Ta Quang Tung - 104222196

SWE30009 - Project Report

Task 1 - Random testing

Subtask 1.1 - Present understanding of random testing

Random testing is a black-box testing methodology in which test cases are chosen randomly and independently from the input domain without any guiding information. It can be used when the input domain is too large for exhaustive testing.

The test cases can be chosen randomly with either **uniform** or **operational distribution**. With uniform distribution, each input in the input domain has an equal chance of being selected as a test case. With operational distribution (or profile), an input has a higher chance of being selected if it is more likely to occur in the real world. For example, consider a program that calculates the recommended calorie intake for a person based on their age (from 1 to 100). Using uniform distribution, each age from 1 to 100 has an equal chance of being chosen as a test case. However, using operational distribution, the tester may notice that most people using the program are between 20 and 50 and thus give higher probabilities to inputs in this range, making them more likely to be selected for testing.

The random testing process consists of the following steps:

1. Determine the input domain.
2. Randomly select inputs from the domain to form the test set using either uniform or operational distribution. This can be done programmatically using a random number generator and assigning the resulting number to a particular input.
3. Test the program using the test set.
4. Compare the test results with the specifications to identify any failures.
5. Locate the error and fix the program if any failures are found.

Some applications of random testing include:

1. Black-box testing of programs with input domains too large for exhaustive testing.
2. Estimating software reliability.
3. Testing how a program responds to random inputs that may have not been anticipated by other testing methods, especially if the other methods introduce bias.
4. Designing some initial test cases to complement another testing method, such as metamorphic testing.

Subtask 1.2 - Apply random testing methodology

The first step in random testing is to determine the input domain. From the program specifications, one can deduce that ***the input domain consists of all non-empty lists of integers that may contain duplicated numbers***. Non-empty means that each list can contain from 1 up to potentially millions or billions of numbers. Each of these numbers can theoretically range from negative to positive infinity, but realistically they are constrained by the integer data type of the chosen programming language.

The next step is to generate the test cases. For this problem, uniform distribution is used because every list in the input domain has an equal chance of being used. A JavaScript program will be written to generate the test cases as seen below. It takes the number of test cases to generate as input and creates that many cases. The program first picks a random size for the test case and then fills it with random numbers. To make it easier to show the output in this report, the array size is limited to 15 and the numbers range between -100 and 100.

```
function generateTestCases(numCases) {
    const maxLength = 15;
    const minInt = -100;
    const maxInt = 101; // due to how Math.random() works, 101 will not be included.

    const testCases = [];
    for (let i = 0; i < numCases; i++) {
        const testCase = [];
        const arrayLength = Math.ceil(Math.random() * maxLength);

        for (let j = 0; j < arrayLength; j++) {
            const number = minInt + Math.floor((maxInt - minInt) * Math.random());
            testCase.push(number);
        }

        testCases.push(testCase);
    }

    return testCases;
}

console.log(generateTestCases(2));
```

Fig. 1: Random test case generator.

The program outputs the following 2 test cases:

Test case #1 (10 numbers, positive and negative integers included, no duplicates):

- Input: [4, -80, 29, -21, 86, 74, -88, -66, -34, -69]
- Expected output: [-88, -80, -69, -66, -34, -21, 4, 29, 74, 86]

Test case #2 (13 numbers, positive and negative integers included, with duplicates):

- Input: [95, -65, 57, 32, -91, -43, 61, 75, 90, -35, 43, -99, -99]
- Expected output: [-99, -99, -91, -65, -43, -35, 32, 43, 57, 61, 75, 90, 95]

Task 2 - Metamorphic testing

Subtask 2.1 - Present understanding of metamorphic testing

A program is considered **testable** if the output for any given input is verifiable in practice. The procedure through which an output is verified is called the **test oracle**. Consider a quadratic equation solver that calculates roots x_0 and x_1. This program is testable because

one can easily substitute the outputs into the equation to verify their correctness. In this case, the test oracle is the act of substitution.

In contrast, a program is considered **untestable** (or non-testable) if *some* of its outputs cannot be verified in practice. One example is a program that calculates the sine of a number x. If x is not a special value like 0 or 90 (degrees), it is very difficult to verify whether the output is correct or not.

A program is said to have the **test oracle problem** if it has no test oracle or it is too computationally expensive to apply one. By definition, untestable programs have the test oracle problem.

Metamorphic testing (MT) is a strategy that leverages the properties of the program under test to generate test cases. Its motivation is to test untestable programs where independent pairs of input and output are not sufficient to reveal errors. The intuition behind MT is that some sets of inputs and outputs are related according to some intrinsic properties of the problem, and the tester can check if the program correctly maintains these properties.

The program properties that MT is interested in are called **metamorphic relations (MRs)**. They are necessary properties of the implemented algorithm and must involve multiple inputs and outputs. A problem can have many MRs with varying levels of effectiveness. In the sine program above, one MR is that sin(x) = sin(x + 360), and one can verify the MR by executing the program with 10 and 370 to see if the outputs are equal.

The process of metamorphic testing consists of the following steps:

1. Design source inputs (SIs) with some test selection strategies and run the program with them to obtain the source outputs (SOs).
2. Identify the metamorphic relations of the problem.
3. Create and execute follow-up inputs (FIs) from the source test cases based on the MRs.
4. Check if the follow-up outputs (FOs) violate the MRs. If they do, the program is faulty and must be fixed.

Some applications of metamorphic testing include:

1. **Compilers**: Metamorphic testing found multiple errors in the popular GCC and LLVM compilers for the C programming language.
2. **Machine learning applications**: Accenture and Weka.
3. **Simulation platforms**: Facebook's Web-Enabled Simulation and the US Department of Energy's epidemiological simulation.
4. **Graphic drivers**: GraphicsFuzz used MT to test the drivers of Android phones.

Subtask 2.2 - Propose 2 MRs to test a program

MR1: Permutation Invariance

- **Description**: If FI is a permutation of SI, then FO = SO.
- **Intuition**: The initial order of the numbers does not matter because after sorting, they will always end up in the same sorted order.

- **How this MR works**:
    - Create a source input and execute the program with it.
    - Shuffle the source input to obtain a permutation.
    - Execute the program again with this permutation.
    - Verify that the outputs of the source list and its permutation are equal.
- **Example metamorphic group (using case #1 of subtask 1.2 as SI)**:
    - SI [4, -80, 29, -21, 86, 74, -88, -66, -34, -69]
    - FI [-80, 29, 4, -69, -21, -34, 86, 74, -66, -88]
    - SO [-88, -80, -69, -66, -34, -21, 4, 29, 74, 86]
    - FO [-88, -80, -69, -66, -34, -21, 4, 29, 74, 86]

MR2: Addition of a Constant

- **Description**: If FI is constructed by adding each number in SI with an integer constant C, then FO should be SO with C added to each number.
- **Intuition**: If a and b are integers and a > b, then a + C > b + C. The same holds for a < b and a = b. Essentially, when a constant C is added to all numbers in a list, their sorted order in the new list is the same as their sorted order in the old list.
- **How this MR works**:
    - Create a source input and execute the program with it.
    - Choose an integer constant C and add it to all the numbers in the source input to obtain the follow-up input.
    - Execute the program again with the follow-up input.
    - Verify that the follow-up output is the source output with C added to each number.
- **Example metamorphic group (using case #2 of subtask 1.2 as SI)**:
    - Constant C = 10
    - SI [95, -65, 57, 32, -91, -43, 61, 75, 90, -35, 43, -99, -99]
    - FI [105, -55, 67, 42, -81, -33, 71, 85, 100, -25, 53, -89, -89]
    - SO [-99, -99, -91, -65, -43, -35, 32, 43, 57, 61, 75, 90, 95]
    - FO [-89, -89, -81, -55, -33, -25, 42, 53, 67, 71, 85, 100, 105]

Subtask 2.3 - Compare the advantages and disadvantages of random testing and metamorphic testing

The advantages (green) and disadvantages (red) of random testing vs. metamorphic testing are summarized in the following table:

Random testing	Metamorphic testing
▲ Much easier to select test cases. Test cases can be selected independently and randomly.	▼ More difficult to generate test cases because the tester needs to design initial test cases and identify MRs. Identifying MRs can be very challenging for some problems.
▼ Test case generation is not guided by any problem-specific information.	▲ Test case generation is guided by the problem's intrinsic properties (MRs).

▲ Test case selection is unbiased and may include unanticipated edge cases.	▼ Test case selection may be biased depending on the metamorphic relations.
▲ Executing a test case once and comparing its output to the specifications is enough to detect faults.	▼ The program must be executed multiple times with the source and follow-up test cases. The tester then has to verify if the MRs have been violated.
▼ The outputs of the test cases can only be verified if the program has a test oracle.	▲ Works even if the program does not have a test oracle.
▼ Only useful for simple or non-critical applications.	▲ Useful for highly complex systems.

Table 1: Advantages and disadvantages of random testing and metamorphic testing.

Task 3 - Test a program

Program information

- **Name**: Substitution Cipher
- **GitHub repository**: https://github.com/BrianHook1183/Decoder-Ring
- **Programming language**: JavaScript
- **Source file**: All of the code relevant to this program is contained in a single file on https://github.com/BrianHook1183/Decoder-Ring/blob/main/src/substitution.js

Description

The program description can be found on its GitHub README. However, it is reiterated here for clarity.

This program implements a simple substitution cipher that can encode or decode a string of text. A substitution cipher is a cipher that requires two alphabets: a standard alphabet (assumed to be the English alphabet with letters a-z) and a substitution alphabet. The substitution alphabet has the same number of characters as the standard alphabet but the characters are randomized. During encoding, each standard character in the input string is replaced by the character at its position in the substitution alphabet. Decoding does the reverse, replacing each substituted character with a character at its position in the standard alphabet.

The image below shows an example standard alphabet (top) and substitution alphabet (bottom).

Fig. 2: Example alphabets.

Using these alphabets for encoding, the letter A becomes X, B becomes O, etc. On the other hand, for decoding, X becomes A, O becomes B, etc. The string “THIS IS A SECRET” will be encoded into “JRUD UD X DMYHMJ”.

Specifications

The program specifications can be found on its GitHub README. However, it is reiterated here for clarity.

The program consists of a `substitution` function that takes 3 inputs:

- `input`: A string of text to encode or decode. The string can include spaces, letters, digits, and special characters.
- `alphabet`: A string of exactly 26 unique (non-repeating) characters that represent the substitution alphabet. This alphabet can include special characters in addition to letters. The standard alphabet is assumed to be the English alphabet (a-z).
- `encode`: A boolean value (default is true). If true, the function will encode the message with the alphabet. If false, the function will decode the message with the alphabet.

Each test case for the program will thus be a tuple of 3 values (`input`, `alphabet`, `encode`).

The output of the function should be:

- `false` if the `alphabet` does not have exactly 26 characters or has a repeating character.
- The encoded/decoded string if the provided `alphabet` is valid. The output string should maintain spaces throughout and convert uppercase letters to lowercase. Because the standard alphabet is assumed to be the 26-letter English alphabet, the encoding process should only encode a-z characters. In contrast, the decoding process should only decode the characters that appear in the substitution alphabet.

One detail that the specifications do not mention is how the program should handle characters that do not appear in the standard alphabet (during encoding) or the substitution alphabet (during decoding). Typical implementations of this algorithm either keep these characters unchanged or remove them altogether. Here, it is assumed that either implementation is valid.

Metamorphic relations

MR1: Permutation of Input String

- **Description**: If FI is constructed from SI by permutating the characters of `input` while keeping `alphabet` and `encode`, then FO is a permutation of SO.
- **Intuition**: If a substitution cipher is applied to a permutation of the input text, the result should correspond to the permutation of the original encoded/decoded text.
- **Example**: If `"HELLO"` encodes into `"dqxxp"`, then `"OLLEH"` must yield `"pxxqd”`, which is a permutation of `"dqxxp"`. Similarly, if `“DQXXP”` decodes into `”hello”`, `”PXXQD”` must yield `”olleh”`.

MR2: Self-concatenation

- **Description**: If FI is constructed from SI by concatenating `input` with itself n times while keeping `alphabet` and `encode`, then FO will be SO concatenated with itself that many times.
- **Intuition**: With the same substitution alphabet, a substitution cipher always encodes or decodes a string in the same way. Therefore, if that string is duplicated (concatenated) multiple times, its output should repeat that many times.
- **Example**: If the input string `"HELLO"` encodes into `"dqxxp"`, then the follow-up input `"HELLOHELLOHELLO"` (concatenating `"HELLO"` with itself 2 more times) will encrypt into `"dqxxpdqxxpdqxxp"`.

MR3: Encode-Decode Symmetry

- **Description**: If a string `I` is encoded (or decoded) into a string `O` using the substitution alphabet `A`, then the string `O` must be decoded (or encoded) into the *lowercase* version of `I` using the same alphabet.
- **Intuition**: Using a substitution cipher, each letter in the standard alphabet maps to exactly one letter in the substitution alphabet, and vice versa. Therefore, we can easily encode and decode between the two alphabets.
- **Condition**: This relation only holds if all characters in `I` (case-insensitive, excluding spaces) appear in the standard alphabet for encoding or the substitution alphabet for decoding. This is because the specifications do not specify how unknown characters should be handled. If the implementation removes unknown characters, one cannot encode or decode the output back into the original input. As an example, if the string `"hello +"` is encoded into `"rmwwl "` without the `"+"`, the reverse process can never get back the original string.
- **Example**: If the input string `"HELLO"` encodes into `"rmwwl"` with the alphabet `"xoyqmcgrukswaflnthdjpzibev"`, then `"rmwwl"` must decode into `"hello"`.

Mutant generation

A total of 23 non-equivalent mutants were manually generated using 7 different mutation operators:

1. Logical operator replacement: 2
2. Relational operator replacement: 5
3. Constant replacement: 2
4. Wrong return: 2
5. Wrong argument to function: 5
6. Array name replacement: 4
7. Wrong order of conditional ternary operator: 3

No.	Mutation operator	Line	Original	Mutated
1	Logical operator replacement	4	`if (!alphabet || ...`	`if ( alphabet || ...`
2	Logical operator replacement	4	`if (!alphabet || ...`	`if (!alphabet && ...`
3	Relational operator replacement	4	`... alphabet.length !== 26 ...`	`... alphabet.length === 26 ...`
4	Constant replacement	4	`... alphabet.length !== 26 ...`	`... alphabet.length !== 25 ...`
5	Wrong return	4	`... return false;`	`... return true;`
6	Constant replacement	7	`"abcdefghijklmnopqrstuvwxyz"`	`"abcdefghijklmmopqrstuvwxyz"`
7	Wrong argument to function	7	`... .split("");`	`... .split(" ");`
8	Wrong argument to function	8	`... .split("");`	`... .split(" ");`
9	Wrong argument to function	9	`... .split("");`	`... .split(" ");`
10	Relational operator replacement	13	`... self.indexOf(item) === index`	`... self.indexOf(item) !== index`
11	Relational operator replacement	15	`... onlyUniqueChars.length !== alphabet.length ...`	`... onlyUniqueChars.length === alphabet.length ...`
12	Wrong return	15	`... return false;`	`... return true;`
13	Array name replacement	20	`... realAlphabetArray.indexOf(char);`	`... subAlphabetArray.indexOf(char);`
14	Array name replacement	21	`... subAlphabetArray[charIndex];`	`... realAlphabetArray[charIndex];`
15	Relational operator replacement	26	`char === " " ...`	`char !== " " ...`

16	Wrong order of conditional ternary operator	26	`... ? result.push(" ") : encode(char);`	`... ? encode(char) : result.push(" ");`
17	Wrong argument to function	28	`return result.join("");`	`return result.join(" ");`
18	Array name replacement	34	`... subAlphabetArray.indexOf(char);`	`... realAlphabetArray.indexOf(char);`
19	Array name replacement	35	`... realAlphabetArray[charIndex];`	`... subAlphabetArray[charIndex];`
20	Relational operator replacement	40	`char === " " ...`	`char !== " " ...`
21	Wrong order of conditional ternary operator	40	`... ? result.push(" ") : decode(char);`	`… ? decode(char) : result.push(" ");`
22	Wrong argument to function	42	`return result.join("");`	`return result.join(" ");`
23	Wrong order of conditional ternary operator	46	`return encode ? encodeMessage() : decodeMessage();`	`return encode ? decodeMessage() : encodeMessage();`

Table 2: Generated mutants.

Testing procedure

Test case design

Five test cases were created to serve as the source inputs of the MRs. They are denoted `si1`, `si2`, …, respectively. These test cases were manually designed to cover a wide range of **valid** input scenarios. For each component of the test case, the scenarios are:

- `input`: (1) empty string, (2) letters and spaces only, (3) special characters and spaces only, (4) one type of character only, (5) all characters.
- `alphabet`: (1) letters only, (2) letters with digits and special characters.
- `encode`: (1) true or (2) false.

Invalid inputs were not considered because they cannot test the metamorphic relations.

From these 5 source test cases, 5 follow-up test cases were designed for MR1 and MR2 each, and 4 for MR3.

The follow-up test cases for MR1 were designed by shuffling the `input` string in their corresponding sources while keeping `alphabet` and `encode` unchanged. They are denoted `mr1_fi1`, `mr1_fi2`, …, respectively. Together with the source cases, they form the metamorphic groups for MR1 (denoted `mr1_mgs`).

The follow-up test cases for MR2 were designed by concatenating the `input` string in their corresponding sources with itself while keeping `alphabet` and `encode` unchanged. They are denoted `mr2_fi1`, `mr2_fi2`, …, respectively. Together with the source cases, they form the metamorphic groups for MR2 (denoted `mr2_mgs`).

The follow-up test cases for MR3 cannot be formed without the outputs of the source test cases. Therefore, these are formed during run-time after executing the mutants with the source test cases. The follow-up cases are formed using this output, the same `alphabet` as the source case, and a flipped `encode` value.

The test cases can be found in the file `test_cases.js`.

Testing original program and measuring the effectiveness of MRs

A test script was written (in JavaScript) to run the testing and MR evaluation process. To test the original program, the script executes it with the metamorphic groups of each MR. It then checks if any MRs are violated. The output screenshot below shows that the program does not violate the MRs, meaning that the MRs do not find any faults in the original implementation.

To evaluate the MRs' effectiveness, the script executes every mutant with their metamorphic groups. For each MR, the script keeps track of the number of violations, number of satisfactions, and set of mutants detected. An MR's effectiveness is measured by the ratio between its number of violations and the sum of its violations and satisfactions:

$$effectivenessRatio = \frac{number\ of\ violations}{number\ of\ violations + number\ of\ satisfactions}$$

The result of this process is seen below. For more details about the script, refer to the file `run_tests.js`.

```
pine@pine-Vostro-3670:~/Documents/Programming/testing/task3$ node run_tests.js
==========Testing original program==========

MR1 - Permutation of Input String results: Not violated.

MR2 - Self-concatenation results: Not violated.

MR3 - Encode-Decode Symmetry results: Not violated.

==========Measuring MR effectiveness with mutants==========

MR1 - Permutation of Input String results:
        30 violations, 85 satisfactions. Effectiveness ratio: 0.2608695652173913
        6 mutants killed: 1, 3, 4, 9, 10, 11

MR2 - Self-concatenation results:
        34 violations, 81 satisfactions. Effectiveness ratio: 0.2956521739130435
        8 mutants killed: 1, 3, 4, 9, 10, 11, 17, 22

MR3 - Encode-Decode Symmetry results:
        62 violations, 30 satisfactions. Effectiveness ratio: 0.6739130434782609
        20 mutants killed: 1, 3, 4, 6, 7, 8, 9, 10, 11, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23
```

Fig. 3: Testing and evaluation output.

Results and discussion

The following tables summarize the testing results:

Mutant no.	Killed by MR1	Killed by MR2	Killed by MR3	Note
1	✓	✓	✓	
2	×	×	×	Mutated code only runs when input is invalid.
3	✓	✓	✓	
4	✓	✓	✓	
5	×	×	×	Mutated code only runs when input is invalid.
6	×	×	✓	
7	×	×	✓	
8	×	×	✓	
9	✓	✓	✓	
10	✓	✓	✓	
11	✓	✓	✓	
12	×	×	×	Mutated code only runs when input is invalid.
13	×	×	✓	
14	×	×	✓	
15	×	×	✓	
16	×	×	✓	
17	×	✓	✓	
18	×	×	✓	
19	×	×	✓	
20	×	×	✓	
21	×	×	✓	
22	×	✓	✓	
23	×	×	✓	

Table 3: Mutant status.

MR	# of violations	# of satisfactions	Effectiveness ratio	# of mutants killed
1 - Permutation of Input String	30	85	0.260	6 out of 23
2 - Self-concatenation	34	81	0.295	8 out of 23
3 - Encode-Decode Symmetry	62	30	0.674	20 out of 23

Table 4: Effectiveness and mutant kill count of MRs. *(Note: The total number of violations and satisfactions for MR3 is less than MR1 and MR2 because it was tested with only 4 metamorphic groups instead of 5.)*

Discussion

Out of the three metamorphic relations, MR1 is the least effective, killing only 6 mutants. MR2 is slightly more effective, killing 8, but pales in comparison to MR3, which killed 20. MR3 was able to kill all the mutants detected by both MR1 and MR2. This shows that for the substitution cipher problem, checking the encode-decode symmetry property is the most powerful way to find errors.

The result also shows that mutation testing is an effective way to evaluate metamorphic relations. If mutation testing had not revealed the low kill rate of MR1 and MR2, a tester would think that just these two relations are sufficient. MR3 was only added because, during the early stages of this testing project, MR1 and MR2 were found to be too ineffective. This proves that while a problem can have many metamorphic relations, some are more powerful than others. Mutation testing is a good way to determine which MRs to prioritize.

One problem that all 3 MRs suffer from is their inability to kill mutants 2, 5, and 12. The reason is that their mutated code can only run if the provided alphabet is invalid. More specifically, mutant #2 requires the alphabet to be missing AND not have a length of 26, mutant #5 requires the alphabet to be missing OR not have a length of 26, and mutant #12 requires the alphabet to have repeating characters. As all the metamorphic groups consist of valid inputs and thus valid alphabets, it is expected that they cannot kill these mutants. This shows that metamorphic testing should be used in conjunction with other testing methods. For example, random testing can also be used to generate random, potentially invalid, test cases that can reach and kill the mutated code.