runs where predicate p observed to be true, $\mathbf{F}(p)$ refers to a set of failing runs where predicate p observed to be true. The algorithm calculates **Failure**(p) as a conditional probability of program crashes given that p observed to be true, so it is given by Equation (1):

$$\mathbf{Failure}(p) = \frac{\mathbf{F}(p)}{\mathbf{F}(p) + \mathbf{S}(p)} \tag{1}$$

Although **Failure**(p) is a useful measure, it is not good enough to determine the main cause of the bug; some predicates may get a high failure score just because the predicate that cause the failure is located before it.

Therefore, Cause Isolation algorithm calculates another measure to increase the ranking accuracy, which is **Context**(p). **Context**(p) depends on observation of predicate p, regardless to its value (true or false). p_{observed} means that the program execution reached predicate p and evaluated it. **Context**(p) calculated as a conditional probability of program crashes given that p is observed using Equation (2).

$$\mathbf{Context}(p) = \frac{\mathbf{F}(p_{observed})}{\mathbf{F}(p_{observed}) + \mathbf{S}(p_{observed})} \tag{2}$$

Now, Cause Isolation algorithm calculates the rank of predicate p, as shown in Equation (3). Once all predicates are ranked, the highest score predicate p_1 is discarded, so all runs where $\mathbf{R}(p_1 = 1)$ are discarded, and the algorithm applied recursively on the remaining runs.

$$\mathbf{Increase}(p) = \mathbf{Failure}(p) - \mathbf{Context}(p) \tag{3}$$

Consider the following Python code snippet as an example of Cause Isolation algorithm:

```
1. f = value
2. if f is none:
3. x = 0
4. f. some Function
```

In the code snippet, consider predicate p_1 (f is none) in line (2), which would be recorded as a branch predicate. Clearly, this predicate is very bug relevant. The program will crash whenever this predicate is true. So, **Failure** (p_1) will evaluate to be 1. To understand the importance of context measurement, consider the predicate p_2 (x = 0), which will recorded as a scalar predicate. **Failure**(p_2) will evaluate to be 1 too. But the difference between p_1 and p_2 lies in the context. In fact, **Context**(p_2) will be evaluated to be 1, thus, the value of **Increase** (p_2) will be (1 − 1 = 0). Therefore, the algorithm will not rank p_2 as a bug relevant predicate.

3.2. SOBER Algorithm

In [8], C. Liu *et al*. proposed their statistical debugging algorithm SOBER as an enhancement of Cause Isolation algorithm. They showed that in some cases, Cause Isolation algorithm fails to determine a predicate that cause a bug. As mentioned in [8], the potential problem in Cause Isolation algorithm is that it does not consider the false evaluations of predicates, so it loses its discrimination power in some cases where **Failure**(p_1) is close to **Context**(p), which ranks predicate p as bug irrelevant, but in fact predicate p is strongly relevant to the bug. Therefore, SOBER algorithm considers both true and false evaluations of predicates, and then statistically evaluates the difference between true and false evaluations of predicate p in fail and successful runs. If the evaluation of predicate p in successful runs significantly differs from its evaluation in fail runs then predicate p is a candidate to be bug relevant predicate [8]. **Algorithm 2** shows a pseudo code for SOBER algorithm.

As shown in **Algorithm 2**, SOBER ranking process is performed as the following: For a given program ρ, let $\mathbf{T} = \{\mathbf{t}_1, \mathbf{t}_2, \cdots, \mathbf{t}_n\}$ is a set of test cases for ρ, each test case has an input and expected output. **T** could be divided into two distinct sets, $\mathbf{T}_f$ which contains the failed test cases that its actual output is not identical to expected output, and $\mathbf{T}_p$ which contain the passed test cases. Let $\mathbf{N}_t$ is the number of times that the predicate p evaluates to be true, and $\mathbf{N}_f$ is the number of times that it evaluates to be false in one execution. Then the evaluation bias of predicate p, $\boldsymbol{\pi}(p)$ is given in Equation (4):

$$\boldsymbol{\pi}(\mathbf{p}) = \frac{\mathbf{N_f}}{\mathbf{N_f} + \mathbf{N_t}} \tag{4}$$

Given a random test case t from **T**, let x be the random variable for the evaluation bias of predicate p from the execution of t. Let $\mathbf{f}(x|\theta p)$ and $\mathbf{f}(x|\theta f)$ be the probability density functions for the evaluation bias of predicate p on $\mathbf{T}_p$ and $\mathbf{T}_f$ respectively. Then, the difference between $f(x|\theta p)$ and $f(x|\theta f)$ indicates the bug relevancy of predicate p. So if $\mathbf{L}(p)$ was a similarity function such that:

$$\mathbf{L}(\mathbf{p}) = \mathbf{sim}\left(\mathbf{f}\left((\mathbf{x} \mid \theta\mathbf{p}), \mathbf{f}(\mathbf{x} \mid \theta\mathbf{f})\right)\right). \tag{5}$$

```
1 Data: R;
2 Result: Sorted predicates on incremented value;
3 for each predicate p in R do
4 |  Failure (p) = Calculate Failure (p);
5 |  Context (p) = Calculate Context (p);
6 |  Increase (p) = Failure (p) −Context (p);
7 end
```

Algorithm 1. Algorithm describing pseudo cod for Cause Isolation algorithm.

```
1 Data: predicates record R;
2 Result: Sorted predicates descending based on S value;
3 for each predicate p in R do
4 |  Bias (p | passed) = Calculate Bias (p) from passed cases;
5- |  Bias (p | failed) = Calculate Bias (p) from failed cases;
6 |  L (p) = similarityFun (Bias (p | passed), Bias (p | failed))
7 |  S (p) = -Log (L (p))
7 end
```

Algorithm 2. Algorithm describing pseudo cod for SOBER algorithm.

where **sim** = similarity function. Then, applying any one to one decreasing function on **L**(p) could define the ranking score of predicate p, **S**(p). Choosing the minus logarithm function for example, we can get the following equation for a ranking score:

$$\mathbf{S}(p) = -\log(\mathbf{L}(p)) \tag{6}$$

4. Experimental Instrumentation

In this section, we will discuss the experimental setup to study the effectiveness in terms of fault localization using the two selected statistical debugging algorithms on programs written using a dynamic programming language (Python). To the best of our knowledge, most of previous studies of statistical debugging techniques are performed on static programming languages, mainly C++ and Java. Some studies were performed on dynamic programming language (Ruby).

4.1. Experimental Environment

All the experiment steps were performed on a 2.20 GHz Intel core i7 MQ CPU with 8 GB physical memory running Microsoft windows 7 professional service pack 1. We implemented SOBER and Cause Isolation algorithms using Matlab R2009a. Python version that was used in writing tested programs was 2.7.10. We recorded only two types of predicates in order to monitor the programs behavior, branches and returns which were described in the previous section. Scalar pairs predicates were dropped, this dropping reduced the overhead by half without affect the localization effectiveness significantly [8].

4.2. Programs under Study

In order to evaluate statistical debugging algorithms on programs coded in Python, we implement five programs to be tested. These programs vary in their complexity and structure. **Table 1** shows the programs with their Source Lines of Code (SLOC) in Python and cyclometic complexity, which is a quantitative measure of the number of linearly independent paths through a program's source code. As shown in **Table 1**, programs could be classified into three levels according to their complexity and structure:

- Level one: This level contains simple and short programs. Programs are based on basic branching control, they neither contain loops nor function return statements.
- Level two: a program in this level has more SLOC than programs in first level, more cyclometic complexity. It is based on branching control and looping structure.

Table 1. The details of tested programs.

Program	Level	Program characteristics		
		Source lines of code (SLOC)	Cyclometic complexity	Number of branches
Median calculator	Level 1	41	4	7
Triangle type	Level 1	34	8	9
Product table generator	Level 2	82	12	20
Polynomial evaluator	Level 3	119	16	34
Linear predictive coding calculator	Level 3	570	62	129

- Level three: Programs in this level have most cyclometic complexity and most SLOC, they contains branching, looping, functions calls and returns, exceptions handling.

In our experiment, for each program we implemented two versions, one is correct while the other is manually injected by bug. For each program, we instrument, manually, a set of predicates as mentioned in the previous section. Also we generate a set of test cases for each program randomly.

4.3. Experimental Procedures

Our experiment, as shown in **Figure 1**, goes through five main steps:

Step 1: Subject programs are coded in Python programming language. As mentioned earlier, for each program both correct version and bug injected version are implemented, where the injected bugs lead to unexpected outputs.

Step 2: Add predicates to the source code in order to monitor the behavior of programs during the execution.

Step 3: Includes running the programs on generated test cases. Test cases are generated automatically for each program such that they cover all aspects of the program. Test cases are generated randomly in order to avoid test guidance. During the execution, for each test case, predicates are recorded with corresponding to passing or failing of the test case. Predicates records are fetched and reformatted in the fourth step.

Step 4: Predicates are ranked using SOBER and Cause Isolation algorithms. The aim of ranking is to determine predicates that are most relevant to the bugs. The input of SOBER and Cause Isolation algorithms is the predicates records, while the output is predicate ranking corresponding to their bug relevancy.

5. Results and Discussion

In this section, we will discuss the experimental results for each tested program in details. Then we will discuss the limitation of SOBER algorithms.

5.1. Median Calculator Program

It is a simple program, belongs to level one. The function of this program is to find the median of three numbers. The input is a list of three numbers and the output is their median value. Five predicates were instrumented in the source code of this program at variant branching points. Predicates values are recorded during the execution in order to rank them later. Ten test suites are generated for this program, each one contains 1000 random test cases, each test case contains a list of three numbers between (−1000, 1000) in addition to the expected output. Three bugs injected versions were tested; one bug injected, two bugs injected and three bugs injected. In single bug version, there were 166 failed test cases and 834 passed test cases. Both SOBER and Cause Isolation algorithms worked perfectly. Both of them rank the bug related predicate $p4$ as the most bug relevant. $p4$ was at the top of predicates in ranking. **Table 2** and **Table 3** summarize the results of predicates ranking for two other buggy versions. Ranking process was performed 10 times, given different predicates records recorded from different executions on 10 test suites. Results of all ranking processes are averaged in the tables.

In the two buggy version, predicates $p4$ and $p5$ are directly related to injected bugs. Failed test cases was 339, while passed test cases was 661 in average. As shown in **Table 2**, SBOER ranks $p1$ as most bug relevant predicate, although there is no failed test cases when $p1$ is true, the cause of this missing is that both $p4$ and $p5$ are observed only when $p1$ is false. SOBER measuring the difference between true evaluations and false evaluations

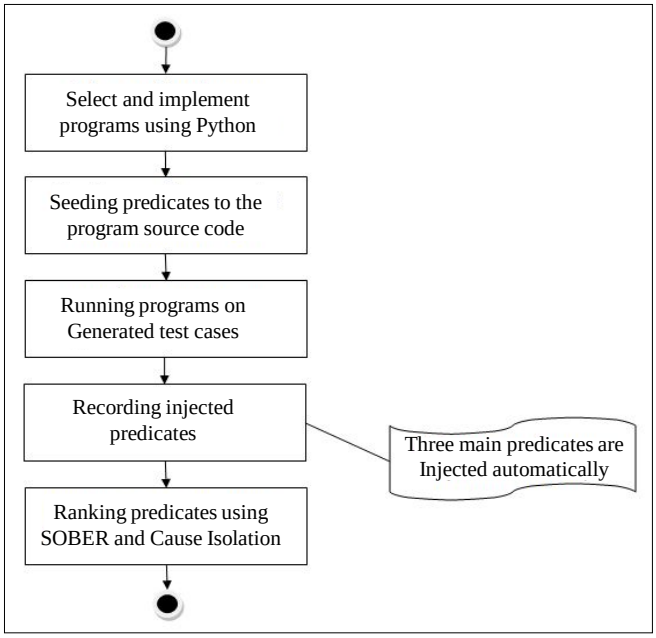

Figure 1. Figure showing the flow chart of presented method.

Table 2. Median calculator predicates ranking results of two bugs injected version.

Predicate	Average number of failed test case caused out of 1000	Rank out of 5	
		SOBER	Cause isolation
Predicate 1 ($p1$)	0	1	5
Predicate 2 ($p2$)	0	4	3
Predicate 3 ($p3$)	0	5	4
Predicate 4 ($p4$)	174	3	2
Predicate 5 ($p5$)	165	2	1

Table 3. Median calculator predicates ranking results of three bugs injected version.

Predicate	Average number of failed test case caused out of 1000	Rank out of 5	
		SOBER	Cause isolation
Predicate 1 ($p1$)	0	3.2	4
Predicate 2 ($p2$)	0	5	5
Predicate 3 ($p3$)	167.5	1.6	1.4
Predicate 4 ($p4$)	160.5	3.8	3
Predicate 5 ($p5$)	170	1.4	1.8

of $p1$, so it will rank it as bug related. Meanwhile, SOBER ranked the bug related predicates $p4$ and $p5$ as 3 and 2 respectively, which is almost good ranking. Cause Isolation ranked $p4$ and $p5$ as 2 and 1 respectively, which is the correct result.

In the three buggy version, predicates 3, 4 and 5 are directly related to injected bugs, while predicates 1 and 2 are bug irrelevant. Passed test cases were 502, while failed test cases were 498 in average. As shown in **Table 3**, considering number of failed test cases caused, we can sort predicates as ($p5$, $p3$, $p4$, $p1$ and $p2$), we can say that $p5$ is the most important or most bug relevant predicate. SOBER algorithm ranks the predicates as ($p5$, $p3$, $p4$, $p1$ and $p2$). It successfully ranked $p5$, $p3$ and $p2$ while failed to rank $p1$, $p4$ correctly. The reason is that $p1$ and $p4$ are nested predicates in the program, such that whenever $p4$ evaluated as true then $p1$ is true too while the

opposite is incorrect.

So, since SOBER ranks predicates based on the difference between passing and failing for predicate in both true and false evaluation, then it will measure a difference between the evaluations of $p1$ in both failed and passed test cases. Therefore it ranks $p1$ as bug related (3^{rd} out of 5). On other hand, Cause Isolation algorithms ranks the three bugs related predicates correctly to be at the top of predicates, but it failed to sort them based on importance. It ranked $p3$ as the most important predicate although it caused failures less than $p5$, this because it did not consider the test cases where predicates evaluated to false. In this part of experiment, all predicates in the program are observed, which reflects the coverage ratio of test cases, this means that test cases cover all aspects of the program.

5.2. Triangle's Type Program

Triangle's type program is another simple program, which belongs to level one. The function of this program is to determine the type of triangle. The input is the length of three sides, the program should specify if these sides can form a triangle or not, if yes, it should return its type (equilateral, isosceles or scalene). Four predicates were instrumented in the source code of this program at variant branching points. Predicates values are recorded during the execution in order to rank them later. Ten test suites are generated for this program, each one contains 1000 test cases generated randomly, each test case contains a list of three numbers between (−2, 25) in order to cover all input cases (negative, zero and positive scalars), in addition to the expected output. Two bugs were injected in the source code of a program. One of the bugs was injected at the condition of fourth predicate $p4$, *i.e.* the condition itself was wrong. The other bug was injected at the branch of first predicate $p1$. Passed test cases were 684, while failed test cases were 316. Recorded predicates were fetched and ranked by SOBER and Cause Isolation algorithms. Ranking results are shown in **Table 4**. As shown in **Table 4**, SOBER algorithm failed to rank $p1$ as a bug related predicate, while it successfully ranked $p4$ as a top predicate. This is an advantage of SOBER algorithm over Cause Isolation which failed to detect this type of bugs, where the bug injected at the predicate condition itself. On other hand, Cause Isolation ranked $p1$ perfectly as a most bug relevant predicate.

5.3. Generate Product Table Program

It is a program with medium complexity, belongs to level two. The input of this program is a positive integer number N, it should generate and print an n∗n matrix contains a product table of N. In such type of programs, a predicate could observe more than one time during a single run if the predicate instrumented within a loop structure. Nine predicates were instrumented in the source code of program at variant branching points. Predicates values are recorded during the execution in order to rank them later. Ten test suites are generated for this program, each one contains 1000 test cases generated randomly, and each test case contains a random integer, in addition to the expected output. Only one bug was injected in the source code of the program. It is injected at a branch point inside a loop structure. The average of passed test cases was 682, while the average of failed test cases was 318. The most bug relevant predicate was $p9$. The result of ranking predicates shows that SOBER failed to rank $p9$ as a bug relevant, it ranked it to be at position 5 in best case. On the other hand, Cause Isolation ranked $p9$ perfectly as a most bug relevant predicate. In our estimation, the cause of this result was the rarity of bug occurrences, so another predicates evaluations affected the ranking process of SOBER since it evaluates both true and false predicates. Since Cause Isolation evaluates only true predicates, it did not affected by the rarity of bug occurrences.

5.4. Polynomial Evaluator Program

It is one of the most complex program in our suite, belongs to level 3. It is based on control structure, looping structure and functions return. This program have two inputs, first one is a scalar and second on is a list of coefficients. Program should do the following steps: Validating the inputs to ensure that they are all numeric; otherwise program will return an invalid message. Generating a string representation of a polynomial function **f**(x) based on the input coefficients. Finally, evaluating the input scalar (a) based on the polynomial **f**(x), *i.e.* calculating **f**(a). The program consists of four methods, validate number, validate coefficients, generate polynomial string and evaluate polynomial scalar. Nineteen predicates were instrumented in the source code of program at variant branching points. Predicates values are recorded during the execution in order to rank them later. Ten

Table 4. Triangle type predicates ranking results.

Predicate	Average number of failed test case caused out of 1000	Rank out of 4	
		SOBER	Cause isolation
Predicate 1 ($p1$)	282	3	1
Predicate 2 ($p2$)	0	4	2
Predicate 3 ($p3$)	0	2	4
Predicate 4 ($p4$)	34	1	3

test suites are generated for this program, each one contains 1000 test cases generated randomly, and each test case contains a list of coefficients and a scalar, in addition to the expected output. Two bugs were injected in the source code of the program. It is injected at a branch point inside a loop structure. The average of passed test cases was 752, while the average of failed test cases was 248. The directly bug relevant predicates was $p8$ and $p19$. Recorded predicates were fetched and ranked by SOBER and Cause Isolation algorithms. Ranking results of top 5 predicates are shown in **Table 5**.

As shown in **Table 5**, Cause Isolation ranked $p8$ and $p19$ correctly as a top two predicates, but it failed to order them based on importance, it ranks $p19$ as top one although it caused bugs less than $p8$. SOBER algorithm ranked predicates $p6$, $p7$, $p8$ and $p9$ as a top predicates. Actually, their ranking results were very close to each other, sometimes they were equivalent. This because that the predicate $p8$ (the cause of bug) is a scalar return predicate indicates that a scalar $x < 0$, while $p7$ is the opposite predicate, it indicates that a scalar $x \geq 0$. So, whenever $p8$ is true, $p7$ will be false and vice versa. This makes a big difference between $\mathbf{f}(X|\theta p)$ and $\mathbf{f}(X|\theta f)$ for both $p8$ and $p7$. Same issue applied on $p6$ and $p9$, where $p6$ was $x > 0$ and $p9$ was $x \leq 0$.

5.5. Linear Predictive Coding (LPC) Coefficients Calculator

LPC is a speech coding and compression algorithm. It is the idea of extracting a set of coefficients from a speech frame. These coefficients can be used to regenerate the original speech frame [16]. Our LPC coefficients calculator program has only one input, the sound frame. The program consists of 17 functions. The output of the program should be the LPC coefficients extracted from the input speech frame. We assumed that frame length is 320 samples, LPC coefficients count is 13. We instrumented 257 predicates in the source code of program at variant branching points. Predicates values are recorded during the execution in order to rank them later. Ten test suites are generated for this program, each one contains 1000 test cases generated randomly, and each test case contains a randomly generated speech samples, in addition to the expected output. Two bugs were injected in the source code of the program. It is injected at a branch point inside a loop structure. The average of passed test cases was 830, while the average of failed test cases was 170. First bug was directly related with two predicates, $p247$ and $p249$. Second bug was directly related with four predicates, $p109$, $p244$, $p246$ and $p250$. Ranking results of these predicates are shown in **Table 6**.

As result shows, both SOBER and Cause Isolation did not rank the bug related predicates perfectly, this because the huge number of predicates and the program complexity. However, even though the result is not perfect but it still intelligible. The bug related predicates were ranked at the top 7% predicates. In real world, such result will significantly reduce the bug localization process resources.

5.6. Limitations of SOBER and Cause Isolation Algorithms

Based on the experiment results shown in Section 5, we can list the limitations of SOBER algorithm as follow:

- Nested predicates: When two predicates were nested, and the inner on is directly bug related, then SOBER may rank the outer predicate to be the most bug related predicate.
- Rarely observed predicates: When the most bug related predicate is rare to observe, then ranking process of SOBER may affected by the evaluations of another predicates.
- Complement predicates: when two predicates were instrumented where one of them is the complement of the other, then both of them will get the same rank which may confuse the software debugger.

Similarly, we can list the main limitations of the Cause Isolation algorithm as follow:

- Sorting the top predicates: Although the algorithm ranks top predicates successfully, it fails to sort them based on the importance.

Table 5. Polynomial evaluator predicates ranking results.

Predicate	Average number of failed test case caused out of 1000	Rank out of 19	
		SOBER	Cause Isolation
Predicate 6 (*p*6)	0	3	12
Predicate 7 (*p*7)	0	1	13
Predicate 8 (*p*8)	194	2	2
Predicate 9 (*p*9)	0	4	3
Predicate 19 (*p*19)	54	5	1

Table 6. LPC calculator predicates ranking results.

Predicate	Average number of failed test case caused out of 1000	Rank out of 257	
		SOBER	Cause Isolation
Predicate 109 (*p*109)	7	8	2
Predicate 244 (*p*244)	19	17	8
Predicate 246 (*p*246)	32	10	12
Predicate 247 (*p*247)	28	15	15
Predicate 249 (*p*249)	46	16	6
Predicate 250 (*p*250)	38	4	18

- Bugs in predicates: When the bug is at the condition of predicate itself, the algorithm fails to rank the predicate as bug related predicate.

6. Studied Algorithms Comparison

In this section we conduct an effectiveness comparison between SOBER and Cause Isolation algorithms when they applied on Python programs. This comparison is derived from the experiments that explained above.

6.1. Effectiveness Metrics

Bug localization quality can be measured using T-Score. This measure was originally proposed by Renieris *et al*. [17], and was later adopted by Cleve H. and Zeller A. [11]. We briefly summarize this measure as the following:

- Given a (buggy) program **P**, the program dependency graph is written as **G**, where each statement is a node and there is an edge between two nodes if two statements have data and/or control dependencies.
- The buggy statements are marked as defect nodes. The set of defect nodes is written as **Vdefect**.
- Given a bug localization report **R**, which is a set of suspicious statements, their corresponding nodes are called blamed nodes. The set of blamed nodes is written as **Vblamed**.

A programmer can start from **Vblamed** and perform the breadth-first search until he reaches one of the defect nodes. The set of statements covered by the breadth first search is written as **Vexamined**. The T-score, calculated using Equation (7), measures the percentage of code that has been examined in order to reach the bug,

$$\mathbf{V} = \frac{|\mathbf{Vexamined}|}{|V|} \times 100\% . \tag{7}$$

where |**V**| is the size of the program dependence graph. T-score roughly measures the real cost in locating a bug. Whenever the code to be examined is less, the quality of a predicates ranking algorithm is high. A good algorithm should generate a high quality predicates ranking requiring minimal code checking.

6.2. Comparison Result

In this comparison we used the set of Python programs described Section 4. Corresponding only one buggy version for each program, the total number of bug related predicates is 13. The maximum percentage of examined

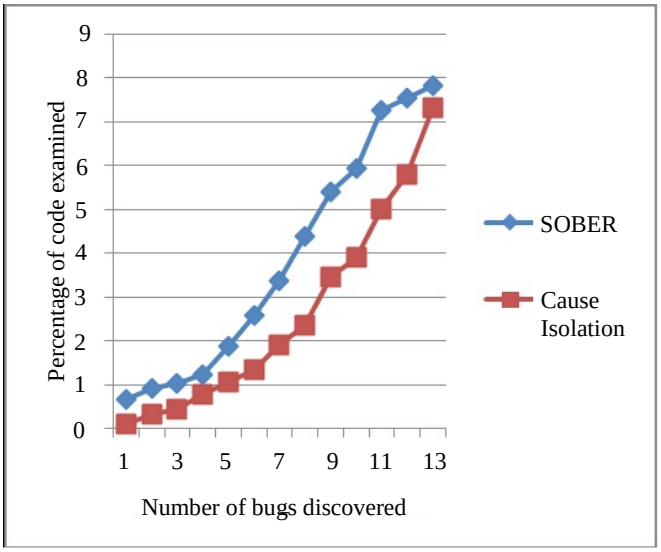

Figure 2. Figure showing a comparison of studied algorithms.

code required to discover all bug related predicate was less than 8%.The result of the comparison between SOBER and Cause Isolation algorithms is shown in **Figure 2**. As shown in the figure, SOBER algorithm required more code examination than Cause Isolation when applied on a set of Python programs. For example, to discover 9 bugs using Cause Isolation algorithm, we need to examine about 3.4% of the code, while we need to examine 5.3% of the code when using SOBER algorithm.

7. Conclusion and Future Works

In this paper, two Statistical Debugging SOBER and Cause Isolation are implemented and compared in terms of fault localization on five python programs. The experiments showed that in programs that contain a single bug, both of SOBER and Cause Isolation almost rank the bug related predicate correctly. In programs that contain multi bugs, SOBER has limitations related to nested predicates, rare predicates and complement predicates. Cause Isolation has limitations related to sorting predicates based on importance and detecting bugs in predicate condition. Both of SOBER and Cause Isolation algorithm have a limitation related to the programs size. Their rank accuracy is decreased when using in a big programs, but they are still reasonable. We also compared the quality of both algorithms. The comparison showed that when applied on a set of Python programs, SOBER algorithm requires more code examination than Cause Isolation to discover the bugs.

Experiments shown on this research are applied on relatively simple Python programs. As a future work, more complex open source programs have to be experimented in order to achieve better understanding of algorithms' limitations. Addition comparisons can be performed to compare statistical algorithms with other non-statistical algorithms. Applying statistical debugging algorithms on other dynamic programming languages can lead to generalize the results to include dynamic programming languages category. Some modifications on statistical debugging algorithms could be added to overcome the limitations listed above, for example a new predicate instrumentation approach could be proposed in order to overcome the SOBER algorithm limitations.

References

[1] Sebesta, R.W. and Mukherjee, S. (2002) Concepts of Programming Languages. Vol. 281, Addison-Wesley, Reading.

[2] Scott, M.L. (2000) Programming Language Pragmatics. Morgan Kaufmann.

[3] Saeed, F.S.M. (2008) Systematic Review of Verification & Validation in Dynamic Languages. MS. Thesis, Blekinge Institute of Technology, Sweden.

[4] Ousterhout, J.K. (1998) Scripting: Higher Level Programming for the 21st Century. *Computer*, **31**, 23-30. http://dx.doi.org/10.1109/2.660187

[5] Tratt, L. and Wuyts, R. (2007) Guest Editors' Introduction: Dynamically Typed Languages. *IEEE Software*, **24**, 28-30.

http://dblp.uni-trier.de/db/journals/software/software24.html#TrattW07
http://dx.doi.org/10.1109/MS.2007.140

[6] Yu, Y., Jones, J.A. and Harrold, M.J. (2008) An Empirical Study of the Effects of Test-Suite Reduction on Fault Localization. *Proceedings of the* 30*th International Conference on Software Engineering*, 201-210. http://doi.acm.org/10.1145/1368088.1368116

[7] Jiang, L. and Su, Z. (2007) Context-Aware Statistical Debugging: From Bug Predictors to Faulty Control Flow Paths. *Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering*, 184-193. http://dx.doi.org/10.1145/1321631.1321660

[8] Liu, C., Yan, X., Fei, L., Han, J. and Midkiff, S.P. (2005) SOBER: Statistical Model-Based Bug Localization. *ACM SIGSOFT Software Engineering Notes*, **30**, 286-295. http://dx.doi.org/10.1145/1081706.1081753

[9] Andrzejewski, D., Mulhern, A., Liblit, B. and Zhu, X. (2007) Statistical Debugging Using Latent Topic Models. *European Conference on Machine Learning*, 6-17. http://dx.doi.org/10.1007/978-3-540-74958-5_5

[10] Jones, J.A., Harrold, M.J. and Stasko, J. (2002) Visualization of Test Information to Assist Fault Localization. *Proceedings of the* 24*th International Conference on Software Engineering*, 467-477. http://dx.doi.org/10.1145/581396.581397

[11] Cleve, H. and Zeller, A. (2005) Locating Causes of Program Failures. *Proceedings of the* 27*th International Conference on Software Engineering*, Saint Louis, 15-21 May 2005, 342-351. http://dx.doi.org/10.1109/icse.2005.1553577

[12] Liblit, B., Naik, M., Zheng, A.X., Aiken, A. and Jordan, M.I. (2005) Scalable Statistical Bug Isoltion. *ACM SIGPLAN Notices*, **40**, 15-26. http://dx.doi.org/10.1145/1064978.1065014

[13] Liblit, B. (2007) Cooperative Bug Isolation: Winning Thesis of the 2005 ACM Doctoral Dissertation Competition (Vol. 4440). Springer.

[14] Liu, C., Lian, Z. and Han, J. (2006) How Bayesians Debug. *Sixth International Conference on Data Mining*, Hong Kong, 18-22 December 2006, 382-393. http://dx.doi.org/10.1109/ICDM.2006.83

[15] Akhter, A. and Azhar, H. (2010) Statistical Debugging of Programs Written in Dynamic Programming Language: RUBY.

[16] Chu, W.C. (2004) Speech Coding Algorithms: Foundation and Evolution of Standardized Coders. John Wiley & Sons.

[17] Renieres, M. and Reiss, S.P. (2003) Fault Localization with Nearest Neighbor Queries. 18*th IEEE International Conference on Automated Software Engineering*, 6-10 October 2003, 30-39. http://dx.doi.org/10.1109/ASE.2003.1240292