COSC3P95: Software Analysis & Testing

Assignment 1

Sneh Patel (6698484)

October 16, 2023

1. Explain the difference between "sound" and "complete" analysis in software analysis. Then, define what true positive, true negative, false positive, and false negative mean. How would these terms change if the goal of the analysis changes, particularly when "positive" means finding a bug, and then when "positive" means not finding a bug. **(10 pts)**

   Sound analysis: A type of analysis that does not produce any **false positives**. This is useful in detecting errors that may occur when the software is used for its intended purpose. Sound analysis produces few results, however its very accurate in the results in does produce.

   Complete Analysis: A type of analysis that finds all **true positives** (without missing any). While using this analysis technique not detect any issue, one can be certain that, that issue does not exist. A complete analysis is used to find all issues that may occur in a software without overlooking anything. However due to this, it may produce false positives.

   True Positive (TP): If a model predicts a case to be positive and it is positive. For example, if a model detect cancer in a patient and the patient does have cancer.

   True Negative (TN): If a model predicts a case to be negative and it is negative. For example, if a model does not detect cancer in a patient and the patient does not have cancer.

   False Positives (FP): If a model predicts a case to be positive and it actually is negative. For example, if a model detects cancer in a patient and the patient does not have cancer.

   False Negative (FN): If a model predicts a case to be negative and it actually is positive. For example, if a model does not detect cancer in a patient, and the patient does have cancer.

   Finding a Bug: If you were analyzing a software with the purpose of finding bugs. TP means, the analysis found a bug and it is a bug. TN means, the analysis did not find a bug and its not a bug. FP means, the analysis found a bug and it is not a bug. FN means, the analysis did not find a bug and there is a bug.

   Not Finding a bug If you were running a software with the intend of using for its purpose, and run into a bug. TP means software did not run into any bugs as there were none. TN means that the software ran into a bug and it was a bug. FP means that the software did not run into any bugs, while there were some bugs. FN means that the software ran into a bug when there were none.

2. Using your preferred programming language, implement a random test case generator for a sorting algorithm program that sorts integers in ascending order. The test case generator should be designed to produce arrays of integers with random lengths, and values for each sorting method.

   (a) Your submission should consist of:

      i. Source code files for the sorting algorithm and the random test case generator.

      ii. Explanation of how your method/approach works and a discussion of the results (for example, if and how the method was able to generate or find any bugs, etc.). You can also include bugs in your code and show your method is able to find the input values causing that.

iii. Comments within the code for better understanding of the code.

iv. Instructions for compiling and running your code.

v. Logs generated by the print statements, capturing both input array, output arrays for each run of the program.

vi. Logs for the random test executions, showing if the test was a pass and the output of the execution (e.g., exception, bug message, etc.).

For my approach I used a simple merge sort algorithm, the algorithm takes in an array and then returns a sorted array. For the random test generator, I created a function `random_test_gen()` that evaluates my sorting algorithm. The `random_test_gen()` function creates an array of size random (the max size is 10 and min size is 0) of random integers. Then I used Python's built-in sorting method as a ground truth to compare my sorting algorithm with it. Then I compared the output I obtained with my algorithm with the output the built-in method provides. If the two arrays match I print a test case pass statement, if it fails I show the user which test case it failed, the output obtained from the merge sort and the output obtained from Python's default sorting method. The merge sort I programmed for this question is programmed correctly, meaning that it passes each test case. However, if I were to change the ¡ sign into a ¿ on `line 33` the merge sort would sort the array in descending order. It would fail each test case except for the test cases where the array is of size 1 or less. To run the code execute the `question_2.py` file by using the command `python3 question_2.py`

(b) Provide a context-free grammar to generate all the possible test-cases

$S \rightarrow [A]$

$A \rightarrow \texttt{\#}, A | \epsilon$

$\texttt{\#} \rightarrow$ any integer

3. (a) For the following code, manually draw a control flow graph to represent its logic and structure. [1]

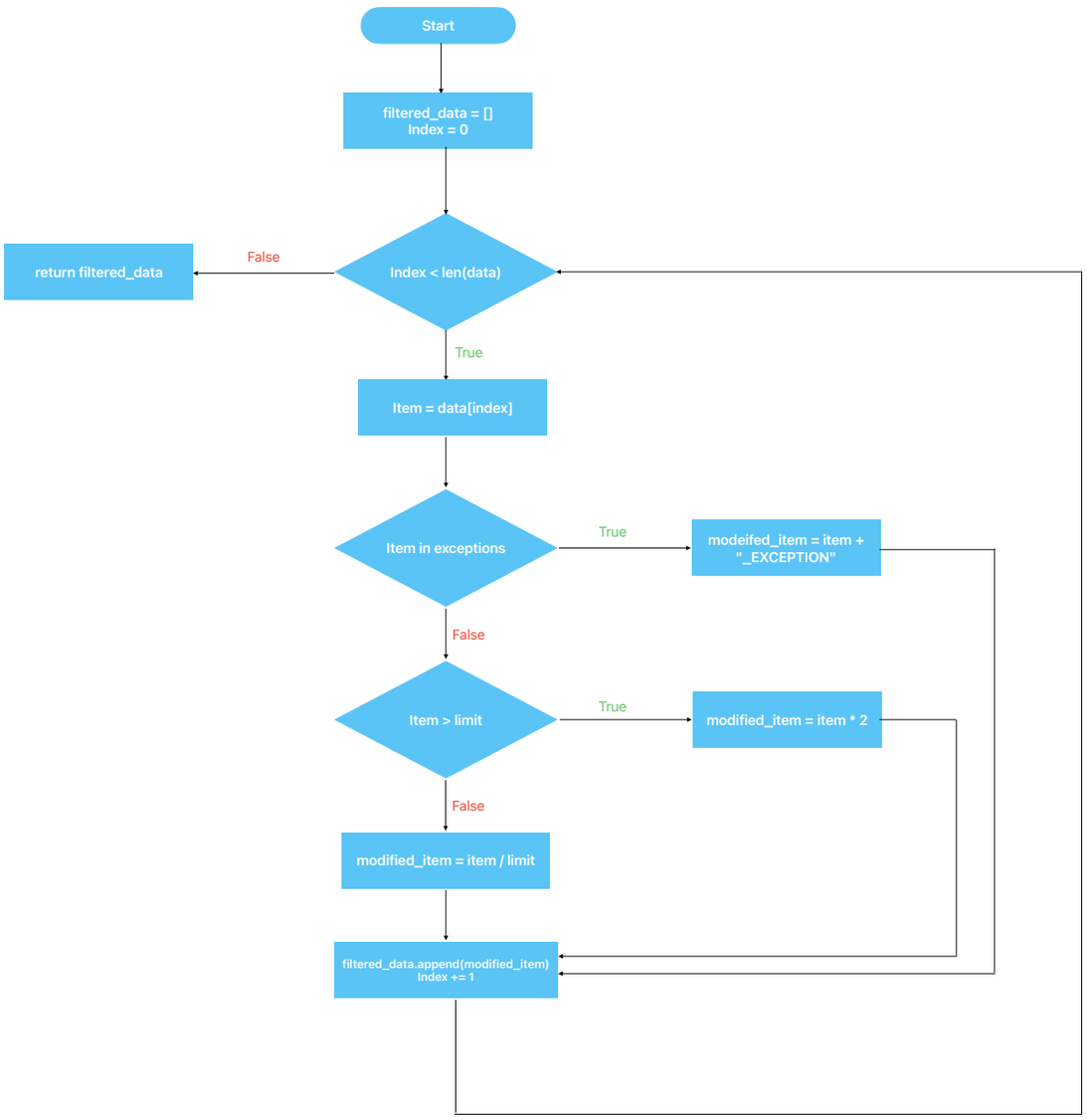

Figure 1: This is a flow chart for the code provided for this question

(b) Explain and provide detailed steps for "random testing" the above code. No need to run any code, just present the coding strategy or describe your testing method in detail. **(8 + 8 = 16 pts)**

To do a "random testing" create test cases with 3 inputs: data (random length with random integers), limit (an integer), and exceptions (an array of integers that are exceptions), the output would be a filtered_data array, then randomly pick a test case from the test cases and run the program with the input data from the test cases, after compare the output retrieved with the test cases output if they match then the test was a success else it failed, iterate this process for a x number of times (x ¿ 0). Things we should test for are:

A. data is empty and limits and exceptions exist

B. data is not empty and exceptions are empty.

C. in data every number is below the limit

[1] The code was missing quotes after the word EXCEPTION on line 7 also in python3+ you cannot add a string to an int you have to convert the int to a string

D. in data ever number is above the limit

4. (a) Develop 4 distinct test cases to test the above code, with code coverage ranging from 30% to 100%. For each test case calculate and mention its code coverage.

```python
def test_low_coverage_div():
    # This tests the division funcationality (else statement)
    data = [10, 20, 30, 40]
    limit = 100
    exception = []
    output = filterData(data, limit, exception)
    ground_truth = [10 / 100, 20 / 100, 30 / 100, 40 / 100]
    assert output == ground_truth
```

(a) Code coverage: 33%

```python
def test_low_coverage_mult():
    # This tests the multiplcation functaionality (item > limit)
    data = [10, 20, 30, 40]
    limit = 5
    exceptions = []
    output = filterData(data, limit, exceptions)
    ground_truth = [10 * 2, 20 * 2, 30 * 2, 40 * 2]
    assert output == ground_truth
```

(b) Code coverage: 33%

```python
def test_mid_coverage():
    # This tests the multiplication and division funcationality together
    data = [10, 20, 30, 40]
    limit = 20
    exception = []
    output = filterData(data, limit, exception)
    ground_truth = [10 / 20, 20 / 20, 30 * 2, 40 * 2]
    assert output == ground_truth
```

(c) Code coverage: 67%

```python
def test_all_coverage():
    data = [10, 20, 30, 40]
    limit = 20
    exception = [30]
    output = filterData(data, limit, exception)
    ground_truth = [10 / 20, 20 / 20, "30_EXCEPTION", 40 * 2]
    assert output == ground_truth
```

(d) Code coverage: 100%

Figure 2: Four distinct test cases with code coverage ranging from 30% to 100%

Figure 2 displays four different test cases that range from 30% to 100%. Figure 2a displays a test case that covers 33% of the code as all of the data elements are less than the limit only one part of the if-statement gets executed (the division part), and since there are 3 sections to the if-statement, it only covers 33% of the code. Figure 2b displays a test case that covers 33% of the code as all the data elements are greater than the limit only one part of the if-statement gets executed (the * 2 part), hence it's 33%. Figure 2c displays a test case that covers 67% of the code some of the data is greater than the limit and some is less than the limit. Due to this 2 parts of the if statements get executed, resulting in 67% of the code coverage. Figure 2d is the same as the code shown in figure 2c, however, there is an exception, therefore all the parts of the if-statements get run, resulting in 100% of coverage.[2]

(b) Generate 6 modified (mutated) versions of the above code.

The code for this section is found in file `question_4_b.py` to run the file use the command `python3 question_4_b.py`

(c) Assess the effectiveness of the test cases from part A by using mutation analysis in conjunction with the mutated codes from part B. Rank the test cases and explain your answer.

i. The first mutation changes the division and multiplication signs therefore every test case that multiplied the functionality changes to division and vice-versa. This mutation fails every test as it changes the most essential part of the code.

ii. The second mutation gets rid of the exception, meaning most of the test cases should still work the same way. To be specific only `test_all_coverage()` fails the rest pass.

iii. The third mutation gets rid of the multiplication part of the code, therefore the only test case that would work properly would be `test_low_coverage_div()` since that's the only test that does not test the multiplication.

[2]The code for this in the file `question_4_a.py` you can run it using the command `python3 question_4_a.py`

iv. The fourth mutation multiplies the data element each time regardless of the condition. `test_low_coverage_mult()` is the only function that would test that mutation (by itself) would pass since that test only checks the multiplication.

v. The fifth mutation divides the data element each time regardless of the condition. `test_low_coverage_div()` is the only function that would test that division (by itself) would pass since that test only checks the multiplication.

vi. The sixth mutation produces the same output as the second mutation, therefore again it passes all the test cases other than the `test_all_coverage()`

Rank: Sixth and the second are the best out of all the mutations since they pass all tests except for 1. The third, forth, and fifth mutations all rank second, as they all at least pass 1 test. The first mutation is the worst since it fails all the tests.

(d) Discuss how you would use path, branch, and statement static analysis to evaluate/analyze the above code. **(4 * 8 = 32 pts)**

i. Path Analysis: For specifically the code above I would create test cases that ensure that all parts of the if statements get executed

ii. Branch Analysis: For specifically the code above I would do have some sort of checkpoints for each part of the if-statement this lets me know what part of the if-statement gets run and which didn't

iii. Statement static analysis: For specifically the code above I would analyze the code manually going line by line and would try to assume what the code does per line.

5. (a) Identify the bug(s) in the code. You can either manually review the code (a form of static analysis) or run it with diverse input values (a form of manual random testing). If you cannot pinpoint the bug using these methods, you may utilize a random testing tool or implement a random test case generator in code. Please provide a detailed explanation of the bug, identify the line of code causing it, and describe your strategy for finding it

   The bug is on `line 7` when the code checks if the character is a number or not it multiplies the number by 2. Since the character is a sub-type of string when multiplied by 2 it creates a copy of itself and adds it to the `output_str` variable. Therefore the string `QweR12tY` becomes `qWEr1122Ty`, and since we want the numbers to remain the same with no change this would be considered a bug for us. I found this bug while doing a static analysis and executing the code with a string containing lower case letters, upper case letters, and numbers to confirm my initial hypothesis. Also to fix this bug change the 2 to a 1, or get rid of the * 2 together.

   (b) Implement Delta Debugging, in your preferred programming language to minimize the input string that reveals the bug. Test your Delta Debugging code for the following input values provided.

   i. "abcdefG1"

   ii. "CCDDEExy"

   iii. "1234567b"

   iv. "8665"

   Briefly explain your delta-debugging algorithm and its implementation and provide the source code in/with your assignment. **(4 + 12 = 16 pts)**

   The code for this question is in the file `delta_debugging.py`. Delta-debugging is a method used to reduce the size of an input to make it into a smaller input size to find a problem within the code. The function I created (`delta_debugging()` does exactly that. It first creates a subset of the original test case then it checks if there is a bug within that subset, if there is then that subset becomes that new subset, if not it, creates a new subset. After checking all subsets of that size the size of the subset decreases by half. If the size of the subsets decreases less than or equal to 1 we have reached the end. [3] The outputs for each test case according to my code:

   A. For "abcdefG1" the reduced test case is "G1".

   B. For "CCDDEExy" the reduced test case is "CCDDEExy" (means there were no bugs).

   C. For "1234567b" the reduced test case is "7B"

   D. For "8665" the reduced test case is "65"

[3] Please note that for each test case include a ground truth to compare the output of `processString`