Fault Localization Evaluation Framework

IN4MATX 215

Milestone 4

Team Member:

Chiyu Cheng

Yi Zhou

Motivation and Research Question

Testing is one of the most critical and expensive tasks in the development of software. The process of detecting and locating bugs in a software system is called fault localization. Manually fault localization is acceptable for the small software system but unbearable for the large-scale system. Therefore, an automated fault localization technique is necessary for the real-world complex software system. There are many fault localization techniques like Tarantula, Jugular, and etc. However, there is no a comprehensive evaluation framework to evaluate and compare those techniques. An evaluation framework will guide the researchers to choose the right direction and method in the fault localization field. In addition, the process of evaluating different fault localization techniques will help us to understand this field better, and also help us to answer our research question - what is the most efficient, feasible, and accurate fault localization technique to locate the faults in the software system?

EMPIRICAL METHODOLOGY

Due to the limited amount of time, we are only be able to set up the test environment for one small program – “Triangle” [Appendix A] and its test suite [Appendix B] at this time. The main challenge we have now is verifying the correctness of the predicted result from different techniques, which can be time-consuming and tedious for the large-scale open source program. In the future direction section, we provide details about our extended test plan for extending our experiments on the large test set defect4j. In our current test plan, we are focused on running several techniques including Ochiai, Barinel, Dstar, and Tarantula on the Triangle program to locate the bugs. For the Tarantula, we use the Jacoco library to generate the code-coverage

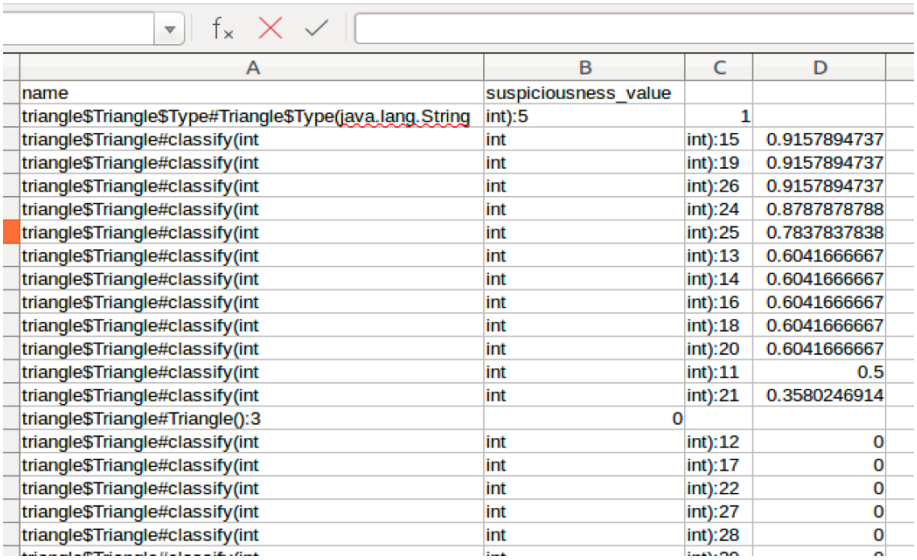
analysis report and use the Tarantula program to get the list of predicted buggy line. For the rest of techniques, we use the maven-plugin toolset named GZoltar to generate the predicted buggy line list.

There are three main criteria, includes feasibility, efficiency, and accuracy, that we used to evaluate the results of different techniques. In this project, feasibility means the easiness to implement and deploy one technique to existing programs. Efficiency is the amount of time that one technique spends to finish analyzing a certain number of testcases. Accuracy stands for the success rate to find the real concurrent bugs under a certain number of executions. Since the fault localization techniques provide a list of recommended buggy line number, which ranked based on its suspicious score, we record the position of the real buggy line in the recommended buggy line number list as the accuracy score. For example, if one FL technique generates a list of potential bug line number list {16, 15, 17, 20, 1, 2, etc} and line 15 is the real buggy line, then the accuracy score of this FL technique is 2.

Results

At this point, we still can't successfully build the automated run and verifying test script to integrate our selected techniques on the defect4j dataset. Thus, all the results in this section are based on the Triangle program, and all those data are collected and verified manually.
The experiments we performed are based on the modifying of Triangle source code. One of the examples of our experiments is modifying the line 18 in Triangle program to be the buggy line (See Appendix A for the detail), and run those techniques to analyze the potential buggy

location. We provide the detailed result of one trial and provide the overall result chart in the end.



	A	B	C	D
	name	suspiciousness_value		
	triangle$Triangle$Type#Triangle$Type(java.lang.String	int):5	1	
	triangle$Triangle#classify(int	int	int):15	0.9157894737
	triangle$Triangle#classify(int	int	int):19	0.9157894737
	triangle$Triangle#classify(int	int	int):26	0.9157894737
	triangle$Triangle#classify(int	int	int):24	0.8787878788
	triangle$Triangle#classify(int	int	int):25	0.7837837838
	triangle$Triangle#classify(int	int	int):13	0.6041666667
	triangle$Triangle#classify(int	int	int):14	0.6041666667
	triangle$Triangle#classify(int	int	int):16	0.6041666667
	triangle$Triangle#classify(int	int	int):18	0.6041666667
	triangle$Triangle#classify(int	int	int):20	0.6041666667
	triangle$Triangle#classify(int	int	int):11	0.5
	triangle$Triangle#classify(int	int	int):21	0.3580246914
	triangle$Triangle#Triangle():3	0		
	triangle$Triangle#classify(int	int	int):12	0
	triangle$Triangle#classify(int	int	int):17	0
	triangle$Triangle#classify(int	int	int):22	0
	triangle$Triangle#classify(int	int	int):27	0
	triangle$Triangle#classify(int	int	int):28	0

For the trial that modifying the line 18 of triangle program to be the buggy line, here is the one example output from the Tarantula technique, and it places the buggy line 18 in the 11th. The funny thing is this technique can't distinct the real buggy line with the line that cause failure. For example, line 19 is position that leads to the failure of testcases, which is placed in second with the highest suspiciousness score. However, line 18 is the real buggy line that leads to the wrong behavior of line 19. All four techniques failed to find the real buggy line 18 but placed line 19 in the beginning. The following table is the result for this trial.

Technique	Feasibility(min)	Efficiency(s)	Accuracy
Ochiai	15	5.1	10
Barinel	15	5	12
Dstar	15	5.1	9
Tarantula	30	7.2	11

After repeating four similar experiments that changing one line of code to be buggy line, we come up this conclusion table that contains the average score of all trials.

Technique	Feasibility(mins)	Efficiency (s)	Accuracy
Ochiai	15	4.9	5.1
Barinel	15	4.7	8.4
Dstar	15	5.1	9.1
Tarantula	30	8.9	7.2

Based on our experiments, we can conclude that the Ochiai is the most accurate technique among the selected techniques that can track the real buggy line, and it took more time to run the Tarantula technique than others. In addition, we observe that all of the techniques always place the line that leads to the failure of tests with higher rank than the real buggy code that is the root of the failure. In the next section, we discuss the potential problem and limitations of our current experiments, and the plan to improve it before the submission of final report.

Potential Problem and Future Direction

There are several flaws in our current experiment plan that we will improve in our final report. First, we only test the program that only has one bug at same time. However, there are generally existing multiple bugs at same time for the real-world software program. As we mentioned before, we will solve this problem after finishing integrate our selected techniques with the defect4j dataset. Second, we only evaluate the techniques for one software program with small number of trials, which leads to the less trustworthy of our current experiment result and conclusion. We will solve this flaw after finishing our automation script to verify the correctness

of the predictions for the defect4j project. In addition, we will add more techniques as many as possible before the deadline of final report.

Appendix A – Triangle

Credit to https://gitlab.cs.washington.edu/kevinb22/fault-localization-research/tree/master/src/triangle

```
1.  package triangle;
2.
3.  public class Triangle {
4.
5.      public enum Type {
6.          INVALID, SCALENE, EQUILATERAL, ISOSCELES
7.      };
8.
9.      public static Type classify(int a, int b, int c) {
10.         int trian;
11.         if (a <= 0 || b <= 0 || c <= 0)
12.             return Type.INVALID;
13.             trian = 0;
14.         if (a == b)
15.             trian = trian + 1;
16.         if (a == c)
17.             trian = trian + 2;
18.         if (b == a) //inserted bug: should be b == c
19.             trian = trian + 3;
20.         if (trian == 0)
21.             if (a + b <= c || a + c <= b || b + c <= a)
22.                 return Type.INVALID;
23.             else
24.                 return Type.SCALENE;
25.         if (trian > 3)
26.             return Type.EQUILATERAL;
27.         if (trian == 1 && a + b > c)
28.             return Type.ISOSCELES;
29.         else if (trian == 2 && a + c > b)
30.             return Type.ISOSCELES;
31.         else if (trian == 3 && b + c > a)
32.             return Type.ISOSCELES;
33.         return Type.INVALID;
34.     }
35. }
```

Appendix B – Test Cases for Triangle

```
1.  package triangle;
2.
3.  import junit.framework.TestCase;
4.  import static triangle.Triangle.Type.*;
5.
6.  public class TestSuite extends TestCase {
7.
8.     public void test1() {
9.          assertEquals (triangle.Triangle.classify(0,1301,1), INVALID);
10.    }
11.    public void test2() {
12.         assertEquals (triangle.Triangle.classify(1108,1,1), INVALID);
13.    }
14.    public void test3() {
15.         assertEquals (triangle.Triangle.classify(1,0,-665), INVALID);
16.    }
17.    public void test4() {
18.         assertEquals (triangle.Triangle.classify(1,1,0), INVALID);
19.    }
20.    public void test5() {
21.         assertEquals (triangle.Triangle.classify(582,582,582), EQUILATERAL);
22.    }
23.    public void test6() {
24.         assertEquals (triangle.Triangle.classify(1,1088,15), INVALID);
25.    }
26.    public void test7() {
27.         assertEquals (triangle.Triangle.classify(1,2,450), INVALID);
28.    }
29.    public void test8() {
30.         assertEquals (triangle.Triangle.classify(1663,1088,823), SCALENE);
31.    }
32.    public void test9() {
33.         assertEquals (triangle.Triangle.classify(1187,1146,1), INVALID);
34.    }
35.    public void test10() {
36.         assertEquals (triangle.Triangle.classify(1640,1640,1956), ISOSCELES);
37.    }
38.    public void test11() {
39.         assertEquals (triangle.Triangle.classify(784,784,1956), INVALID);
40.    }
41.    public void test12() {
42.         assertEquals (triangle.Triangle.classify(1,450,1), INVALID);
43.    }
44.    public void test13() {
45.         assertEquals (triangle.Triangle.classify(1146,1,1146), ISOSCELES);
46.    }
47.    public void test14() {
48.         assertEquals (triangle.Triangle.classify(1640,1956,1956), ISOSCELES);
49.    }
50.    public void test15() {
51.         assertEquals (triangle.Triangle.classify(-1,1,1), INVALID);
52.    }
53.    public void test16() {
54.         assertEquals (triangle.Triangle.classify(1,-1,1), INVALID);
55.    }
```

```
56.    public void test17() {
57.         assertEquals (triangle.Triangle.classify(1,2,3), INVALID);
58.    }
59.    public void test18() {
60.         assertEquals (triangle.Triangle.classify(2,3,1), INVALID);
61.    }
62.    public void test19() {
63.         assertEquals (triangle.Triangle.classify(3,1,2), INVALID);
64.    }
65.    public void test20() {
66.         assertEquals (triangle.Triangle.classify(1,1,2), INVALID);
67.    }
68.    public void test21() {
69.         assertEquals (triangle.Triangle.classify(1,2,1), INVALID);
70.    }
71.    public void test22() {
72.         assertEquals (triangle.Triangle.classify(2,1,1), INVALID);
73.    }
74.    public void test23() {
75.         assertEquals (triangle.Triangle.classify(1,1,1), EQUILATERAL);
76.    }
77.    public void test24() {
78.         assertEquals (triangle.Triangle.classify(0,1,1), INVALID);
79.    }
80.    public void test25() {
81.         assertEquals (triangle.Triangle.classify(1,0,1), INVALID);
82.    }
83.    public void test26() {
84.         assertEquals (triangle.Triangle.classify(1,2,-1), INVALID);
85.    }
86.    public void test27() {
87.         assertEquals (triangle.Triangle.classify(1,1,-1), INVALID);
88.    }
89.    public void test28() {
90.         assertEquals (triangle.Triangle.classify(0,0,0), INVALID);
91.    }
92.    public void test29() {
93.         assertEquals (triangle.Triangle.classify(3,2,5), INVALID);
94.    }
95.    public void test30() {
96.         assertEquals (triangle.Triangle.classify(5,9,2), INVALID);
97.    }
98.    public void test31() {
99.         assertEquals (triangle.Triangle.classify(7,4,3), INVALID);
100.        }
101.        public void test32() {
102.             assertEquals (triangle.Triangle.classify(3,8,3), INVALID);
103.        }
104.        public void test33() {
105.             assertEquals (triangle.Triangle.classify(7,3,3), INVALID);
106.        }
107.   }
```