ECE 322
Lab Report 4

Arun Woosaree
XXXXXXX

November 4, 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 appplication, 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 algotithm which the program implements is outlined as follows:

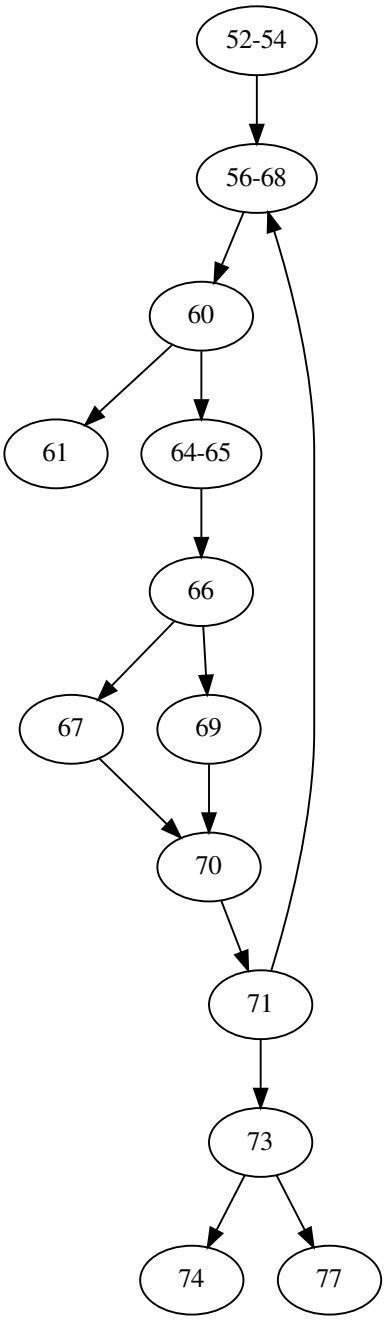
1. Calculate c, the midpoint of the interval: $c = \frac{a+b}{2}$

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**


52-54
56-68
60
61
64-65
66
67
69
70
71
73
74
77

3 Task 2

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 x 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

```java
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.Test;

class BisectTest {

    @Test
    void runTest61() {
        Bisect b = new Bisect(value -> Math.pow(value, 2) + 1);
        assertThrows(Bisect.RootNotFound.class, () -> b.run(-1, 1));
    }

    @Test
    void runTest67() throws Bisect.RootNotFound {
        Bisect b = new Bisect(value -> Math.pow(value, 2) - 1);
        b.run(-1.5, 0.5);
    }

    @Test
    void runTestMaxIter(){
        Bisect b = new Bisect(1, value -> value + 100);
        assertThrows(Bisect.RootNotFound.class, ()->b.run(-150, 1000000000));
    }

    @Test
    void runTesttolpolynomialconstructor() throws Bisect.RootNotFound {
        Bisect b = new Bisect(50.0, value -> value);
        assertEquals(0, b.run(-10, 10));
    }

    @Test
    void runTesttolmaxiterpolynomialconstructor() throws Bisect.RootNotFound {
        Bisect b = new Bisect(0, 50, value -> value);
        assertEquals(0, b.run(-10, 10));
    }

    @Test
    void runTestGetMaxIterations(){
        Bisect b = new Bisect(50, value->value);
        assertEquals(50, b.getMaxIterations());
    }

    @Test
    void runTestSetMaxIterations(){
        Bisect b = new Bisect(value -> value);
```

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