


ECE 322 Lab Report #3


 Benjamin Kong
1573684

November 3, 2021

Introduction

The purpose of this lab was to introduce rudimentary techniques of white-box testing, specifically unit testing. Furthermore, this lab introduced pairwise test case generation tools.

Whereas the tester does not know the inner workings of a program when doing black-box testing, for white-box testing, the tester knows the inner workings of the program being tested. This typically means the tester has access to source code. Using this knowledge, the tester can then come up with test cases to test the program. In this lab, we will focus on control flow testing. Control flow testing includes four fundamental coverage criteria: statement coverage, branch coverage, condition coverage, and path coverage. We will be performing control flow testing on a Python program that performs the bisection method (a simple algorithm to solve equations in the form of $f(x) = 0$ given an interval in which the root can be found).

In this lab, we also look at pairwise testing. Instead of producing test cases for every possible combination of inputs, pairwise testing makes testing more efficient by allowing testers to construct a set of test cases that covers all combinations of test data for each pair of variables. In this lab, we will be using a pairwise test case generation tool to build a set of test cases. Specifically, in this lab, we will be using allpairspy, a Python library that calculates approximations to orthogonal arrays which are then used to generate test cases. We will be considering a system with three independent variables A , , and C that take on three possible values 0, 1, and 2.

Questions

Q1

Unit testing is a white-box testing technique. Unit testing is used to verify individual units of code. Typically, in a codebase, individual functions will be tested. Often, each function will have several unit tests in order to test different end cases and scenarios. When a unit test fails, then the tester knows that the function being tested may have an error somewhere. Unit tests should be done as often as possible. We use unit tests because they are helpful for helping

developers make sure their changes do not affect other individual units of code. Furthermore, code is typically better designed when testing is done as part of the development process.

Q2

Code coverage is a metric for measuring the adequacy and completeness of test cases in a given program and is a useful metric to help developers understand how much of their code is tested. It can be a useful metric in assessing test performance and quality of software. There are four fundamental criteria for code coverage:

- Statement coverage: every statement in the code has to be executed at least once,
- Branch coverage: every branch in the code has to be executed at least once,
- Condition coverage: all possible combinations of conditions in compound decisions must be exercised, and
- Path coverage: all possible logical paths must be exercised.

Q3

Pairwise testing is a black-box test technique. Whereas combinational testing produces test cases with every permutation of each input, pairwise testing is significantly more efficient since it produces test cases that covers all combinations of the test data for each pair of variables. Essentially, a subset of all combinations of inputs is produced. While it is not exhaustive, it is still very effective at finding bugs since most defects are caused by interaction of at most two inputs. Furthermore, it produces significantly less test cases than combinational testing.

Part 1 — Bisect

Q1

The bisection method is a simple algorithm that aims to solve equations of the form $f(x) = 0$ when given an interval $x_1 \leq x \leq x_2$ in which the root x can be found. Our function in the program `mybisection.py` solves this via iteration. First, the solver is initialized with the function `func` and the function is given the variables `x1` and `x2` which represent the start and end of the interval in which to search for the root. Then, in each iteration, the function

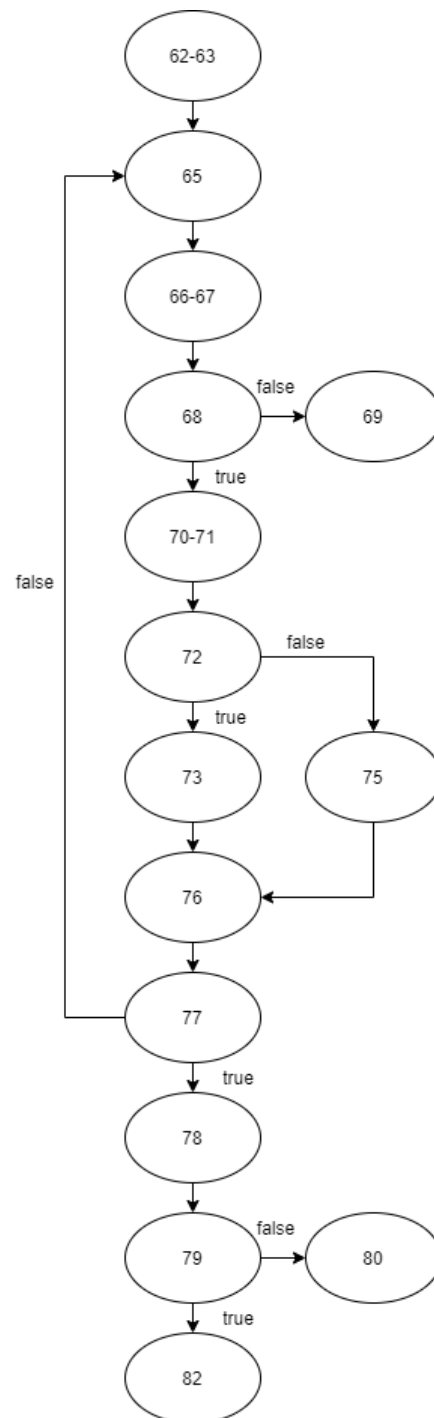
1. Finds the values of the function at the endpoints `f1` and `f2`,
2. Checks if the root is not found (errors if no root found),
3. Calculates the midpoint `mid` and the value of the function at the midpoint `fmid`,
4. Checks whether to change `x1` or `x2` to `mid` depending on the sign of `fmid`,
5. Increments the iteration number `iterNum`, and
6. Breaks the iteration if `fmid` is within the tolerance or `iterNum` has exceeded the maximum iteration count. Otherwise, return to step 1 and repeat.

Once the iteration has ended, the program then checks if the maximum iteration count has been exceeded, throwing an error if it has. Otherwise, it returns the current value of `mid`.

Q2

The control flow graph diagram is presented below.

Figure 1: Control flow graph diagram. Numbers represent the line numbers in `mybisect.py`.



Q3

The table is displayed below. The default tolerance is 0.000001 and the default max iterations is 50 unless otherwise stated in the description.

Table 1: Test cases for `bisect.py`

Test ID	Description	func	x1	x2	Expected	Pass
1	Normal case	$x + 1$	-10	10	0	Y
2	Exception when $f(x1) > 0$ and $f(x2) > 0$	$x^2 + 1$	-10	10	ValueError	Y
3	Exception when maximum iterations exceeded	x	-1	1e10	ValueError	Y
4	Default constructor (no args)	n/a	-10	10	TypeError	Y
5	Custom tolerance (use <code>tol = 10.0</code>)	x	-10	10	0	Y
6	Custom max iterations (max it = 1000)	x	-1	1e10	0	Y
7	Custom tolerance & max iterations (tol = 10.0, max it = 1000)	x	-10	10	0	Y
8	Getter for tolerance	x	n/a	n/a	tol = 1e-7	Y
9	Setter for tolerance (use <code>tol = 10.0</code>)	x	n/a	n/a	tol = 10.0	Y
10	Getter for max iterations	x	n/a	n/a	max it = 50	Y
11	Setter for max iterations (use <code>max it = 500</code>)	x	n/a	n/a	max it = 5	Y

Q4

The results of running the coverage report in Python are displayed in question 5. Note that the 86% coverage of `mybisect.py` is a result of not testing the `main` function as testing `main` is practically pointless.

1. Statement coverage is relatively easy to obtain, but requires a decent amount of test

- cases. For example, the getters and setters each had to be called in order to obtain statement coverage for them. Out of the criteria, this was the easiest to achieve.
2. All branches of the program were tested. It is not too difficult to attain full coverage of all branches, but doing so requires many test cases and some analysis in order to enter all branches. It is a bit of additional work compared to statement coverage.
 3. The tests were able to achieve full statement and branch coverage. Again, the only lines of code that were not tested were from the `main` function in `mybisection.py`.
 4. No errors were discovered. However, coverage tests do not mean the tests were effective. If more tests were created, errors may be revealed even though we have 100% coverage.
 5. Path testing would be practically impossible. The number of paths to test is typically $n + 1 \leq \# \text{Tests} \leq 2^n$ where n is the number of branches. As we can see, the number of test cases we'd need to complete quickly spirals out of control, making path testing impractical.

Q5

The code is provided below.

```
1  import unittest
2  from mybisection import Polynomial
3  from mybisection import MyBisection
4
5  class TestMyBisection(unittest.TestCase):
6      def test_normal(self):
7          # Normal test case
8          f = Polynomial(1, 1) # x + 1
9          b = MyBisection(f)
10         result = b.run(-10, 10)
11         self.assertEqual(0, f(result), None,
12                          None, b.tolerance)
13
14     def test_both_positive(self):
15         # f(x1) > 0 and f(x2) > 0
16         f = Polynomial(1, 0, 1) # x^2 + 1
17         b = MyBisection(f)
18         self.assertRaises(ValueError, b.run, -10, 10)
19
20     def test_exceed_max_iteration(self):
21         # Test max iterations exceeded
22         # This takes more than 50 iterations
23         f = Polynomial(1, 0) # x
24         b = MyBisection(f)
```

```
25         self.assertRaises(ValueError, b.run,
26                             -1, 1000000000000)
27
28     def test_default_constructor(self):
29         # Test default constructor (no args)
30         b = MyBisect()
31         self.assertRaises(TypeError, b.run, -10, 10)
32
33     def test_tolerance_constructor(self):
34         # Test tolerance constructor (tol = 10.0)
35         f = Polynomial(1, 0) # x
36         b = MyBisect(10.0, f)
37         result = b.run(-10, 10)
38         self.assertEqual(0, result)
39
40     def test_max_it_constructor(self):
41         # Test max iterator constructor (max it = 100)
42         # Increasing the max iteration acc should
43         # allow us to find the result unlike
44         # test_exceed_max_iteration() which reaches
45         # the max iteration count.
46         # This should take 51 iterations
47         f = Polynomial(1, 0) # x
48         b = MyBisect(100, f)
49         result = b.run(-1, 1000000000000)
50         self.assertAlmostEqual(0, f(result), None,
51                                 None, b.tolerance)
52
53     def test_tolerance_and_max_it_constructor(self):
54         # Test tolerance and max iterator constructor
55         # (tol = 10.0, max it = 100)
56         f = Polynomial(1, 0) # x
57         b = MyBisect(10.0, 100, f)
58         result = b.run(-1, 1)
59         self.assertAlmostEqual(0, f(result), None,
60                                 None, b.tolerance)
61
62     def test_tolerance_getter(self):
63         # Test tolerance getter
64         f = Polynomial(1, 0) # x
65         b = MyBisect(f)
66         self.assertEqual(0.000001, b.tolerance)
67
68     def test_tolerance_setter(self):
69         # Test tolerance setter
70         f = Polynomial(1, 0) # x
71         b = MyBisect(f)
72         b.tolerance = 10.0
73         self.assertEqual(10.0, b.tolerance)
74
75     def test_max_iterations_getter(self):
76         # Test max iterations getter
77         f = Polynomial(1, 0) # x
78         b = MyBisect(f)
79         self.assertEqual(50, b.maxIterations)
```



```

80
81     def test_max_iterations_setter(self):
82         # Test max iterations setter
83         f = Polynomial(1, 0) # x
84         b = MyBisect(f)
85         b.maxIterations = 5
86         self.assertEqual(5, b.maxIterations)
87
88 if __name__ == "__main__":
89     unittest.main()

```

An image of the coverage report is provided below.

Figure 2: Coverage report of mybisect.py. The only relevant value has been marked with a red arrow.

Coverage report: 58%

Module	statements	missing	excluded	coverage
C:\Program Files\JetBrains\PyCharm 2021.2.3\plugins\python\helpers\pycharm_jb_runner_tools.py	173	54	0	69%
C:\Program Files\JetBrains\PyCharm 2021.2.3\plugins\python\helpers\pycharm_jb_serial_tree_manager.py	56	12	0	79%
C:\Program Files\JetBrains\PyCharm 2021.2.3\plugins\python\helpers\pycharm_jb_unittest_runner.py	24	8	0	67%
C:\Program Files\JetBrains\PyCharm 2021.2.3\plugins\python\helpers\pycharm_jb_utils.py	71	34	0	52%
C:\Program Files\JetBrains\PyCharm 2021.2.3\plugins\python\helpers\pycharm\teamcity_init_.py	6	1	0	83%
C:\Program Files\JetBrains\PyCharm 2021.2.3\plugins\python\helpers\pycharm\teamcity\common.py	98	59	0	40%
C:\Program Files\JetBrains\PyCharm 2021.2.3\plugins\python\helpers\pycharm\teamcity\diff_tools.py	59	31	0	47%
C:\Program Files\JetBrains\PyCharm 2021.2.3\plugins\python\helpers\pycharm\teamcity\messages.py	153	63	0	59%
C:\Program Files\JetBrains\PyCharm 2021.2.3\plugins\python\helpers\pycharm\teamcity\unittestpy.py	217	127	0	41%
C:\Users\Ben\OneDrive\10 - 2021 Fall\ECE 322\Lab3\Code\mybisect.py	72	10	0	86%
C:\Users\Ben\OneDrive\10 - 2021 Fall\ECE 322\Lab3\Code\mybisect_test.py	55	19	0	65%
Total	984	418	0	58%

coverage.py v6.1.1, created at 2021-11-03 01:38 -0600


Part 2 — Pairwise

We will be investigating pairwise testing in this section. We will be considering a system with three independent variables A , B , and C that take on three possible values 0, 1, and 2.

Q1

The closest match was this orthogonal array found at <https://www.york.ac.uk/depts/maths/tables/19.gif>. It had 9 rows and 4 columns, so we simply remove the last column.


Table 2: Test cases from closest orthogonal array.

Test ID	A		C
1	0	0	0
2	0	1	1
3	0	2	2
4	1	0	1
5	1	1	2
6	1	2	0
7	2	0	2
8	2	1	0
9	2	2	1

Q2

Using the `allpairs` estimation tool, we were able to generate pairwise test cases. This is displayed below. Furthermore, the code is provided in Q4.

Table 3: Test cases from the `allpairs` estimation tool.

Test ID	A		C
1	0	0	0
2	1	1	0
3	2	2	0
4	2	1	1
5	1	0	1
6	0	2	1
7	0	1	2
8	1	2	2
9	2	0	2

Q3

1. Both the custom program and finding an orthogonal array resulted in the same number of test cases. However, finding and modifying the orthogonal array was a little bit more work than generating them using code.
2. Both these tools were effective in generating test cases. They were also both equally effective at reducing the number of test cases. Compared to combinational testing, both methods reduced the number of test cases by a factor of 3 (since combinational testing would result in $3 \times 3 \times 3 = 27$ test cases).
3. Pairwise testing is effective at catching errors since many bugs are usually caused by single inputs or the interaction between two inputs.
4. Pairwise testing drastically reduces the number of test cases compared with combinational testing. However, it is not as effective as combinational testing as a few edge cases may slip through. For example, a bug may be caused by an interaction between three or more inputs. However, these cases are very rare, and the benefit of drastically reducing the number of test cases makes pairwise testing a good choice in general for testing.

Q4

The code is provided below.

```
1  from allpairsy import AllPairs
2
3  def getallpairsy(parameters):
4      print("PAIRWISE:")
5      for i, pairs in enumerate(AllPairs(parameters)):
6          print("{:2d}: {}".format(i, pairs))
7
8  def main():
9      parameters = [
10         [0, 1, 2],
11         [0, 1, 2],
12         [0, 1, 2],
13     ]
14
15     getallpairsy(parameters)
16
17 if __name__ == "__main__":
18     main()
```

Conclusion

The purpose of this lab was to introduce rudimentary techniques of white-box testing, specifically unit testing. Furthermore, this lab introduced pairwise test case generation tools.

The first part of this lab focused on coverage testing. There are four fundamental coverage criteria: statement coverage, branch coverage, condition coverage, and path coverage. We also created a control flow graph to analyze the program. By having the source code on hand, we were able to have a detailed analysis of the code. This allowed us to create unit tests that touched all parts of the code. We were able to cover the statement and branch criterion of coverage testing. While coverage testing by itself does not mean the code is error free, it can help show what parts of the code are not tested. When coverage testing is combined with further unit tests and other tests, we have a strong case that the code we have written is error free.

The second part of this lab focused on pairwise testing. Pairwise testing is effective at drastically reducing the number of test cases. We first created test cases by finding an appropriate orthogonal array, then generated test cases using the `allpairs` estimation tool. We found that both these methods were effective at generating pairwise test cases, and that both were equally adept at reducing the number of test cases compared to combinational testing. We also discussed the pros and cons of pairwise testing: pairwise testing drastically reduces the number of test cases, but may let some cases slip through when three or more inputs interact with each other.