Software Testing and Analysis | Assignment 1

Max Moverley
6992267
COSC 3P95
October 2023

1.

Soundness in software analysis refers to the program's capability to detect genuine errors or vulnerabilities in the code. A sound program will never report a bug that doesn't exist. Completeness is the program's capability to detect *all* errors or vulnerabilities in the code, regardless of whether or not they exist.

A true positive is when the analysis tool correctly finds an error/bug in the program that is a legitimate bug. In other words it's "doing its job".

A true negative is when the tool doesn't report a bug that doesn't exist. If the snippet of code it's given doesn't have any bugs, and the tool doesn't report any, it is a true negative.

A false positive is when the analysis tool reports a bug in the code that doesn't exist. Essentially a "false alarm".

A false negative is the opposite of a false positive, wherein a tool fails to report a problem that exists.

Going back to soundness and completeness, another way to look at them would be that soundness prevents false positives (all errors reported are legitimate errors) and completeness prevents false negatives (there are no errors in the code that a complete algorithm hasn't reported).

The true/false positive/negative terminology would be flipped on its head if the word "positive" meant not finding a bug. In this unusual circumstance:

A true positive would be the program correctly not finding a bug (formerly true negative).

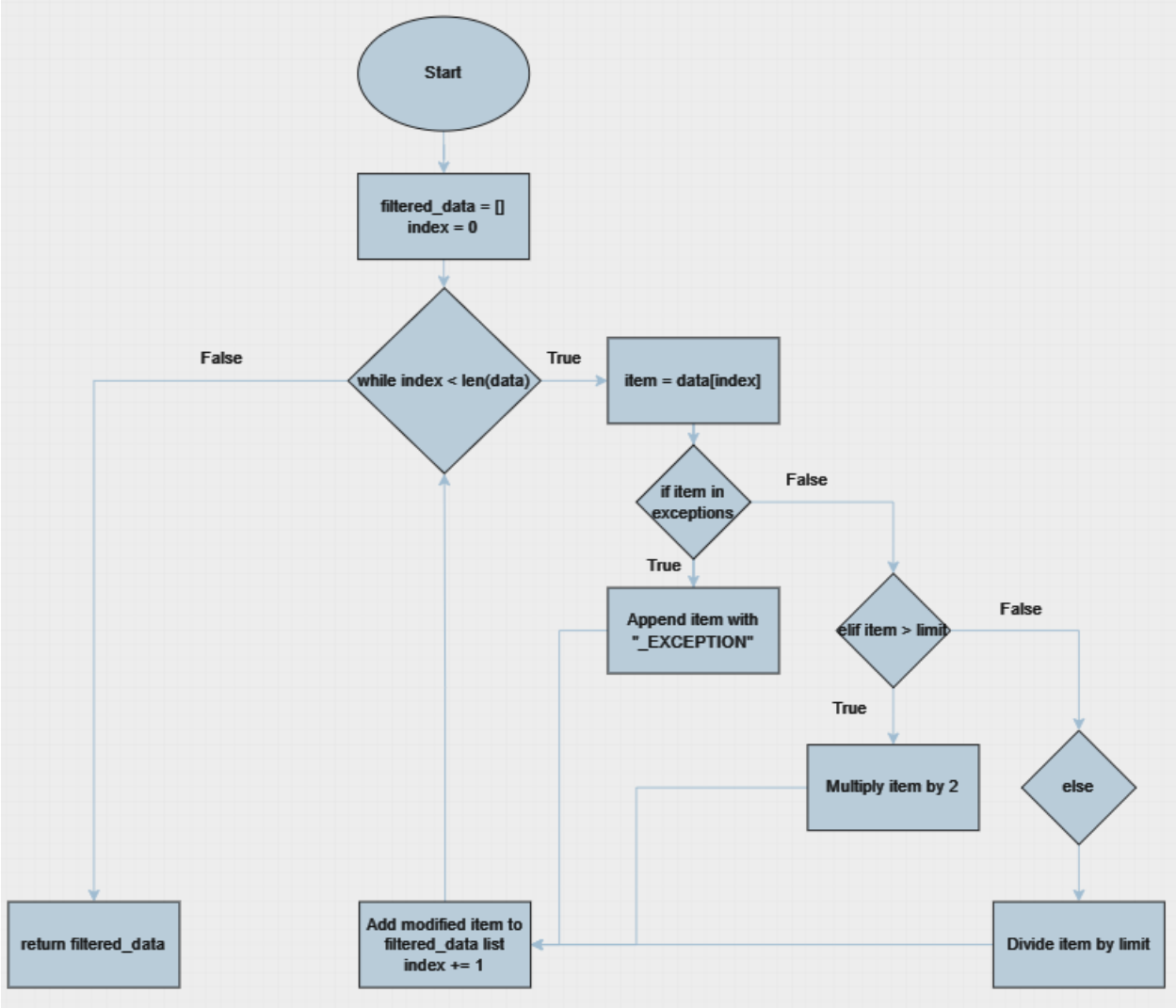
A true negative would be the program correctly finding a bug (formerly true positive).

A false positive would be the program reporting no bugs when there is in fact a bug (formerly false negative).

A false negative would be the program reporting a bug when there isn't one (formerly false positive).

As you can see, all of the definitions get flipped on their heads, as opposed to the original definitions I provided, wherein the word "positive" means finding a bug.

3.



To implement a “random testing” program to test the above code, I would generate a random number between 1 and 100 to represent the number of passes. Through each pass, I would randomly generate

- A data set, with a length between 0 and 500 consisting of integers between 0 and 1000
- A limit, between 0 and 1000
- A list of exceptions, between 0 and len(data) consisting of integers between 0 and 1000

I would then write a method to return the expected results of the function, and compare the actual results with the expected for every pass. If they’re not equal, I would output the data set and both results, as well as the pass number for the tester to examine.

4.

a) Test case 1:
data = {1}; limit = 5; exceptions = {1,2};
This test will run the while loop once. During that run, it will execute the “if item in exceptions” statement, modify the item to append “_EXCEPTION”, and return the filtered_data array, which will consist of the element {“1_EXCEPTION”}.
This test case will execute ⅓ of the branches for ~33.34% code coverage.

Test case 2:
data = {1,2}; limit = 5; exceptions = {1,3}
This test case will run the while loop twice. During the first loop, it will execute the “if item in exceptions” statement and modify data[0] to append “_EXCEPTION”. The second loop will execute the else statement, as data[0] is neither in the exceptions array nor greater than the limit. It will divide itself by the limit. Once the loop exists, it will return the filtered_data array, consisting of {“1_EXCEPTION”, 0.4}.
This test case will execute ⅔ of the branches for ~66.67% code coverage.

Test case 3:
data = {3,6}; limit = 4; exceptions = {0}
This test will run the while loop twice. During the first loop, it will execute the else statement, dividing data[0] by 4. During the second loop, the code will execute the “elif item > limit” statement, and multiply data[1] by 2. Once the loop exits, the program will return the filtered_data array consisting of {0.75, 12}.
This test case will execute ⅔ of the branches for ~66.67% code coverage.

Test case 4:
data = {3, 7, 16}; limit = 8; exceptions = {5, 6, 7}
This test will run the while loop 3 times. During the first loop, it will execute the else statement, modifying data[0] to be divided by 8. During the second loop, it will execute the “if item in exceptions” statement, and append “_EXCEPTION” to data[1]. During the third and final loop, it will execute the “elif item > limit” statement, multiplying data[2] by 2. Once the loop exits, it will return the filtered_data array, consisting of {0.375, “7_EXCEPTION”, 8}.
This test case will execute all 3 branches for 100% code coverage.

b)

```
def filterData(data, limit, exceptions):
filtered_data = []
index = 0
while index < len(data):
item = data[index]
if item in exceptions:
modified_item = item + "_EXCEPTION”
elif item == limit:
modified_item = item * 2
else:
modified_item = item / limit
filtered_data.append(modified_item)
index += 1
return filtered_data

def filterData(data, limit, exceptions):
filtered_data = []
index = 0
while index < len(data):
item = data[index]
if item in exceptions:
modified_item = item + "_EXCEPTION
elif item < limit:
modified_item = item * 2
else:
modified_item = item / limit
filtered_data.append(modified_item)
index += 1
return filtered_data

def filterData(data, limit, exceptions):
filtered_data = []
index = 0
while index < len(data):
item = data[index]
if item in exceptions:
modified_item = item + "_EXCEPTION
elif item > limit:
modified_item = item * 5
```

```
else:
modified_item = item / limit
filtered_data.append(modified_item)
index += 1
return filtered_data

def filterData(data, limit, exceptions):
filtered_data = []
index = 0
while index < len(data):
item = data[index]
if item in exceptions:
modified_item = item + "_EXCEPTION
elif item > limit:
modified_item = item * 2
else:
modified_item = item / (limit^2)
filtered_data.append(modified_item)
index += 1
return filtered_data

def filterData(data, limit, exceptions):
filtered_data = []
index = 0
while index < len(data):
item = data[index]
if item in exceptions:
modified_item = item + "_EXCEPTION
elif item > limit:
modified_item = item * limit
else:
modified_item = item / 2
filtered_data.append(modified_item)
index += 1
return filtered_data

def filterData(data, limit, exceptions):
filtered_data = []
index = 0
while index < len(data):
```

```
item = data[index]
if (item in exceptions) == False:
modified_item = item + "_EXCEPTION
elif item > limit:
modified_item = item * 2
else:
modified_item = item / limit
filtered_data.append(modified_item)
index += 1
return filtered_data
```

c)

```
Pass 0
1: ['1_EXCEPTION']
2: ['1_EXCEPTION']
3: ['1_EXCEPTION']
4: ['1_EXCEPTION']
5: ['1_EXCEPTION']
6: [0.2]

Original:
['1_EXCEPTION']

Pass 1
1: ['1_EXCEPTION', 0.4]
2: ['1_EXCEPTION', 4]
3: ['1_EXCEPTION', 0.4]
4: ['1_EXCEPTION', 0.2857142857142857]
5: ['1_EXCEPTION', 1.0]
6: [0.2, '2_EXCEPTION']

Original:
['1_EXCEPTION', 0.4]

Pass 2
1: [0.75, 1.5]
2: [6, 1.5]
3: [0.75, 30]
4: [0.5, 12]
5: [1.5, 24]
6: ['3_EXCEPTION', '6_EXCEPTION']

Original:
[0.75, 12]

Pass 3
1: [0.375, '7_EXCEPTION', 2.0]
2: [6, '7_EXCEPTION', 2.0]
3: [0.375, '7_EXCEPTION', 80]
4: [0.3, '7_EXCEPTION', 32]
5: [1.5, '7_EXCEPTION', 128]
6: ['3_EXCEPTION', 0.875, '16_EXCEPTION']

Original:
[0.375, '7_EXCEPTION', 32]
```

After running the test cases with the original method and the 6 mutated cases, I received this output. From this we can see the "kill count" of each of my test cases.

- Test case 1 killed 1 mutant
- Test case 2 killed 4 mutants
- Test case 3 killed 6 mutants
- Test case 4 killed 6 mutants

I would rank them 3, 4, 2, 1 with 3 being the best and 1 being the worst. Obviously I put 1 and 2 at the bottom because they killed fewer mutants than 3 and 4, and even thought 3 and 4 killed the same number of mutants I put 3 higher because the filtered_data[1] element in 4's test case are almost all '7_EXCEPTION', including the original. Compared to this, 3 had more unique and varied outputs.

d)

Looking at this code I know there are 3 possible outcomes:

- item in exceptions
- elif item > limit
- else (item not in exceptions and item <= limit)

It's important to note the if/elif/else structure to this code. If our limit is 3, our exceptions are 12 and 25, and the item is also 25, then the second case (elif item > limit) won't be executed even though 25 > 3. This is because 25 is in the exceptions, so that line would be executed instead. To perform branch, path, or statement analysis on this code, I would ensure that there is at least one element in the data set that is in the exceptions, one that is greater than the limit *but not* in exceptions, and one that is neither. This would trigger every branch, test every path, and execute every statement thanks to the simplicity of this program.

5.

a. To test this code, I set up a simple python program that would manually test 3 strings. When creating these strings, I ensured they would each have numbers, letters, and non-numeric characters (such as a space or semicolon). This was the result of these tests:

```
In: ABc123deF
Out: abC112233DEf
In: oK;;69
Out: Ok;;6699
In: 60 70 bYe ByE
Out: 6600 7700 ByE bYe
```

After analyzing the results of these tests and the source code, I've come to the conclusion that the bug is the line

```
elif char.isnumeric():
```

```
output_str += char * 2
```

This line duplicates numeric characters, so an input of “1” will return “11”.

b. Please note: the program I coded for this portion of this assignment is done in Java, despite the previous code being Python. I did the first part in Python to more precisely pinpoint the bug, however I’m more comfortable coding solutions in Java. The Java program produces a slightly different bug (different char multiplication) however the delta debugging works as intended. Both source codes are included in this assignment.

   My algorithm recursively searches for problematic characters in the given string and adds them to a list. It will split an input string into two halves and test them. If a half is problematic and its length is greater than 1, the method will call itself and pass the half in to be searched. If the length isn’t greater than 1, it means we’ve found the problematic char and it can be added to the list. This method is a private one called by the instructor of the class, so that the user cannot modify the problem char list. My program constructs 4 instances of the class, each with one of the provided input values. It then prints the results of each debug.