SWE 30009 Software Testing & Reliability TP 2 2024

Title: Project Report

Name (Student ID): Joseph Linao (104556329)

Lab class: Wednesday / 10:30 AM to 11:30 AM / BA709

Due Date: Monday 14th October 2024 at 11:00 PM

Task 1:

Subtask 1.1: Effectiveness matrix

The effectiveness matrices, which are used to evaluate the quality and success of software testing methods like random testing and partition testing. The

- P-Measure:
  P-measure calculates the probability of detecting at least one failure when running a test case.

In random testing, the probability is based on how uniformly failure-causing inputs are distributed across an entire input space.	For partition testing, it focuses on the likelihood of detecting failures within each partition.
$P_r = 1 - (1 - \theta)^n$	$P_p = 1 - \prod_{i=1}^{k} (1 - \theta_i)^{n_i}$
This formula calculates the Pr of detecting at least one failure in random testing where: - Θ is the proportion of failure-causing inputs in the entire input domain - N is the number of test cases	This formula calculates the Pp of detecting at least one failure in partition testing where: - K is the number of partitions - Θi is the proportion of failure-causing inputs in the i-th position - Ni is the number of test cases allowed to the i-th position.
Example: Assume n=2 Θ = 2+3+5/ 100 + 200 + 250 = 5/550 Pr = 1 – (1 – 10/550)^2 Pr = 1 - (1-0.01818)^2 Pr = 0.96367	Example: Assume n = 3 Partition 1: 100/550 * 3 = 0.55 Partition 2: 200/550 * 3 = 1.09 Partition 3: 250/550 * 3 = 1.36 Pp = 1 – ((1 - 0.03)^1 * (1- 0.025)^1 * (1-0.008)^1) Pp = 1-(0.937074) Pp = 0.062926

- E-Measure:
  E-Measure estimates the number of failures detected by the test cases.

In random testing, it considers the entire input domain without distinguishing between partitions.	Partition testing, however, uses the E-measure to reflect how effectively the tests are distributed across different partitions, factoring in the number of failures in each partition.
$E_r = n\theta$	$E_p = \sum_{i=1}^{k} n_i\theta_i$
This formula is the expected number of failures in random testing.	This formula calculates the expected number of failures in partition testing.

- Er = is the expected number of failures in random testing - n is the number of test cases - Θ is the overall proportion of failure-causing inputs in the entire input domain.	- Ep is the expected number of failures in partition testing - k is the number of partitions - ni is the number of test cases allocated to partition i. - Θi is the failure rate of partition i.
Example: Assume n = 2 Θ = 2+3+5/ 100 + 200 + 250 = 5/550 Er = 2 * 5/550 Er = 0.01818181818	Example: Assume n = 3 Partition 1: 100/550 * 3 = 0.55 Partition 2: 200/550 * 3 = 1.09 Partition 3: 250/550 * 3 = 1.36 Ep = E1 + E2 + E3 Ep = 0.03 + 0.025 + 0.008 = 0.063

- F-Measure:
  Expected number of test cases to detect the first failure
- Proportional Sampling Strategy (PSS):

  Proportional Sampling Strategy is a simple method that can be applied to almost any partitioning scheme, with the given size ratios of the partitions.

Subtask 1.2: Metamorphoric Testing

Overview:

Metaphoric Testing is a testing technique for generating test cases and testing the untestable problems. A test oracle is a mechanism or procedure against which the computed outputs could be verified.

Motivation and intuition:

Metamorphic testing can be motivated by the need of test cases operating clearly without the need for a test oracle. It makes uses of an identified relation among multiple test cases via metamorphic relationships - necessary properties of the algorithm to be implemented, which involve multiple related inputs and their outputs.

Intuition behind metamorphic testing is that whether the relationship between the input and the output are consistent across numerous test cases.

Process of Metamorphic Testing:

1. Identify the Metamorphic Relationship:

   The first step is to identify suitable MRs for the software under test. These relations are based on the understanding of the software’s functionality and domain properties.

2. Generate test cases:

   Initial test cases are generated without the concern of knowing the exact output.

3. Follow-up tests

   Based on the MRs, new test cases are (follow-up test cases) are generated by altering the initial test case.

4. Execution of test cases

   Both the initial and the follow-up test cases are executed.

5. Verification of MRs

   Check whether the MR hold for the output of the initial and the follow-up test cases. If violated, a defect, may be present.

Metamorphic Relationships (MRS) Examples :

Example 1: (Testing a Sorting Algorithm)

- MR identification: a basic Mr for a sorting algorithm could be that if you shuffle the elements of the input list, the output (sorted list) should remain the same
- Test Case Generation: Generate a random list of numbers (initial test cases) and a shuffled version of this list (follow-up test cases)
- Verification: After sorting both lists, the outputs should be identical. If not, there is a flaw in the sorting algorithm.

Example 2: (Machine Learning Model)

- MR identification: for a regression model, an MR could be that multiplying all input features by a positive constant should result in the output being multiplied by the same constant.
- Test case generation: create an input dataset and a modified version where all features are scaled by a constant
- Verification: compare the outputs; they should be scaled consistently accordingly to the metamorphic testing. Deviations might indicate issues in the model or data processing.

Example 3: (Testing a File Compression Program)

- MR identification: a basic Metamorphic Relation for a file compression program could be that compressing a file and then decompressing it should result in the original file
- Test case generation: Take an initial file (test case), compress it using the program, and then decompress the command file (follow-up test case)
- Verification: after decompression, the output should be identical to the original file. If the decompressed file differs from the original file, there is a fault in the compression or decompression algorithm.

Application:

Metamorphic testing is particularly useful in domain such as:

- Machine Learning and AI: in AI algorithms, especially in machine learning, predicting the exact output is often impractical. MT can be used to verify the consistency and correctness of these models.
- Web services and APIs: MT can be testing web services and APIs, especially when the services are black-box, and the internal workings are unknown.
- Scientific Computing: in scientific applications where the results can complex and exact values are not known, MT helps in validating the functionality of the software.

Subtask 1.3: Mutation Testing

Overview

Mutation testing is a software testing technique used to evaluate the effectiveness of the automated tests that already exist in the system. In other words, the assumption is that you already have a suite of tests, and you want to know if they are effective, i.e., truly capable of detecting bugs and regressions.

Mutants

To do this, a mutation testing tool makes small modifications to the production code, generating a version of the code referred to as a mutant. Mutants can sightly mutated versions of a given program which is compiled slightly and goes through transformations rules via mutation operations.

Mutation Operators

Mutation operators define the rules for generating mutants. Each mutant operator introduces a specific type of change, such as altering logical operators, modifying arithmetic operations, or changing variable values.

Examples:

- Change of arithmetic operators
  - A + B becoming A * B
- Change of arithmetic variables
  - C = A + B becoming D = A + B
- Replacement of variables by constraints
  - A = A + B becoming A = A + 1
- Change of relational operators
  - (A+B >= A * B) becoming (A + B > A * B)
- Change of logical operators
  - (A + B >= A * B) && (A-B > A/B) becoming (A + B>= A* B) || (A - B> A/B) becoming (A + B >= A * B) || (A - B > A / B) where && means AND, || means OR
- Change of logical variables
  - X && Y becoming X && Z

- Replacement of logical variables by Boolean constraints
    - X && Y becoming X && true

How mutants killed

- A mutant is killed when the test cases detect the change introduced by the mutant and the cause the test to fail. If the mutant remains undetected (i.e., all tests pass even with the mutant in place), it is considered a live mutant, suggesting that the test suite might need some improvements.

Task 2:

1. Select program P for testing
   The selected program is a Python file called bubblesort.py that performs the following tasks:
    - Takes a list of numbers as a program
    - It repeatedly takes steps through the list, compares adjacent items, and swaps them if they are in the wrong order. The process is repeated until the list is sorted
    - The sorted list is returned or printed as output
2. Define Metamorphic Relations (MRs)

- MR1: Reversing the inputs
    - Sorting should be independent of the initial order of the input. Whether the input list is reversed or not, the final sorted output should be the same.
- MR2: Concatenation of sorted lists
    - Sorted a concatenated list of two sorted lists should yield the same result as merging the two sorted lists directly. Sorting should be consistent whether lists are sorted before or after concatenation.

3. Generate & prepare test cases
   MR1:

MTG	Description	Expected behaviour	Test case
1	Sort a normal list and its reverse	Sorting the original list and its reverse should yield the same sorted output	Sort [3, 1, 4, 1, 5] and compare with the sorted result of [5, 1, 4, 1, 3]
2	Sort a list with repeated elements and its reverse	Both the normal and reversed list should result in the same sorted output	Sort [5, 2, 2, 8] and compare with [8, 2, 2, 5]
3	Sort an already sorted list and its reverse	Sorting an already sorted list and its reverse should result in the same sorted output	Sort [1, 2, 3, 4] and compare with [4, 3, 2, 1]
4	Sort a list with negative and positive numbers and its reverse	Sorting the original list and its reverse should yield the same sorted output	Sort [-3, 0, 2, 1] and compare with [1, 2, 0, -3]
5	Sort a long list of random numbers and its reverse	Both the original and reversed list should result in the same sorted output	Sort [9, 7, 5, 3, 1] and compare with [1, 3, 5, 7, 9]

MR2:

MTG	Description	Expected behaviour	Test case

1	Concatenate two small, sorted lists and sort	Sorting two merged concatenated sorted lists	Sort the concatenation [1,3] and [2, 4] and compare with the merged result [1, 2, 3, 4]
2	Concatenate two large, sorted list and sort	Sorting two large concatenated sorted lists	Sort the concatenation [1, 3, 5, 7] and [2, 4, 6, 8] and compare with the merged result [1, 2, 3, 4, 5, 6, 7, 8]
3	Concatenate two sorted lists with repeated elements and sort	Sorting two concatenated sorted lists with repeated elements	Sort the concatenation [2, 2, 5] and [5, 7, 7] and compare with the merged result [2, ,2, 5, 5, 7, 7]
4	Concatenate two lists with negative and positive numbers, sort them, and compare	Sorting two concatenated sorted list with positive and negative numbers	Sort the concatenation of [-5, -3, 1] and [2, 4, 6] and compare with the merged result [-5, -3, 1, 2, 4, 6]
5	Concatenate one sorted and one unsorted list and sort	Sorting the concatenation of a sorted and unsorted lists	Sort the concatenation of [1, 3, 5] and [7, 2, 6] and compare with the result of merging the sorted versions [1, 3, 5] and [2, 6, 7]

4. Generate & evaluate mutants

Mutant ID	Description	Original code	Changed code	Mutation operator
M1	Changed array index from i to j	Tmp=array[i]	tmp=array[j]	Change of arithmetic variables
M2	Changed array index from i to k	Tmp=array[i]	tmp=array[k]	Change of arithmetic variables
M3	Changed array index from i to 0	Tmp=array[i]	tmp=array[0]	Change of arithmetic variables
M4	Changed array index from i to 1	Tmp=array[i]	tmp=array[1]	Change of arithmetic variables
M5	Modified assignment from array[i] to array[j]	array[i] = array[j]	array[j]=array[j]	Replacement of logical variables by constraints
M6	Modified assignment from array[i] to array[k]	array[i] = array[j]	array[k]=array[j]	Replacement of logical variables by constraints
M7	Modified the array[j] to array[1]	array[j] = tmp	array[1]=tmp	Change of arithmetic variables
M8	Modified the array[j] to array[0]	array[j] = tmp	array[0]=tmp	Change of arithmetic variables
M9	Modified the array[j] to array[k]	array[j] = tmp	array[k]=tmp	Change of arithmetic variables
M10	Modified the array[j] to array[i]	array[j] = tmp	array[i]=tmp	Change of arithmetic variables
M11	Altered the length by addition	n = len(array)-1	n=len(array)+1	Change of arithmetic operators
M12	Altered the length by multiplication	n = len(array)-1	n=len(array)*1	Change of arithmetic operators
M13	Altered the length by division	n = len(array)-1	n=len(array)//1	Change of arithmetic operators
M14	Altered the length by modulus	n = len(array)-1	n=len(array)%1	Change of arithmetic variables

M15	Modified conditionals from array[j] to array[k]	if array[j] > array[j+1]:	if array[k]>array[j+1]:	Change of arithmetic variables
M16	Modified conditionals from array[j] to array[1]	if array[j] > array[j+1]:	if array[1]>array[j+1]:	Change of arithmetic variables
M17	Modified conditionals from array[j] to array[i]	if array[j] > array[j+1]:	if array[i]>array[j+1]:	Change of arithmetic variables
M18	Modified conditionals from array[j] to array[0]	if array[j] > array[j+1]:	if array[0]>array[j+1]:	Change of arithmetic variables
M19	Modified conditionals from operator > to operator !=	if array[j] > array[j+1]:	if array[j]!=array[j+1]:	Change of relational operators
M20	Within the swap function, j is subtracted from 1	swap(array, j+1, j)	swap(array,j-1,j)	Change of arithmetic operators
M21	Within the swap function, j is multiplied from 1	swap(array, j+1, j)	swap(array,j*1,j)	Change of arithmetic operators
M22	Within the swap function, j is divided from 1	swap(array, j+1, j)	swap(array,j//1,j)	Change of arithmetic operators
M23	Within the swap function, j is modulus from 1	swap(array, j+1, j)	swap(array,j%1,j)	Change of arithmetic operators
M24	Negation of the condition that satisfies array[j] greater than array[j+1]	if array[j] > array[j+1]:	if not array[j]>array[j+1]:	Change of logical operators
M25	Always-true statement of the condition that satisfies array[j] greater than array[j+1]	if array[j] > array[j+1]:	if True or array[j]>array[j+1]:	Replacement of logical variables by Boolean constraints
M26	Modified the if __name__ == "__main__": condition by adding false	if __name__ == "__main__":	if False and __name__=="__main__":	Replacement of logical variables by Boolean constraints
M27	Altered the negative sign to plus	array = [17, 9, 13, 8, 7, -5, 6, 11, 3, 4, 1, 2]	array=[17,9,13,8,7,+5,6,11,3,4,1,2]	Change of arithmetic operators
M28	Altered the negative sign to multiplication	array = [17, 9, 13, 8, 7, -5, 6, 11, 3, 4, 1, 2]	array=[17,9,13,8,7,*5,6,11,3,4,1,2]	Change of arithmetic operators
M29	Changed from addition to division in array[j+1]	if array[j] > array[j+1]:	if array[j]>array[j//1]:	Change of arithmetic operators
M30	Changed the operators from > to >= (greater than or equal)	if array[j] > array[j+1]:	if array[j]>=array[j+1]:	Change of relational operators

5. Verify MRs against relevant Metaphoric Groups (MGs) & their outputs

Metamorphic Relation 1: Reversing the inputs

Mutant ID	MT1:	MT2	MTG 3	MTG 4	MTG 5
M1	Survived	Survived	Survived	Survived	Survived
M2	Killed	Killed	Killed	Killed	Killed
M3	Survived	Survived	Survived	Survived	Survived
M4	Survived	Survived	Survived	Survived	Survived
M5	Survived	Survived	Survived	Survived	Survived
M6	Killed	Killed	Killed	Killed	Killed
M7	Survived	Survived	Survived	Survived	Survived

M8	Survived	Survived	Survived	Survived	Survived
M9	Killed	Killed	Killed	Killed	Killed
M10	Survived	Survived	Survived	Survived	Survived
M11	Killed	Killed	Killed	Killed	Killed
M12	Killed	Killed	Killed	Killed	Killed
M13	Killed	Killed	Killed	Killed	Killed
M14	Survived	Survived	Survived	Survived	Survived
M15	Killed	Killed	Killed	Killed	Killed
M16	Survived	Survived	Survived	Survived	Survived
M17	Survived	Survived	Survived	Survived	Survived
M18	Survived	Survived	Survived	Survived	Survived
M19	Survived	Survived	Survived	Survived	Survived
M20	Survived	Survived	Survived	Survived	Survived
M21	Survived	Survived	Survived	Survived	Survived
M22	Survived	Survived	Survived	Survived	Survived
M23	Survived	Survived	Survived	Survived	Survived
M24	Survived	Survived	Survived	Survived	Survived
M25	Survived	Survived	Survived	Survived	Survived
M26	Survived	Survived	Survived	Survived	Survived
M27	Survived	Survived	Survived	Survived	Survived
M28	Survived	Survived	Survived	Survived	Survived
M29	Survived	Survived	Survived	Survived	Survived
M30	Survived	Survived	Survived	Survived	Survived

Metamorphic Relation 2: Concatenation of sorted lists

Mutant ID	MT1:	MT2	MTG 3	MTG 4	MTG 5
M1	Survived	Survived	Survived	Survived	Survived
M2	Killed	Killed	Survived	Survived	Killed
M3	Survived	Survived	Survived	Survived	Survived
M4	Survived	Survived	Survived	Survived	Survived
M5	Survived	Survived	Survived	Survived	Survived
M6	Killed	Killed	Survived	Survived	Killed
M7	Survived	Survived	Survived	Survived	Survived
M8	Survived	Survived	Survived	Survived	Survived
M9	Killed	Killed	Survived	Survived	Killed
M10	Survived	Survived	Survived	Survived	Survived
M11	Killed	Killed	Killed	Killed	Killed
M12	Killed	Killed	Killed	Killed	Killed
M13	Killed	Killed	Killed	Killed	Killed
M14	Survived	Survived	Survived	Survived	Survived
M15	Killed	Killed	Killed	Killed	Killed
M16	Survived	Survived	Survived	Survived	Survived
M17	Survived	Survived	Survived	Survived	Survived
M18	Survived	Survived	Survived	Survived	Survived
M19	Survived	Survived	Survived	Survived	Survived
M20	Survived	Survived	Survived	Survived	Survived
M21	Survived	Survived	Survived	Survived	Survived
M22	Survived	Survived	Survived	Survived	Survived

M23	Survived	Survived	Survived	Survived	Survived
M24	Survived	Survived	Survived	Survived	Survived
M25	Survived	Survived	Survived	Survived	Survived
M26	Survived	Survived	Survived	Survived	Survived
M27	Survived	Survived	Survived	Survived	Survived
M28	Survived	Survived	Survived	Survived	Survived
M29	Survived	Survived	Survived	Survived	Survived
M30	Survived	Survived	Survived	Survived	Survived

6. Effectiveness

   To assess the effectiveness and measure of these test cases, a mutation score is used to represent the percentage of mutants that were killed by the test suite:

   Mutation score = (number of killed mutants /total number of mutants killed or surviving) * 100

MR1:

Number of mutants killed: 35
Total number of mutants killed or surviving: 150
Mutation score = (35 / 150) * 100
= 23.33

MR2:

Number of mutants killed: 26
Total number of mutants killed or surviving: 150
Mutation score = (26 / 150) * 100
= 17.33

Mutation Score	MR1	MR2
	23.33%	17.33%

Conclusion: the first MR (with a mutation score of 23.33%) is better than second one (with a mutation score of 13.33%) because it detects more faults, suggesting that the first MR is more thorough and effective in killing mutants.

7. Test case Execution

```
C:\Users\cucum\Downloads\SWE30009>python bubblesort.py
[-5, 1, 2, 3, 4, 6, 7, 8, 9, 11, 13, 17]
```

```
============================ test session starts =============================
collecting ... collected 10 items

test_bubblesort.py::TestBubbleSort::test_mtg10_concatenate_sorted_unsorted_list PASSED [ 10%]
test_bubblesort.py::TestBubbleSort::test_mtg1_sort_normal_and_reverse PASSED [ 20%]
test_bubblesort.py::TestBubbleSort::test_mtg2_sort_repeated_elements_and_reverse PASSED [ 30%]
test_bubblesort.py::TestBubbleSort::test_mtg3_sort_already_sorted_and_reverse PASSED [ 40%]
test_bubblesort.py::TestBubbleSort::test_mtg4__sort_negative_positive_and_reverse PASSED [ 50%]
test_bubblesort.py::TestBubbleSort::test_mtg5_sort_long_random_list_and_reverse PASSED [ 60%]
test_bubblesort.py::TestBubbleSort::test_mtg6_concatenate_two_small_sorted_lists PASSED [ 70%]
test_bubblesort.py::TestBubbleSort::test_mtg7_concatenate_two_large_sorted_lists PASSED [ 80%]
test_bubblesort.py::TestBubbleSort::test_mtg8_concatenate_repeated_elements_and_sort PASSED [ 90%]
test_bubblesort.py::TestBubbleSort::test_mtg9_concatenate_negative_positive_lists PASSED [100%]

============================ 10 passed in 0.02s ==============================
```

References:

https://medium.com/@mailtodevens/metamorphic-testing-a-new-horizon-in-software-testing-6fdec595dba8

https://github.com/ztgu/sorting_algorithms_py/blob/master/insertionsort.py

https://softengbook.org/articles/mutation-testing

https://i.cs.hku.hk/~tse/Papers/1990s/fteffTR.pdf

https://www.techtarget.com/searchitoperations/definition/mutation-testing#:~:text=The%20mutation%20score%20is%20the,of%20mutants%2C%20multiplied%20by%20100.

F. T. Chan, T. Y. Chen, S. C. Cheung, M. F. Lau and S. M. Yiu, Application of Metamorphic Testing in Numerical Analysis, Proceedings of the IASTED International Conference on Software Engineering, 191-197, 1998.

S. Segura, G. Fraser, A. B. Sanchez and A. Ruiz-Cortes, A Survey on Metamorphic Testing, IEEE Transactions on Software Engineering, Vol. 42(9), 805-924, 2016.

Y. Jia and M. Harman, “An Analysis and Survey of the Development of Mutation Testing”, IEEE Transactions on Software Engineering, Vol. 37(5), 649-678, 2011.

Test code obtained by GitHub:
https://github.com/ztgu/sorting_algorithms_py/blob/master/bubblesort.py