

ECE 322

Lab Report 4

Arun Woosaree
XXXXXXX

November 5, 2019

1 Introduction

The purpose of this lab was to serve as an introduction to cause-effect graphs and decision tables. The goal was to become familiar with the use of these graphs and rules in black box testing, and to put into practice generating test cases using these tools. We tested two GUI applications written in Java. The first application, InsuranceCalculator takes in 3 inputs, a gender (Male/Female), an age, and number of claims. A cause-effect graph (Appendix A) was derived from 5 rules which were given. From the cause-effect graph, a decision table is derived, and test cases were made using the decision table. Depending on the input, this application outputs either \$750, \$1000, \$1500, or \$3000. The second application, BoilerShutdown is also a Java GUI application. It takes in 10 inputs labelled from A-J, which are checkboxes. Depending on the inputs, it will output either OK or FAIL. In this case, a cause-effect graph was provided, however, it was unsimplified. By tracing back, a simplified cause-effect graph was created (Appendix B), from which a decision table was derived. Finally, from the decision table, test cases were created. The purpose of cause-effect graphs is to divide the software specification into workable pieces. Once a decision table is generated, the test cases are derived using boundary value analysis, and also looking at the columns of the decision table.

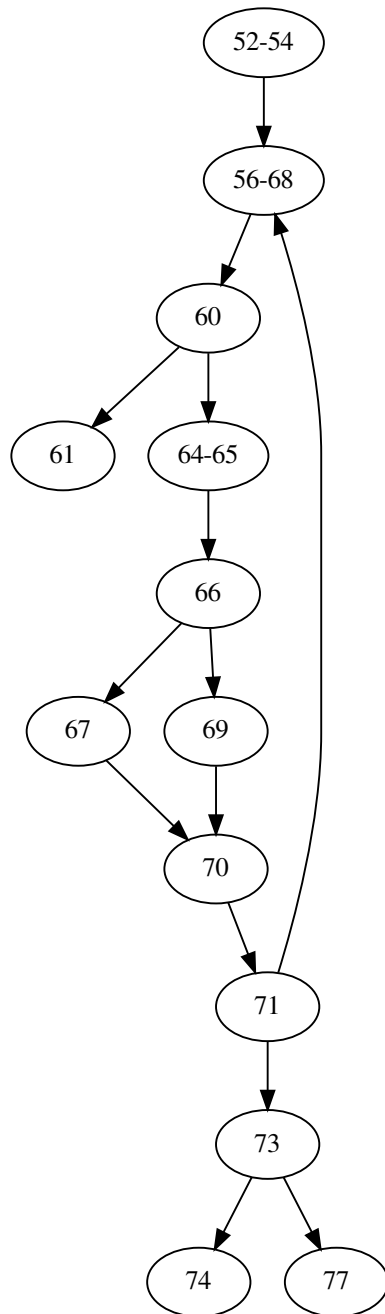
2 Task 1

For task one in this lab, the Bisect application was tested using White box testing methods. Bisect is a program written in Java, which uses the well-known bisection algorithm in mathematics to attempt to find the root of any polynomial in a given interval that crosses $x = 0$ somewhere. That is, wherever the polynomial intersects $x = 0$. The algorithm which the program implements is outlined as follows:

1. Calculate c , the midpoint of the interval: $c = \frac{a+b}{2}$

2. Calculate the polynomial's value at $f(c)$
3. If $|f(c)|$ is within the tolerance, stop
4. Check the sign of $f(c)$ and replace either $(a, f(a))$ or $(b, f(b))$ with $(c, f(c))$ such that the interval crosses $x = 0$ somewhere and repeat until step 3 quits

Since this is white box testing, we have access to the source code for this application. By inspecting the source code, the following control flow graph was generated for the application. The numbers in each node represent the line numbers in the source code in **Bisect.java**.



By inspecting the source code and understanding the bisection algorithm, some JUnit test cases were generated. These test cases can be found in Appendix A. A summary of these test cases is outlined in the table below.

Test Id	Description	tolerance	max iterations	polynomial	x_1	x_2	Expected	Actual
1	Exception is thrown when both $f(x_1)$ and $f(x_2)$ are > 0	0.000001	50	$x^2 - 1$	-5	5	Exception	Exception
2	Exception is thrown when maximum iterations are exceeded	0.000001	1	$x + 100$	-150	1000000000	Exception	Exception
3	Test constructor with tolerance	50.0	50	x	-10	10	0	0
4	Test constructor with tolerance and max iterations	0	50	x	-10	10	0	0
5	Max iterations getter	0.000001	500	x	n/a	n/a	500	500
6	Max iterations setter	0.000001	500	x	n/a	n/a	500	500
7	Tolerance getter	0.0	50	x	n/a	n/a	0.0	0.0
8	Tolerance setter	0.0	50	x	n/a	n/a	0.0	0.0
9	Normal test case, make sure that $f(result) \leq result$, and it iterates more than once	0.000001	50	$x^2 - 1$	-1.5	0.5	-1	-1

While code coverage is a useful metric for determining the quality of tests, it should not be the only factor in determining the quality of the tests. After all, it is possible to have all tests cover all lines and branches, but the tests may not be checking for the correctness in execution of these statements and branching conditions. What is useful however, is knowing which lines of the code have been executed in the tests. That way, if a line is missed for example, code coverage criterion will bring to light that fact that the line of code was not covered. Furthermore, branch coverage will also ensure that all branches have been executed in the tests. In short, coverage criterion is effective for letting us know when a part of the code *for sure* is not being tested. It does not however, guarantee that the correctness of the covered statements and branches.

Based solely on the coverage criteria, it would be in my opinion foolish, or at best naive to be completely confident in the application being bug-free. Having well-thought out tests that adequately cover the functionality of the programs *in addition to* the code coverage criterion, however, does increase my confidence in the program working. Tests that are written such that they only cover the code coverage criterion will not adequately cover the correctness. The tests have to

be made with the knowledge of how the Bisection algorithm works, and testing that the program computes the result as expected. With the test cases made in this lab, (available in Appendix A), I am fairly confident that the portion of the program that does the Bisection algorithm works correctly, since they were crafted with the knowledge of how the Bisection algorithm works.

In general, the number of paths available to test is a number between $n+1 \leq x \leq 2^n$, where n is the number of branches. This depends on when the branches merge. Additionally, for this application, there is a branch decision on every loop iteration. So, the number of test cases depends on the number of maximum iterations. Even with the default value of 50 iterations, we can see how quickly the number of tests needed for path coverage can blow up exponentially to ridiculous proportions, which is why path coverage is not realistic.

3 Task 2

For the second part of the lab, we are to assume that we have a system with three independent variables: A, B, C. Each variable has three possible values: 0, 1, 2. There is no actual testing for this portion, it is a conceptual exercise.

In total, there are $3 \times 3 \times 3 = 27$ test cases if we were to do combinatorial testing. We are looking for a standard orthogonal array that can fit 3^3 . From <http://neilsloane.com/oadir/>, we find that the $L_9(3^4)$ standard orthogonal array works. Below is the set of test cases using the orthogonal array mentioned above:

0	0	0	0
0	1	1	2
0	2	2	1
1	0	1	1
1	1	2	0
1	2	0	2
2	0	2	2
2	1	0	1
2	2	1	0

The PICT program (<https://github.com/microsoft/pict>) was used to generate test cases with the following input:

```
A: 0, 1, 2
B: 0, 1, 2
C: 0, 1, 2
```

It should be noted that strangely, there seems to be different outputs depending on the operating system that pict is run on.

On GNU/Linux systems, the output is as follows:

```
A      B      C
0      0      0
```

0	2	1
2	0	2
1	0	1
2	1	0
1	1	2
0	1	1
1	2	0
0	2	2
2	2	1

On Windows, the output is as follows:

A	B	C
1	2	0
2	0	2
0	1	0
2	2	1
0	0	1
1	1	2
2	0	0
0	2	2
1	0	1
2	1	1

On MacOS, the output is as follows:

A	B	C
0	2	2
2	2	1
0	0	0
2	1	0
2	0	2
1	2	0
0	1	1
1	1	2
1	0	1

What is strange is that on the Windows and MacOS systems, 10 test cases are generated, while on GNU/Linux, 11 test cases are generated.

Given the inconsistent outputs of the pict tool depending on the operating system that the program is run on, I am not sure about the effectiveness of the tool for test case generation. Ideally, the test cases generated should be consistent, since this they are not supposed to be chosen randomly. If the program worked consistently, however, it would be quite effective since not much effort would be required to come up with the test cases, and we have the guarantee that every pair of inputs is tested.

Compared to the orthogonal array, the pict tool generated more test cases

Pairwise testing is fairly useful, especially when you consider it versus combinatorial testing. In this toy example, the number of test cases was not reduced that much, (27 for combinatorial testing, versus 9 with an orthogonal table), but one can imagine the reduction in test cases when there are more inputs, and more possible values that these inputs can take on. Compared with random combinations, pairwise testing gives us the guarantee that we are testing every single pair of input factors, whereas random combinations does not have this guarantee, and there are far more combinations to choose from. However, random combinations can by chance reveal errors that pairwise testing would not catch.

Pairwise testing catches errors where two inputs interact with each other. It does not take into account the effect that multiple inputs might have together. For example, imagine an application that requires three inputs to be in a certain state before something gets activated. Using pairwise testing there is a possibility that we would miss testing the part of the app, where three inputs are in the state for something to happen, since the pairwise tests would only focus on the combination of pairs of the inputs. In this way, core functionality of the application can be missed in testing. Furthermore, with pairwise testing it is possible to miss the more probable combinations when selecting the test data. However, given the reduction in the number of test cases versus exhaustive testing, not to mention the cost of running so many tests, these weaknesses may be justifiable to some testers, given that pairwise testing usually results yields higher defect detection rates, increases the test coverage, all while using significantly less test cases. As mentioned above, to cover all possible inputs for the application in this example, a total of 27 test cases are required.

4 Conclusion

In this lab, we were introduced to the black box testing strategy of creating cause-effect graphs and decision tables to help come up with test cases. While it would not be fair to say that one testing strategy is universally better than the other, since it would be better to match each problem at hand to the appropriate testing strategy, compared to other testing methods previously learned like dirty testing, error guessing, EPC, and weak $n \times 1$ strategy, this method seems to demand the most amount of thought and effort for creating the test cases. From a subjective standpoint, making cause effect graphs and deriving test cases from them is an okay testing technique, but not one of my favourites. While in terms of number of test cases, it is efficient in that it generates very few test cases while still having generally good coverage, the effort required to generate the test cases does not seem like it is worth it for most applications. Additionally, it seems to not cover some scenarios, like in Task 1 with the CarInsurance program, where invalid inputs were not tested. There are some scenarios, however, where the testing technique seems appropriate to use. For example, in Task 2 with the BoilerShutdown application, it was impossible to enter an invalid input, and we were strictly concerned with the inputs (causes), and their effects. By creating

a simplified cause effect graph, we created few, but effective test cases. Overall, making cause-effect graphs and decision tables definitely has its place in black box testing. Like with any other testing technique, there are scenarios where it shines, and other cases where another testing technique would work better.

A Bisect JUnit Tests














```
1 import static org.junit.jupiter.api.Assertions.*;
2 import org.junit.jupiter.api.Test;
3
4 class BisectTest {
5
6     @Test
7     void runTest61() {
8         Bisect b = new Bisect(value -> Math.pow(value, 2) - 1);
9         assertThrows(Bisect.RootNotFound.class, () -> b.run(-5, 5));
10    }
11
12    @Test
13    void normalTestCase() throws Bisect.RootNotFound {
14        Bisect b = new Bisect(value -> Math.pow(value, 2) - 1);
15        double result = b.run(-1.5, 0.5);
16        assert Math.abs(Math.pow(result, 2) - 1) <= b.getTolerance();
17    }
18
19    @Test
20    void runTestMaxIter(){
21        Bisect b = new Bisect(1, value -> value + 100);
22        assertThrows(Bisect.RootNotFound.class, ()->b.run(-150, 1000000000));
23    }
24
25    @Test
26    void runTesttolpolynomialconstructor() throws Bisect.RootNotFound {
27        Bisect b = new Bisect(50.0, value -> value);
28        assertEquals(0, b.run(-10, 10));
29    }
30
31    @Test
32    void runTesttolmaxiterpolynomialconstructor() throws Bisect.RootNotFound {
33        Bisect b = new Bisect(0, 50, value -> value);
34        assertEquals(0, b.run(-10, 10));
35    }
36
37    @Test
38    void runTestGetMaxIterations(){
39        Bisect b = new Bisect(500, value->value);
40        assertEquals(500, b.getMaxIterations());
41    }
42
43    @Test
44    void runTestSetMaxIterations(){
```

```

45         Bisect b = new Bisect(value -> value);
46         b.setMaxIterations(500);
47         b.setMaxIterations(-1);
48         assertEquals(500, b.getMaxIterations());
49     }
50
51     @Test
52     void runTestGetTolerance(){
53         Bisect b = new Bisect(0.0, value->value);
54         assertEquals(0.0, b.getTolerance());
55     }
56
57     @Test
58     void runTestSetTolerance(){
59         Bisect b = new Bisect(value -> value);
60         b.setTolerance(0.1);
61         b.setTolerance(-1);
62         assertEquals(0.1, b.getTolerance());
63     }
64
65 }

```

B Code Coverage Results

Element	Class, %	Method, %	Line, %	Branch, %
 com				
 images				
 java				
 javax				
 jdk				
 META-INF				
 netscape				
 org				
 sun				
 toolbarButtonGraphics				
 Bisect	100% (2/2)	100% (10/...	100% (39/...	100% (8/8)
 BisectTest	100% (1/1)	100% (9/9)	100% (24/...	0% (0/1)
 Main	0% (0/2)	0% (0/2)	0% (0/8)	0% (0/1)

center

[[all classes](#)] [[<empty package name>](#)]

Coverage Summary for Class: Bisect (<empty package name>)

Class	Method, %	Line, %
Bisect	100% (9/ 9)	100% (38/ 38)
Bisect\$polynomial		
Bisect\$RootNotFound	100% (1/ 1)	100% (1/ 1)
total	100% (10/ 10)	100% (39/ 39)

```

1 public class Bisect {
2
3     private double tolerance;
4     private int maxIterations;
5     private polynomial func;
6
7     public Bisect(polynomial f) {
8         func = f;
9         tolerance = 0.000001;
10        maxIterations = 50;
11    }
12
13    public Bisect(double tol, polynomial f) {
14        func = f;
15        tolerance = tol;
16        maxIterations = 50;
17    }
18
19    public Bisect(int maxIter, polynomial f) {
20        func = f;
21        tolerance = 0.000001;
22        maxIterations = maxIter;
23    }
24
25    public Bisect(double tol, int maxIter, polynomial f) {
26        func = f;
27        tolerance = tol;
28        maxIterations = maxIter;
29    }
30
31    public double getTolerance() {
32        return tolerance;
33    }
34
35    public void setTolerance(double tol) {
36        if (tol > 0)
37            tolerance = tol;
38    }
39
40    public double getMaxIterations() {
41        return maxIterations;
42    }
43
44    public void setMaxIterations(int maxIter) {
45        if (maxIter > 0)
46            maxIterations = maxIter;
47    }
48
49
50    public double run(double x1, double x2) throws RootNotFound {
51
52        int iterNum = 1;
53        double f1, f2, fmid;
54        double mid = 0;
55
56        do {
57            f1 = func.eval(x1);
58            f2 = func.eval(x2);
59
60            if (f1 * f2 > 0) {
61                throw new RootNotFound();

```

```
62         }
63
64         mid = (x1 + x2) / 2;
65         fmid = func.eval(mid);
66         if (fmid * f1 < 0)
67             x2 = mid;
68         else
69             x1 = mid;
70         iterNum++;
71     } while (Math.abs(x1 - x2) / 2 >= tolerance && Math.abs(fmid) > tolerance && iterNum <= maxIterations);
72
73     if (iterNum >= maxIterations) {
74         throw new RootNotFound();
75     }
76
77     return mid;
78 }
79
80 public interface polynomial {
81     public double eval(double value);
82 }
83
84 public class RootNotFound extends Exception {
85
86 }
87 }
```

generated on 2019-11-05 17:29