Applying Spectrum-based Fault Localization on Novice's Programs

Eliane Araujo, Matheus Gaudencio, Dalton Serey, Jorge Figueiredo
Department of Computer Science
Federal University of Campina Grande
Campina Grande, Brasil
{eliane, matheusgr, dalton, abrantes}@computacao.ufcg.edu.br

***Abstract*—Most introductory programming courses count on automated assessment systems (AAS) to support practical programming assignments and give fast feedback. AAS usually rely on tests results to check the program's functional correctness to provide feedback to students. Novice programmers, however, may find it difficult to map such feedback to the root failures' cause in their programs. It can be even more frustrating when the code is "almost right". In this paper we investigated the use of a fault localization technique on programs produced by students of introductory programming. Our proposed approach is grounded on spectrum-based fault localization (SBFL). The results of our empirical study showed that this lightweight technique is promising. It can be easily adapted to different AAS to generate useful feedback not only to students but also to instructors. We also delineate the scope where SBFL use is jeopardized. The main contribution of this paper is to present the benefits and drawbacks of applying SBFL, in the context of programming learning, as a novel source of information about students' programming assignments faults.**

Keywords— programming, automated assessment; software fault diagnosis; novices, experimentation.

I. INTRODUCTION

Nowadays, many programming courses are supported by automated assessment systems (AAS) that provide feedback to the students and also collect data about their interaction with the instructional material. However, the feedback provided by those systems about the students' difficulties in programming assignments are distant from the instructors' enriched feedback. The problem is that AAS may not provide adequate feedback in some phases of the programming process, so that students may feel frustrated and face difficulties to proceed autonomously on their learning pathway.

AAS usually rely on tests results to check the program's functional correctness to provide feedback to students. Novice programmers, however, may find it difficult to map errors in their code with failing test cases [1]. It can be even more frustrating when the code is "almost right". Sometimes, even if the student knows how to solve the proposed problem, she may fail in producing a functionally correct implementation. The failure revealed by tests can be caused by a wrong operator ("greater than" instead of "greater than or equal to"), a wrong value on the "if" conditional statement or even a misplaced parenthesis. An adequate feedback in this situation would help and stimulate the student to solve the problem and move on. In fact, there are on the literature different strategies proposed to find bugs on student code [1][2]. However, they may not be easily adopted by whichever programming course, as they increase instructors' duties requiring the production of new artifacts.

This paper investigates the use of a lightweight fault localization technique on programs produced by students of introductory programming. Spectrum-based fault localization (SBFL) has been used successfully in different areas of software development [3][4][5]. This technique relies on program spectra: program traces that reveal which parts of the code are active during a failed or successful execution. SBFL predicts the likelihood of a software component, for example, to be responsible for faulty executions. This research focuses on programming assignments proposed along with a test cases suite that are automatically executed by an AAS. In this sense, those systems would compute SBFL measures at a low cost. It does not demand artifacts different from those instructors are used to provide.

We conducted an empirical study to investigate the suitability of using SBFL on novice's programs as a novel source of information aimed to AAS feedback generation. We collected data from an entire edition of an introductory programming course comprising more than 10,000 Python programs, referring to almost 300 programming assignments, from approximately 100 students. We analyzed the tests results of each program submission to characterize them. We observed that 25.9% of the submissions in the data set were considered incorrect, as they did not pass the complete set of tests. In order to be adequate to SBFL use, the submission has to pass at least one test. In this sense, 61.6% of incorrect submissions are initially adequate to SBFL application. A broader exploratory study was able to characterize these programs and provides a more comprehensive knowledge of the extent of situations where the technique could be relevant.

Then, we performed a quantitative study with 5 programming assignments to assess the quality of SBFL diagnostics. We used as baseline instructor's assessments and annotations on the programs. On average, using SBFL, it is necessary to look in only ~20% of the program's lines of code to find the flaw. This study also corroborated with the previous findings on literature. We discuss situations where SBFL was inappropriate to provide feedback about the programs' faults.

The contributions of this work, addressed to instructors and AAS developers, are the following:

- We present and adapt SBFL as lightweight alternative to find faults in students programs. It is a new source of information for feedback generation to instructors or students. Instructors or AAS developers must be responsible for modulate the information before deliver it to students, so it could make better sense in pedagogical context.
- We discuss the use limitations of this technique towards introductory programming assignments, in particular Python procedural programs, as lessons learned from an exploratory study.
- We report a case study evaluation, on real programming assignments, highlighting good results in terms of diagnostic accuracy.
- We summarize strategies on how to maximize SBFL use in programming learning context and propose them as future works.

II. Related Works

Automated assessment systems are used for decades in programming learning context [6]. In general, AAS employ comparable approaches and provide similar features [7]. The most common feature assessed by them is code functional correctness. A typical system executes a set of test cases, provided by the instructors, and compares the expected output to the observed output produced by students' programs. Some systems, characterized as grading systems [8], use those results to grade the programming assignment. Grading systems may weigh another factors, besides correctness, such as deadline penalty, resubmission times, type of errors, test coverage [9][10], etc. AAS may also provide features such as quality assessment, in terms of: efficiency [11], static software metrics [12] and programming style [13]. The work hereby described, focuses on fault localization [1][2] and code repair strategies [1], which are discussed in more details in the following subsection.

A. Fault localization and repair

The approach adopted in [2], to localize bugs in student code and provide feedback, is based on the automatic generation of program execution traces. An execution trace is a list of each program execution step, line by line, and the value of the variables at each time. By reading these traces, students can understand their program execution path and how it has evolved to reach the end. In order to generate feedback to students, the authors suggested comparing students' trace to the one generated from the instructors' reference solution. This works resembles the approach here presented, as it is also based on execution traces. However, SBFL can go further as it can map faults to software elements. The code is an artifact students are used to deal with, differently from an execution trace.

In another way, Singh and colleagues' work tries to identify the error in the students' code and guides them to it correction [1]. The authors argue that most of students' errors in programming assignments are predictable as students who are solving the problems were exposed to the same classes and learning materials. For these and other reasons, their errors tend to follow a typical pattern. They generate feedback based on possible fixes to error models that are typically found in particular programming assignments. Their approach could provide detailed information about the error localization and how to solve it. It also allows the message customization according to the level of feedback the teachers want students to see.

However, to use this approach instructors must provide, in addition to the assignment's reference solution, the model of typical mistakes that could be made by students in that assignment. Errors must be described in an Error Model Language - EML proposed by the authors. This approach has been successfully evaluated in MIT online and regular introductory programming courses. We argue that the overhead required to use Singh and colleagues' proposal is higher than to use our approach. We speculate that having the instructors to foresee every error possibility and also learn a new language to model them is a big hurdle to impose. SBFL is simple and easily adaptable to existing AAS, as it does not require additional artifacts besides the test cases already provided by instructors. In contrast, the precision level of faults localization in our approach is lower than the observed with Singh's approach.

In a very recent work, Edmison and Edwards evaluated the use of SBFL on object-oriented programming learning context [14]. They recognized it as a "feasible strategy" to provide feedback on where to look for faults on programs. Differently from our work, addressed to novice programmers, the authors focus CS2 students, which are not complete beginners as they are taken their second or third programming course. Furthermore, the work deals with objected-oriented Java programs with the aim to locate and identify what methods are most likely to contain the fault. Our proposal has a finer granularity as it ranks the lines where the fault could be found in a procedural Python program. In addition, our research goes a step further as it discusses when not to apply SBFL. As a result of an exploratory study, performed in a dataset from over 10,000 programs submissions, we characterized the students' solutions and discussed the scope of the technique: when and why it is useful. The present work considers practical significance of the results as it gives insights into how to make better use of the SBFL in programming learning context.

III. Background

In this section we describe the key concepts related to Spectrum-based Fault Localization (SBFL) technique and how we have adapted it to introductory programming learning context. Introductory programming assignments are usually well-formed specifications of problems to produce relatively small programs. These programs receive inputs and transform them in testable outputs. In this setting, *faults* can be seen as bugs in the programs and *failures* are evidenced by unexpected outputs for a given input [3].

A. Spectrum-based Fault Localization

SBFL is a technique that dynamically analyses a program in order to calculate the likelihood of a given *component* to be faulty. For diagnosis purpose, the concept *component* stands for an element of the system considered to be atomic. In multiple application of SBFL, *components* can be mapped to different targets: blocks of code when analyzing industry software systems [3]; cells in case of spreadsheets analysis [4]; agents when examining multi-agent systems [5] and methods in the study of object-oriented student programs [14].

The idea is to observe multiple runs of the program, where components are exercised in failed and passed executions and calculate how a component is "suspicious" to be faulty. Failure detection is a precondition to fault localization: it is necessary to recognize that something is wrong before trying to locate the fault [3]. In this scenario, we use test cases provided by instructors to each programming assignment. However, seeking for failures through test cases are an elementary way of detecting faults. Some of them may not be disclosed if the set of test cases were not complete. Provided we cannot guarantee this completion, we assume, in this study, that all program's faults are revealed by test cases. In this sense, a failed run occurs when an error is detected – the expected output is different from the observed. On the other hand, a passed run occurs when the output is equal to the expected.

The data collected from failed/pass runs are used to compose a hit-spectra matrix, see Fig. 1. This is an NxM matrix; where N represents the number of *components* inspected in the program and M the number of runs (test executions, for example). Each a_{ij} element of the matrix corresponds to a binary value: (1) if it was hit in that particular run and (0) in the contrary [4]. In practice, this means that we aim to identify which component is "involved" in a failure. Another necessary element used to calculate components' suspicious in SBFL is the error vector. This N-length binary vector holds the information about "fail" and "pass" to N runs, see Figure 1.

$$\begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & \dots & \dots & a_{2n} \\ \dots & \dots & \dots & \dots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix} \begin{bmatrix} e_1 \\ e_2 \\ \dots \\ e_n \end{bmatrix}$$

$$\mathbf{s_1} \quad \mathbf{s_2} \quad \dots \quad \mathbf{s_n}$$

Fig. 1. MxN hit-spectra matrix and N-length error vector. Each column represents a *component* spectrum

After computing the hit-matrix and error vector, the next step is to identify which column in the matrix resembles most the error vector. Similarity coefficients, which are largely known in the literature, are used in this activity. Passos and colleagues cited in their work the use of more than 40 heuristics to compute similarity between vectors [5]. The idea is simple: the more a *spectrum* of a given component is similar to the error vector, the more it is suspicious to be the cause of the detected error.

In this work, we used Jaccard similarity coefficients in order to calculate the value of "suspiciousness index" for a given component. Refer to [3] in order to obtain more information about how to compute those coefficients in SBFL context.

Finally, the coefficient values assigned for each component are ranked in descending order (most similar figures on top most positions). It means that, in order to find the fault in a given code, it is recommended to inspect the components following the SBFL ranking order. It can be noticed that the accuracy of this technique diagnosis is limited: it is a recommendation not a prescription. However, SBFL merit is to greatly reduce the range of code inspection. In the work reported by [3], it exonerated, on average, 80% of the blocks of code of being faulty.

B. Students' Programs Fault Localization with SBFL

We argue that using SBFL to generate information to provide automatic feedback in the context of programming learning is relatively simple. AAS are increasingly been used in programming courses. They can be used to calculate SBFL coefficients and compose the rank, as they already count on a set of tests provided by instructors.

In this paper, we applied SBFL to Python programming assignments of an introductory programming course. Each *component* of the technique is mapped to one line of the program, excluding comments or blank lines. We rely on the set of tests provided by the instructors in order to thoroughly test the code. In this sense, the diagnostic accuracy of the strategy also relies on instructors test quality.

Fig. 2 presents an example of real student code. This assignment specification asks students to write a program to calculate the body mass index of males and females. The value of suspiciousness index *s,* for each line, is showed on the left side. It can be observed that the last two lines (7, 8) obtained the highest values of *s*. The faulty line of code is indeed the last line (8), as the variable used should have been *bmi_female* instead of *bmi_male*.

```
0.5 | 1. genre = raw_input()
0.5 | 2. height = float(raw_input())
0.5 | 3. bmi_male = 72.7 * height - 58
0.5 | 4. bmi_female = 62.1 * height - 44.7
0.5 | 5. if genre == "m" or genre == "M":
0.0 | 6.      print "%.03f" % bmi_male
1.0 | 7. elif genre == "f" or genre == "F":
1.0 | 8.      print "%.03f" % bmi_male
```

Fig. 2. Sample of student code is on the right. Values of suspiciousness index of each line are on the left.

IV. Research Methodology

In this section, we are going to present the research methodology applied in the empirical study of this work. In order to investigate the applicability of SBFL on novice programs, we followed a two-pronged approach: an exploratory study and a case study. The first aimed at having a broader look on students' code production, qualitatively evaluating their errors and evaluating the soundness of SBFL

in the context of programming learning. The latter intended to quantitatively evaluate the approach, measuring its accuracy and other metrics, in a given set of real students' programs sample.

A. Data Collection

This research was performed in an introductory programming course with undergraduate students of Computer Science. In this course students learn programing skills using Python language. Students learn how to use expressions, alternative statements, collections, strings, collection-controlled loops, conditional-controlled loops and functions. Their laboratory activities focus on solving problems by coding programs and submitting them to an AAS to be automatically tested. Each programming assignment presents a basic input and output test. Students are used to test their programs with this basic test before submitting their solution to the system. After the code submission to the AAS, additional hidden tests are executed. As a result, students receive the number of tests failed and passed for that code. They are not penalized for multiple submissions for the same question.

In the exploratory study, we collected students' submissions for programming assignments of a complete course edition. They were collected using an instrumented AAS, which was developed in-house and tailored for our course. The dataset of the exploratory study comprises 10,357 programs, referring to 277 programming assignments, from 115 students. On average, each student submitted 90 programs along the course. Some of them were selected to a more in-depth qualitative analysis.

In the case study, we selected a set of 5 assignments focusing on different programming learning outcomes expected in the course, such as conditional structures (if), iterations (for and while), simple algorithms with data structures (lists). We seek for assignments that students faced difficulties to succeed. For this reason, of 181 submissions that composes this study dataset, 53.6% failed in at least one test case. It is worthy to note that each assignment has their own set of tests and they were also necessary to compute SBFL values.

B. Exploratory Study

In general, the purpose of an exploratory study is to answer research questions about the studied phenomena without formulating any previous hypothesis [15]. In this study we investigated the suitability of using SBFL in different configurations of defects observed on students' programs. From this point on, we are going to refer this "configurations of defects" as scenarios. Our main purpose is to define the scope of action of SBFL in programming learning context. For this reason, we want to answer the research questions:

RQ1) What are the preconditions to use SBFL in students' programs, according to their tests' results?

RQ2) SBFL performance depends on the programs' defects configuration?

RQ3) Which are the scenarios of defects configuration in which SBFL performance is good or is jeopardized?

SBFL is a technique that dynamically analysis a program in order to calculate the likelihood of a given *component* to be faulty. In this study, each logical line of the code (program lines excluding comments, headers and blank lines) is considered a *component*. It means that, the likelihood of being faulty is calculated for each line according to the technique algorithm. This value is referred as index *s*, for suspicious. It represents the similarity between the line *spectrum* and error vector (see Section III.A). We chose Jaccard heuristic to compute the similarity coefficient in this study because it is a simple strategy that yields good results on related work [5].

Initially, we inspected the dataset and determined which program submissions could be used as subject of this investigation. Such definitions of criteria helped us to answer the first research question. In sequence, we executed SBFL strategy to calculate *s* indexes for each program line and to create the rank. These are the steps of the process: Firstly, for each test of the test suite associated with that programming assignment, we generate an execution trace of the program. Each trace has a set of lines that represents the lines exercised during the test execution. Secondly, we compute the hit-matrix and the error vector (Fig. 1) for those executions. Then, we calculate the *s* value of each line using the Jaccard similarity coefficient S_J described in (1) with the values from Table I [3].

In Table I, C_{11} represents the number of failed tests that executed that line. C_{10} is the number of passed tests that hit such line. C_{01} is the number of failed tests that do not exercise the line and C_{00} is the number of passed tests that do not hit the line. All those values are calculated from the hit-matrix and error vector. Finally, we create a rank with the values of *s*. In theory, the lines more likely to be faulty are on the initial ranking positions.

$$S_J = \frac{C_{11}}{C_{11} + C_{10} + C_{01}} \quad (1)$$

TABLE I. DICHOTOMY TABLE REPRESENTING THE STATES OF A LINE

Test Result	**Line Hit**	
	Yes = 1	***No = 0***
Failed = 1	C_{11}	C_{01}
Passed = 0	C_{10}	C_{00}

The index *s*, calculated using Jaccard similarity heuristic, can range from 0 to 1. With this in mind, we perceived four possible situations when we observed the SBFL rank and compared with the real localization of defects in students' programs. Table II presents these scenarios.

In order to answer posed research questions, we searched the dataset and found programs that match each one of these situations. We performed a qualitative evaluation in such programs to better understand SBFL performance on those cases. Furthermore, there were other programs that did not fit in those categories and presented notable characteristics such as: multiple lines of errors, dead code and runtime errors. They also helped on the definitions of SBFL applicability scope.

TABLE II. SCENARIOS OF DEFECTS OBSERVATIONS IN REGARDS TO SBFL SUSPICIOUSNESS INDEX.

s	**Real Defects Found**	
	Yes = 1	***No = 0***
High (>=0.5)	S_1	S_2
Low (< 0.5)	S_3	S_4

C. Case Study

In the case study, we conjectured if SBFL could really help us to identify which are the lines responsible for the faults observed in students' programming assignments. So that, the values obtained through the technique could be used to generate feedback in the context of programming learning. In order to test it, we applied SBFL and used a set of evaluative metrics to analyze its results and practical significance. To conduct this study, we formulated the following research question:

RQ4) Is the quality of diagnosis delivered by SBFL good enough to generate useful feedback about faults localization in novice programming assignments?

In answering this research question, we applied SBFL technique to a set of programming assignments, as described on the previous section. This set of assignments was collected from mid-term exams of different course editions of Programming I. Experienced instructors manually assessed and annotated the programs highlighting its errors. We executed automatic tests and collected faulty program submissions. We also manually analyzed these submissions to make sure that the faulty lines were indeed identified. This inspection was work intensive, but it was fundamental to this study. Its results was used to compose the baseline of the study, an oracle of "true positives" faults, used to compute the evaluation metrics.

We instrumented the AAS to calculate *s* indexes and generate SBFL rank. The order in this rank indicates the likelihood of a line to be faulty. After applying the technique for each submission, we evaluated the success of the fault localization in contrast to the baseline using different metrics. Precision and recall are traditional metrics of information retrieval. They are used in this study with the following meaning:

- *Precision*: Measures the fraction of the "number of lines marked as faulty by SBFL, which are real faulty lines according to the baseline" by the "total number of lines indicated by SBFL".
- *Recall:* Measures the fraction of the "number of lines marked as faulty by SBFL, which are real faulty lines according to the baseline" by the "number of faulty lines according to the baseline".

In addition, we used another metrics proposed by Abreu and colleagues to evaluate SBFL diagnosis quality in terms of *accuracy* (*qd*) and *quality of the error detection* (*qe*) [3]. We are going to briefly describe the equations and the meaning of its compound values. Finer details about the underlying motivation can be found in [3].

Accuracy represents the quality of diagnosis of the technique in locating a faulty line along the program. It means the percentage of the program lines that does not need to be inspected when searching for a fault by traversing the ranking.

Let $d \in \{1, \ldots, N\}$ be the index of the faulty line. For all $j \in \{1, \ldots, N\}$, s_j is the similarity coefficient calculated for the line j. The ranking amplitude τ also considers that when two lines have the same similarity coefficient, we use the average ranking position for them. The first term $|\{j|s_j > s_d\}|$ counts the number of lines ranked before the faulty line. The term $|\{j|s_j \geq s_d\}|$ calculates the number of lines with the same or higher similarity coefficient compared to the faulty line [5].

$$\tau = \frac{|\{j|s_j > s_d\}| + |\{j|s_j \geq s_d\}| - 1}{2} \quad (2)$$

The value of accuracy is calculated considering the rank amplitude τ and the total number of lines of code N, according to (3).

$$q_d = \left(1 - \frac{\tau}{N-1}\right).100\% \quad (3)$$

The metric *error quality detection* aims to quantify a problem of diagnosis based on the observation of tests results. An error only appears when the faulty line is exercised by a test case. In this sense, the purpose of this metric is to measure the "unambiguity of the passed/failed" data in relation to the fault being exercised [3]. Equation 4 computes the metric using the definitions of Table I:

$$q_e = \frac{C_{11}}{C_{11} + C_{10}}.100\% \quad (4)$$

V. RESULTS

A. Exploratory Study

The dataset of this study contains 10,357 students' programs: 7,670 passed all tests and 2,687 failed at least one test. We are investigating what are the preconditions to use SBFL in students' programs, according to their tests' results. As SBFL is a technique to locate faults, clearly we are not interested on functionally correct submissions, i.e. when they pass all test cases. The requirements to use SBFL in students' programs, is to have traces of failed and passed executions. In the subset of codes that failed at least one test 1,031 codes did not passed any test (38.37%). It means that we can apply SBFL in 1,656 of 2,687 codes with defects (61.63%).

Although this result could be considered a large number, for the dataset we evaluated, it is important to understand what kind of submission is not "suitable" for using SBFL as a strategy for fault localization. Code submissions that did not pass in any test case may present failures on the algorithm or strategy to solve the problem. Possibly the student does not understood the problem specification or does not know how to program it correctly and need to start its code over. Depending on the test suite, it also may occur that a program passes "by

chance" in few test cases. Overall, SBFL is not suitable for these cases, as its purpose is to help to pinpoint faults. Students whose code submissions are "almost right" can benefit better of the information obtained from SBFL results.

In order to study the functioning of the technique in regards to the programs' defects configuration, we sought the dataset in order to locate programs that fits on the scenarios S_1, S_2, S_3 and S_4, as defined on Table II. We present examples of real student code for each scenario and discuss how the suspiciousness index *s* can be interpreted. Then we highlight the lessons learned when looking for defects using SBFL, in programs with such defect configurations. For each example code, the values of *s* are on the left side and the real defects are underlined.

S_1) Defect found in line with high *s* value. In this scenario, the faulty line is on the top positions of SBFL rank. This could be considered the ideal case: as *s* index is high. One who is looking for defects in the code can find it almost directly. Fig. 2 (of Section III) shows an example of program of this scenario. This program was tested against four test cases. It passes two and failed other two tests. Lines whose *s* value is 0.5 were executed in all test cases. The value of *s* of line 6 is 0.0, as this line was not executed in a failed test. The value of *s* of line 7 and 8 is 1.0, as they are executed in failed runs. In fact, line 8 is only executed when the test fails, as the real defect is found on it. Lines 7 and 8 are the top-ranked lines according to SBFL.

S_2) Defect not found in line with high *s* value. In this scenario, the faulty line is not on the top position of SBFL rank. The program example, showed in Fig. 3, was tested against 5 test cases and failed 2 tests. In this scenario, the faulty line (line 7) is not the top most line on SBFL rank. The highest value of *s* indexes is 1.0, corresponding to line 6. This happens because this line is executed in all the failed runs. Although the faulty line is not on the first rank position, it is one of the top most lines ranked. In this sense, one who is looking for defects in the code can find it in few lines attempts. So, this scenario also represents a successful case of SBFL use. This program can be fixed if we substitute the *if* statement on line 7 by an *elif* when checking if the sum is divisible by three.

```
0.4 | 1. num1 = int(raw_input())
0.4 | 2. num2 = int(raw_input())
0.4 | 3. num3 = int(raw_input())
0.4 | 4. sum= num1 + num2 + num3
0.4 | 5. if sum % 3 == 0 and sum % 5 == 0:
1.0 | 6.     print "fizzbuzz"
0.6 | 7. if sum % 3 == 0:
0.0 | 8.     print "fizz"
0.0 | 9. elif sum % 5 == 0:
0.0 | 10.    print "buzz"
```

Fig. 3. Student code with suspiciousness value on the left side and faulty statement underlined. Example of scenario 2.

S_3) Defect found in line with low *s* value. In this scenario, all lines have low values of index *s*. This indicates that all lines were executed in at least one successful test. However, some lines were not executed on failed runs. Although, no line in this example presented high values for the suspiciousness index *s*, when the rank is composed the faulty line (line 9) is in one of the highest positions of the rank, see Fig. 4. The problem of the faulty statement of this program is a truncate division operation. In Python version 2.7, the result of an integer division is truncated which may result in a failure for some inputs: when *b* is not a multiple of *2*a.* To effectively correct this code, it is necessary to convert *int* values into *floats*. This could be done in line 9 or in lines 2 and 3. This ambiguity may make it harder to find a way to correct the fault but the technique still gives a good hint. For this reason, we argue that SBFL, in this scenario, also can help to produce useful information about the fault localization.

```
0.1 | 1. import math
0.1 | 2. a = int(raw_input())
0.1 | 3. b = int(raw_input()
0.1 | 4. c = int(raw_input())
0.1 | 5. delta = b**2 - 4*a*c
0.1 | 6. if delta < 0:
0.0 | 7.     print "no real roots"
0.2 | 8. elif delta == 0:
0.3 | 9.     x = -(b)/(2*a)
0.3 | 10.    print "%s = %.2f" % ('x', x)
0.0 | 11.else:
0.0 | 12.    x1 = (-b+math.sqrt(delta))/(2*a)
0.0 | 13.    x2 = (-b-math.sqrt(delta))/(2*a)
0.0 | 14.    print "%s = %.2f" % ('x1', x1)
0.0 | 15.    print "%s = %.2f" % ('x2', x2)
```

Fig. 4. Student code with suspiciousness index value on the left side and faulty statement underlined. Example of scenario 3.

S_4) Defect not found in line with low *s* values. In this scenario, the faulty line does not appear on top most positions of SBFL rank. It means that SBFL failed in suggests the lines that contain the real defect. In this example, presented by Fig. 5, the faulty line is line 7. To better understand this situation, we debugged this code and we found that the failure happened because we short-circuit (don't evaluate) the *elif* statement when the second input (dna_2) is lower than the first input (dna_1). Thus, in any situation in which "dna_3 < dna_2 < dna_1" we will observe the same problem. To correct this program, it is necessary to substitute *elif* to an *if* condition.

This scenario is not common to happen. In fact, it was hard to find a situation where a defect was not indicated correctly by the highest SBFL value. We identified that situations like this can happen when the defect is located in the conditional structure, such as if/elif/else. When the alternative condition is accepted the other conditional test is not executed, even in failed runs. This situation poses a great challenge to this strategy.

Other perceptions about SBFL functioning and interpretation were observed on programs that does not fit on the characteristics of the scenarios previously described. However, SBFL can be applied and give us insights about the code execution or defect location. For example, when you have information about lines that were not executed (dead code) is possible to reason about a possible defect on the condition that make that code unreachable.

```
0.1 | 1. dna_1 = raw_input()
0.1 | 2. dna_2 = raw_input()
0.1 | 3. dna_3 = raw_input()
0.1 | 4. small = dna_1
0.1 | 5. if len(dna_2) < len(smal):
0.5 | 6.      smallest = dna_2
0.0 | 7. elif len(dna_3) < len(smal):
0.0 | 8.      smallest = dna_3
0.1 | 9.print "%s %d" % (smal,len(smal))
```

Fig. 5. Student code with suspiciousness index value on the left side and faulty statement underlined. Example of scenario 4.

B. Case Study

In this subsection we report on the results of the study that helped us to answer the research question that drove our study: Is the quality of diagnosis delivered by SBFL good enough to generate useful feedback about faults localization in novice programming assignments?

We studied five programming assignments (PA), emphasizing different learning outcomes of programming learning: (1) for loops, (2) sorting, (3) conditional structures, (4) lists and (5) while loops. These PA were chosen due to its high failure rate (53.59%). Table III shows each PA and the data about their total number of submissions and the number of failed submissions. Additionally, it shows the number of submissions suitable for SBFL use, failed submissions that pass at least one test.

TABLE III. PROGRAMMING ASSIGNMENTS DATA ABOUT SUBMISSIONS AND TESTS FAILURES

	Submissions	***Failed***	***Passed at least 1 test***
PA_1	29	24	19
PA_2	31	17	3
PA_3	18	10	5
PA_4	16	8	3
PA_5	87	38	28
TOTAL	181	53.59%	59.79%

The quality of SBFL diagnosis was assessed using the metrics defined in Section IV: precision, recall, accuracy and quality of error diagnosis. PA_2 and PA_4 values were omitted from the results table. Each one has only 3 submissions. The manual evaluation of these programs revealed severe defects caused by multiple lines in 4/6 programs. As we learned in the exploratory study, programs with such characteristics are not suitable to SBFL strategy. Table IV shows the average of the evaluative metrics for the others programming assignments.

TABLE IV. AVERAGE VALUES OF EVALUATION METRICS

	Recall	***Precision***	***qd***	***qe***
PA_1	1	0.18	0.84	0.17
PA_3	1	0.13	0.82	0.79
PA_5	1	0.08	0.73	0.30
Avg	1	0.10	79.67%	42.00%

The first two metrics (*recall* and *precision)* present contrasting values. *Recall* value is perfect for each programming assignment presented on Table 4. It means in practice, that all real faulty lines are successfully detected by SBFL technique. However, the precision value is low, meaning that many lines are detected, but some of them are false positives. This behavior was expected since SBFL strategy includes creating a rank of possible faulty lines to be inspected in a given order, so that a great part of the code could be exonerated of being inspected. It is likely that the top-ranked lines present the defect.

The quality of defect diagnosis is measured on metric *qd*. This measure shows the accuracy of SBFL in terms of the percentage of lines of code that do not need to be considered when searching for a defect on the code. The values obtained for each PA are considerable high. The metric quality of the error detection *qe* measures, in practice, how good is the test suite used to apply SBFL to a given programming assignment. It seems that the values obtained for this metric does not follow any trend. They can be considered low values, meaning that the quality of error detection for our dataset is not good. However, in a deeper analysis, we could not observe correlation between the measurements of: accuracy (*qd*) and the quality of error detection (*qe*). This result corroborates Abreu and colleagues' findings [3]. It means that even if the test suite used to apply SBFL technique results in low error detection quality, the diagnosis accuracy is still high.

VI. DISCUSSION

There is a set of necessary preconditions, regarding tests' results, to use SBFL to locate faults in a given programming assignment. Besides failing in at least one test case, what is obvious as our aim is to locate failures, it is also necessary to pass at least on test case. Though, the test suite must have at least 2 test cases. In dataset we evaluated on the exploratory study, 61.63% of code with failures met these requirements and could be used to assess SBFL.

The performance of SBFL technique depends on the programs' defect configuration. In the exploratory study, we identified four error scenarios and characterized other errors configurations. It helped us to better understand the meaning of SBFL suspiciousness indexes and devise hints on how to look for the fault. For example: If there is dead code in the program, it is important to understand why the test suite did not exercise such lines. Possibly, the defect is on the statement that precludes that code to be executed. Another lesson learned on using SBFL strategy is that if the defect were not found on the lines top-ranked consider analyze the neighborhood. It is worthy to verify conditional expressions near those lines to see if there is some defect there, especially if the defect were not find on top ranked lines.

There are indeed some scenarios in which SBFL performance is jeopardized. This technique is useful to highlight obvious and punctual defects. Programs containing multiple lines of errors, structural problems or error on its problem solving strategy is not suitable for applying SBFL. Even when manually evaluating punctual defects, there existed

few situations where SBFL was not able to locate the fault. The type of feedback generated with SBFL information, in programming learning context, is useful and ideal for situations in which the program is "almost right".

Overall, we considered the quality of diagnosis delivered by SBFL good enough to generate useful feedback about fault localization in novice programming assignments. At least for the dataset on the performed case study, SBFL accuracy was on average 80%. It means that guided by SBFL rank, one just need to scan 20% of the program to find it defect. This result approximates to the values obtained by other studies that applied SBFL in contexts such as spreadsheets [4], multi-agent systems [5] and software products [3].

VII. Conclusions and Future Work

This paper investigated the use of SBFL, a fault localization technique, on programs produced by students of introductory programming. This technique relies on program spectra, defined as a set of program' statements that were active during an execution. It predicts the likelihood of each program statement to be responsible for faulty executions. In order to make better use of this information, regarding to pedagogical context, instructors or AAS developers must fine-tune it before delivering to students.

We discussed how to interpret the values of SBFL suspiciousness index and the limitations of use of this technique. Our exploratory study characterized programs according to their errors configurations in scenarios. We claim, as lessons learned from this study, that SBFL is useful to pinpoint punctual defects. It is worthy to note that this technique is not fail-proof and there exist scenarios where looking only for the top most positions in SBFL rank may not be enough to find the fault. We report a case study evaluation, using real programming assignments, highlighting good results in terms of diagnostic accuracy: using SBFL we just need to look at 20% of the code in order to find the fault. This result corroborates with other studies found on the literature and obtained an approximate result when applying SBFL to other software engineering contexts.

The main contribution of this work, to instructors and AAS developers, is the investigation of SBFL benefits and limitations, as promising lightweight alternative to find faults in students programs and, as a new source of information, for student feedback generation. As future work we address the need of further investigation in the scenario in which the technique performance was jeopardized: when the fault is found on the conditional statement. Furthermore, it would be worthwhile to use item response item theory in order to validate the test suite provided to the programming assignment, since its quality is fundamental to this approach.

Acknowledgment

The authors would like to thank Programming I instructors' and students, at UFCG, for their valuable collaboration. This research was partially sponsored by the agreement No 754664/2010 between UFCG and ePol/DPF.

References

[1] R. Singh, S. Gulwani, and A. Solar-Lezama. "Automated feedback generation for introductory programming assignments". In PLDI, pp 15–26, 2013.

[2] M. Striewe and M. Goedicke. "Using run time traces in automated programming tutoring". In ITiCSE, pp 303–307, 2011.

[3] R. Abreu, P. Zoeteweij, and A. van Gemund. "On the Accuracy of Spectrum-based Fault Localization". In Proc. of Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION (TAICPART-MUTATION), pp 89–98, Windsor, UK, 2007. IEEE Computer Society.

[4] R. Abreu, J. Cunha, J. P. Fernandes, P. Martins, A. Perez, and J. Saraiva, "Smelling faults in spreadsheets," in Proc. 30th IEEE Int. Conf. Softw. Maintenance Evol., 2014, pp. 111–120

[5] L.Passos, R. Abreu and R. J. F. Rossetti., "Spectrum-based fault localisation for multi-agent systems", In Proc. 24th International Conference on Artificial Intelligence (IJCAI'15), 2015, AAAI Press, pp. 1134-1140

[6] K. Ala-Mutka, "A Survey of Automated Assessment Approaches for Programming Assignments", In Computer science education, vol. 15, pp 83-102, 2005.

[7] P. Ihantola, T. Ahoniemi, V. Karavirta and O. Seppälä, "Review of recent systems for automatic assessment of programming assignments", In Proc. 10th Koli Calling International Conference on Computing Education Research (Koli Calling '10). ACM, pp. 86-93.

[8] B. Cheang, A. Kurnia, A. Lim, and W. Oon. "On automated grading of programming assignments in an academic institution". Comput. Educ. 41, 2. pp. 121-131, September 2003.

[9] S. H. Edwards, "Improving student performance by evaluating how well students test their own programs", In J. Educational Resources in Computing, Vol 3, 2003, pp.1–24.

[10] H. S. Edwards, J. Snyder, M. A. Pérez-Quiñones, A. Allevato, D. Kim, and B. Tretola. 2009. "Comparing effective and ineffective behaviors of student programmers". In Proc. of the fifth international workshop on Computing education research workshop (ICER '09), pp. 3-14.

[11] S. Gulwani, I. Radiček, and F. Zuleger. "Feedback generation for performance problems in introductory programming assignments". In Proc. of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014), pp. 41-51.

[12] R. Cardell-Oliver, "How can software metrics help novice programmers?". In Proc. of the Thirteenth Australasian Computing Education Conference. Australian Computer Society, Inc. Vol 114, 2011, pp. 55-62.

[13] H. Blau and J. E. B. Moss, "FrenchPress Gives Students Automated Feedback on Java Program Flaws", In Proc. ACM Conference on Innovation and Technology in Computer Science Education, 2015, pp. 15-20.

[14] B. Edminson and S.H. Edwards, "Applying Spectrum-based Fault Localization to generate Debbuging Suggestions for Student Programmers", In Proc. of the 2015 IEEE International Symposium on Software Reliability Engineering Workshops, pp. 93-99.

[15] R. A. Bittencourt, D. M. B. dos Santos, C. A. Rodrigues, W. P. Batista and H. S. Chalegre, "Learning programming with peer support, games, challenges and scratch," *Frontiers in Education Conference (FIE), 2015. 32614 2015. IEEE*, El Paso, TX, 2015, pp. 1-9.