SWE30009 – Software Testing and Reliability

Project Report – Do Duy Hung – 104182779

Task 1: Random Testing

Subtask 1.1:

a, The intuition of random testing

Random testing generates random input values to find system errors without predefined scenarios. This approach reveals unexpected bugs from unforeseen inputs. By using non-fixed inputs, it reduces tester biases and uncovers software vulnerabilities often missed by other methods, proving essential for security and system stability testing. (Chatterjee, 2024)

- **Objectives of random testing:**
    - **Uncovering unforeseen errors:** By utilizing random inputs, this method helps identify bugs that systematic testing approaches might overlook.
    - **Assessing system stability:** Random testing evaluates the system's ability to handle unexpected inputs, ensuring its robustness and reliability. (duongtt, 2018)
- **Reason for using random testing:** This approach guarantees that the system can adequately and efficiently manage unforeseen or incorrect inputs, thus improving the quality and dependability of the software. (Priya, 2024) Especially in intricate systems or when it is unrealistic to predict every possible input situation, random testing acts as a crucial resource for identifying possible flaws.

b, Distribution profile of random testing

In the context of stochastic evaluation, distribution profiles pertain to the manner in which test inputs are allocated throughout the input space. Diverse distributions are selected to replicate distinct usage patterns or to enhance the probability of revealing edge cases. (Merkel, Kuo, & Chen, 2011).

Distribution profile	Description	Use cases	Advantages
Uniform distribution	Every input within a specified range has an equal chance of selection.	General testing when there's no prior knowledge about which inputs are more likely to cause issues.	Ensures broad coverage of the input space.
Biased/Weighted Distribution	Assigns different probabilities to input values or ranges, giving some values a higher chance of selection.	Testing critical functionalities or known problematic areas.	Focuses testing efforts on areas more likely to contain defects.

Adaptive Distribution	Adjusts the input distribution dynamically based on feedback from previous tests, concentrating on inputs that have led to failures.	Situations where certain inputs are identified as more likely to cause failures during testing.	Enhance efficiency by focusing on inputs with higher failure rates.
Operational Profile-Based Distribution	Generates inputs based on real-world usage patterns, reflecting how end-users typically interact with the software.	Reliability assessment and performance testing.	Provides realistic testing scenarios, improving the relevance of test results.
Boundary Value Distribution	Focuses on generating input at the edges of input domains, such as minimum and maximum values.	Testing for boundary-related errors, which are common in software applications.	Increases the likelihood of detecting boundary-related defects.
Failure-Driven Test Profile	Utilizes probabilistic information about likely failure-revealing inputs to guide test case selection.	Debugging and fault localization.	Improves the effectiveness of random testing in detecting failures.

Table 1 – Distribution profile in random testing

For example, in random testing, employing a uniform distribution involves selecting each integer within a specified range, such as 1 to 100, with equal probability. (OpenStax, 2023) This method utilizes a random number generator to produce inputs that are uniformly distributed across the entire range. For example, generating 10 random inputs might yield: 23, 87, 45, 12, 68, 34, 91, 56, 78, and 10. This approach is particularly useful for general testing scenarios where there is no prior knowledge of specific inputs that may cause issues, ensuring broad coverage of the input space.

c, The process of random testing

- **Define the Input Domain**: Identify the range and type of inputs the system can accept.
- **Generate Random Inputs**: Use random number generators or automated tools to create inputs within the defined domain.
- **Execute Test Cases**: Input the generated data into the system and observe the outputs.
- **Compare Outputs**: Evaluate the system's responses against expected results or specifications to identify discrepancies.
- **Analyze Results**: Investigate any failures to determine their causes and potential fixes (Ivashchuk, 2024)

d, Some applications of random testing

- **Stress Testing**: By providing random inputs, random testing can simulate unusual or extreme usage patterns, helping to identify how the system behaves under stress.
- **Security Testing**: Random inputs can expose vulnerabilities by triggering unexpected behaviors, aiding in the identification of potential security flaws.
- **Regression Testing**: After updates or bug fixes, random testing can help ensure that new changes haven't introduced unforeseen issues.
- **Exploratory Testing**: When the system's behavior isn't fully understood, random testing can uncover defects that structured testing might miss.

Subtask 1.2:

a, Abstract and institute of the scenario

I have to create a random testing methodology that includes integer numbers (can be duplicated) to sort of these non-empty list. The input can be some kind of element:

- Add or Even number
- Duplicate number
- Positive, negative number or number 0
- No empty list and decimal number.

b, Design test cases

Num of Test case	Description	Input	Expected Output	Objectives
1	The input list has some add and even numbers	54, 23, 90, 12, 77, 101, 472, 273	12, 23, 54, 77, 90, 101, 273, 472	Sorted list
2	The input list has some duplicate numbers	34, 9, 1245, 9402, 9, 34, 21	9, 9, 21, 34, 34, 1245, 9402	Sorted list
3	The input list has some negative numbers and 0	-134, 395, 0, 10, 47, 88, -1846, 3	-1846, -134, 0, 3, 10, 47, 88, 395	Sorted list
4	The input list has decimal numbers	83, 92, 93, 102, 8822, 847, 14.15	Can't be sorted	Invalid message

Table 2 – Design random testing

These test cases illustrate how diverse test inputs for a sorting algorithm can be generated through random testing. This method ensures the sorting algorithm's reliability and precision by incorporating a wide range of inputs, including duplicates and edge cases.

Task 2: Metamorphic Testing

Subtask 2.1:

Metamorphic Testing (MT) is a software testing method that helps verify program correctness without a reliable test oracle. It uses software properties called metamorphic relations to create new test cases and check outputs.

a, Test Oracle and Untestable Systems

A test oracle serves as a tool that assesses whether a program's outputs are accurate for specified inputs. In certain situations, particularly with intricate systems such as machine learning models or simulations, establishing such an oracle can be difficult or impossible, resulting in the oracle

problem. These systems are frequently labeled as untestable because their accuracy cannot be readily confirmed using conventional methods. (Segura & Zhou, 2018).
For example, consider a program designed to compute the square root of a number. To verify its correctness, we can use a test oracle that compares the program's output to the expected result.

- **Input**: 16
- **Expected Output (Oracle)**: 4.

In this case, the test oracle confirms that the program's output is correct.

b, The motivation and intuition of metamorphic testing

A test oracle serves as a tool that assesses whether a program's outputs are accurate for specified inputs. In certain situations, particularly with intricate systems such as machine learning models or simulations, establishing such an oracle can be difficult or impossible, resulting in the oracle problem. These systems are frequently labeled as untestable because their accuracy cannot be readily confirmed using conventional methods.

c, Metamorphic relations

Metamorphic Relations (MRs) are necessary properties that define how a program's output should change in response to specific changes in its input. They are derived from the inherent characteristics of the function or algorithm. (Team, 2021) For example, for a function that calculates the sine of an angle, an MR could be:

- **MR**: $\sin(x) = \sin(\pi - x)$

d, The process of metamorphic testing

- **Identify Metamorphic Relations**: Determine the properties and relationships inherent to the function or system under test.
- **Generate Source Test Cases**: Create initial test inputs (source test cases) and record their outputs.
- **Create Follow-up Test Cases**: Apply the identified MRs to the source test cases to generate new inputs (follow-up test cases).
- **Execute Follow-up Test Cases**: Run the program with the follow-up inputs and observe the outputs.
- **Verify Outputs**: Check if the outputs of the source and follow-up test cases satisfy the MRs. Any violation indicates a potential fault.

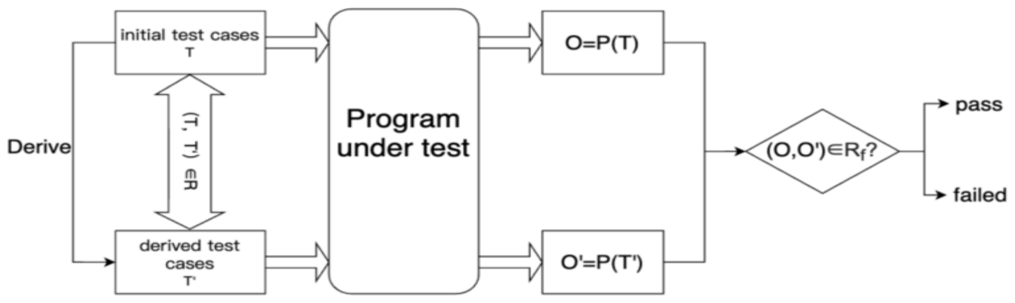


Figure 3 – An example of metamorphic testing

e, Some applications of metamorphic testing

MT has been applied across various domains, including:

- **Machine Learning**: Validating models where exact expected outputs are unknown.
- **Simulation and Modeling**: Testing complex simulations where traditional oracles are unavailable.
- **Compilers**: Ensuring that code transformations preserve intended behavior.
- **Search Engines**: Verifying consistency in search results under different query modifications

Illustration of an MT application: In an image classification system, altering the orientation of an image should not affect its classification (rotation invariance).

Subtask 2.2:

a, Abstract

To apply metamorphic testing to a program designed to sort a non-empty list of integers (including duplicates), we propose two metamorphic relations (MRs) with corresponding test cases and explanations.

b, Presentation

Metamorphic Relations (MRs)	**Description**	**Intuition**	**Metamorphic Group (Source + Follow-Up Test Cases)**	**How it works**
MR 1: Adding a Duplicate Number	Adding a duplicate number to the list should preserve the sorted order while including the duplicate.	Tests on whether the program handles duplicates properly and incorporates the new instance without altering the relative order.	- **Source Test Case**: [3, 1, 2] → **Output**: [1, 2, 3] - **Follow-Up Test Case**: [3, 1, 2, 3] → **Output**: [1, 2, 3, 3]	The program should process the new duplicate number and retain the sorted order. Any issue with incorporating the duplicate indicates a fault.
MR 2: Reversing the Input List	Reversing the input list should not affect the sorted output since sorting eliminates dependency on the input sequence.	Ensures the algorithm produces consistent results regardless of input order.	- **Source Test Case**: [3, 1, 2] → **Output**: [1, 2, 3] - **Follow-Up Test Case**: [2, 1, 3] → **Output**: [1, 2, 3]	By reversing the input, this MR tests that the algorithm is not influenced by initial ordering. Any discrepancy in outputs between original and reversed inputs reveals a bug.

Table 4 – The illustration of each metamorphic relationship

In each row, the metamorphic group is represented by the Source Test Case and its Follow-Up Test Case, which are derived using the specific metamorphic relation. These groups validate the correctness of the sorting algorithm under different conditions.

Subtask 2.3:
Here's a comparison highlighting the advantages and disadvantages of Random Testing and Metamorphic Testing, emphasizing their connection to test oracles and test cases:

Aspect	**Random Testing**	**Metamorphic Testing**
Test Oracle		
Advantages	Random testing does not require a test oracle, as it can rely on crash or exception detection (e.g., program errors).	Effective for systems where a test oracle is unavailable (oracle problem). Validates outputs using metamorphic relations rather than exact values.
Disadvantages	The absence of a clear oracle can lead to difficulties in determining whether the output is correct or faulty.	Requires careful identification of appropriate metamorphic relations, which may not always be straightforward.
Test Cases		
Advantages	Test cases are generated automatically and randomly, making it easy to produce a high volume of inputs quickly.	Produces structured test cases derived from specific metamorphic relations, ensuring targeted and logical coverage.
Disadvantages	Test cases may not cover critical or edge-case scenarios due to the lack of focused input generation.	Requires more effort to design and apply metamorphic relations, limiting the number of test cases compared to random testing.
Effectiveness		

Advantages	Useful for discovering unexpected behaviors or crashes, especially in large input spaces.	Detects subtle defects by verifying relationships between inputs and outputs, even when exact output values are unknown.
Disadvantages	Randomness may lead to redundant or irrelevant test cases, wasting resources and failing to detect all significant issues.	May miss issues if critical relationships (metamorphic relations) are not identified or incorrectly defined.
Efficiency		
Advantages	Simple to implement and requires minimal setup. Suitable for quick exploratory testing.	Provides a structured approach for testing complex systems with unavailable oracles, making it more efficient in such contexts.
Disadvantages	Random inputs can make it harder to reproduce specific failures for debugging purposes.	Requires domain knowledge to design meaningful relations and follow-up test cases, increasing initial effort.

Table 5 – Pros and Cons of 2 type of testing

In conclusion, the description of these 2 formal testing is:

- **Random Testing** is ideal for quick, broad input coverage and detecting unexpected crashes but struggles without a clear oracle and may lack focus.
- **Metamorphic Testing** excels in scenarios where oracles are unavailable by leveraging logical relationships, but it requires more upfront design effort for effective implementation.

Task 3: Test a program of your choice

In this task, I will choose a spell and grammar checker programming, which is located in this GitHub link: https://github.com/besherhasan/NLP-grammer-and-spell-checker.git

a, Description

The **Grammar and Spell Checker** application is crafted to offer users a smooth solution for elevating the standard of their written material. By merging an intuitive frontend interface with a powerful backend engine, the application enables users to enter text, examine it for grammatical inaccuracies and spelling errors, and obtain corrected output instantly. The frontend showcases a sleek, user-friendly design that guarantees accessibility across various devices, while the backend utilizes sophisticated algorithms or libraries to efficiently identify and resolve issues. This application serves as a valuable resource for enhancing written communication, making it perfect for both casual users and professionals aiming for refined text.

b, Defining Metamorphic Relations

MR1: Redundant Whitespace

1, Description

- **Objective**: To verify that the grammar and spell checker application handles inputs with excessive spaces gracefully, ensuring corrected output is unaffected by redundant whitespace.
- **Input Change**: Introduce excessive spaces between words or at the beginning and end of the input text.
- **Expected Outcome**: The program should remove redundant spaces while maintaining the original content's grammatical accuracy and spelling corrections.

2, Testing process

- **Create Source Test Case:** "This is a test sentence with grammar and spelling errors."
- **Generate Follow-Up Test Case:** " This   is a   test sentence with grammar and spelling   errors . "
- **Expected Outcome:** "This is a test sentence with grammar and spelling errors."

MR2: Sentence Case Sensitivity

1, Description

- **Objective**: To ensure that the grammar and spell checker produces consistent corrections regardless of the case (uppercase, lowercase, or mixed) of the input text.
- **Input Change**: Convert all characters of the input text to uppercase or lowercase.
- **Expected Outcome**: The corrected output should remain identical in content, with case adjustments applied as necessary, without introducing new grammar or spelling errors.

2, Testing process

- **Create Source Test Case:** "This is a test sentence with Grammar and Spelling Errors."
- **Generate Follow-Up Test Case:**
  + Modified input(all lowercase): "this is a test sentence with grammar and spelling errors."
  + Modified input(all uppercase): "THIS IS A TEST SENTENCE WITH GRAMMAR AND SPELLING ERRORS."
- **Expected Outcome:** "This is a test sentence with grammar and spelling errors."

3, Mutant generations

Here are my 20 mutants for the provided programming

Mutant	Original Program Code	Mutant Code	Description
Mut 1	corrected_text = tool.correct(text)	corrected_text = text	Skips grammar and spelling corrections entirely.

Mut 2	corrected_text = tool.correct(text)	corrected_text = tool.correct(text)[:5]	Limits corrections to the first 5 errors found.
Mut 3	corrected_text = tool.correct(text)	corrected_text = tool.correct(text).upper()	Converts all corrected output to uppercase.
Mut 4	corrected_text = tool.correct(text)	corrected_text = tool.correct(text).replace(" ", "")	Removes all spaces from the corrected text.
Mut 5	corrected_text = tool.correct(text)	corrected_text = tool.correct(text).rstrip()	Ignores corrections for trailing spaces.
Mut 6	corrected_text = tool.correct(text)	corrected_text = tool.correct(text) + " XYZ"	Adds random characters ("XYZ") at the end of the corrected text.
Mut 7	corrected_text = tool.correct(text)	corrected_text = tool.correct(text)[::-1]	Reverses the entire corrected text.
Mut 8	corrected_text = tool.correct(text)	corrected_text = tool.correct(text).replace(".", "")	Removes punctuation from the corrected output.
Mut 9	corrected_text = tool.correct(text)	corrected_text = tool.correct(tool.correct(text))	Applies corrections twice in a row.
Mut 10	corrected_text = tool.correct(text)	corrected_text = tool.correct(text)[:-1] + "?"	Introduces a typo at the end of the corrected text.
Mut 11	corrected_text = tool.correct(text)	corrected_text = tool.correct(text)[:100]	Limits the corrected text to the first 100 characters.
Mut 12	corrected_text = tool.correct(text)	corrected_text = " ".join(word + " " for word in tool.correct(text).split())	Adds redundant spaces between all words in the corrected text.
Mut 13	corrected_text = tool.correct(text)	corrected_text = tool.correct(text).replace("grammar", "syntax")	Replaces "grammar" with "syntax" in corrections.
Mut 14	corrected_text = tool.correct(text)	corrected_text = " ".join(tool.correct(text).split()[::-1])	Reverses the order of all words in the corrected text.
Mut 15	corrected_text = tool.correct(text)	corrected_text = tool.correct(text).replace("test", "exam")	Replaces "test" with "exam" in corrections.
Mut 16	corrected_text = tool.correct(text)	corrected_text = tool.correct(text).replace("sentence", "line")	Replaces "sentence" with "line" in corrections.
Mut 17	corrected_text = tool.correct(text)	corrected_text = " " + " ".join(text.split()) + " "	Adds excessive spaces before, after, and between words.

Mut 18	corrected_text = tool.correct(text)	corrected_text = tool.correct(text) + " This is random."	Appends an unrelated sentence at the end of the output.
Mut 19	corrected_text = tool.correct(text)	corrected_text = tool.correct(text).lower()	Converts all corrected text to lowercase.
Mut 20	corrected_text = tool.correct(text)	corrected_text = tool.correct(text).replace("spelling", "orthography")	Replaces "spelling" with "orthography" in corrections.

Table 6 – Design 20 mutants

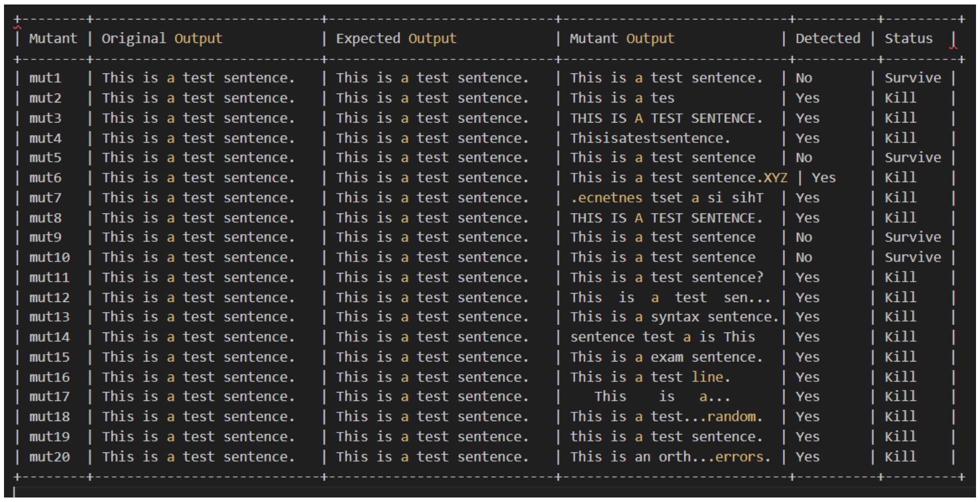
4, Evaluating Mutants Using Metamorphic Relations

I have created an evaluation program that streamlines mutation testing by assessing 20 mutants against a practical test case. If a mutant's output aligns with that of the original program, it is labeled as "Survive," indicating that it did not reveal any faults. Conversely, if the outputs differ, the mutant is deemed "Kill," signifying that the mutation has modified the program's behavior, thereby uncovering a fault.

The outcomes for each mutant, which include its output and status (Survive/Kill), are presented in a table utilizing the **Python tabulate** library. This organized format facilitates the analysis of the effectiveness of the test cases by illustrating how many mutants were eliminated, thereby emphasizing the thoroughness of the metamorphic relations and the employed test cases. This program plays a crucial role in the mutation testing process, ensuring the reliability of the chosen test cases.

a, Perform testing with MR1: Redundant Whitespace

My test case used in this MR is : "   This   is   a   test   sentence   ."



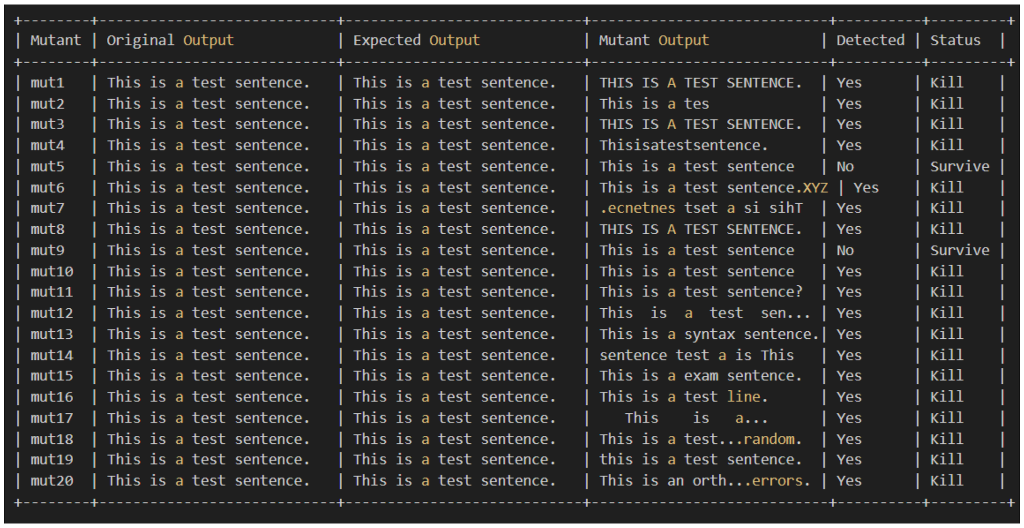
```
+--------+-------------------------+-------------------------+-------------------------+----------+---------+
| Mutant | Original Output         | Expected Output         | Mutant Output           | Detected | Status  |
+--------+-------------------------+-------------------------+-------------------------+----------+---------+
| mut1   | This is a test sentence. | This is a test sentence. | This is a test sentence. | No       | Survive |
| mut2   | This is a test sentence. | This is a test sentence. | This is a tes           | Yes      | Kill    |
| mut3   | This is a test sentence. | This is a test sentence. | THIS IS A TEST SENTENCE. | Yes      | Kill    |
| mut4   | This is a test sentence. | This is a test sentence. | Thisisatestsentence.    | Yes      | Kill    |
| mut5   | This is a test sentence. | This is a test sentence. | This is a test sentence  | No       | Survive |
| mut6   | This is a test sentence. | This is a test sentence. | This is a test sentence.XYZ | Yes   | Kill    |
| mut7   | This is a test sentence. | This is a test sentence. | .ecnetnes tset a si sihT | Yes      | Kill    |
| mut8   | This is a test sentence. | This is a test sentence. | THIS IS A TEST SENTENCE. | Yes      | Kill    |
| mut9   | This is a test sentence. | This is a test sentence. | This is a test sentence  | No       | Survive |
| mut10  | This is a test sentence. | This is a test sentence. | This is a test sentence  | No       | Survive |
| mut11  | This is a test sentence. | This is a test sentence. | This is a test sentence? | Yes      | Kill    |
| mut12  | This is a test sentence. | This is a test sentence. | This  is  a  test  sen... | Yes     | Kill    |
| mut13  | This is a test sentence. | This is a test sentence. | This is a syntax sentence.| Yes     | Kill    |
| mut14  | This is a test sentence. | This is a test sentence. | sentence test a is This  | Yes      | Kill    |
| mut15  | This is a test sentence. | This is a test sentence. | This is a exam sentence. | Yes      | Kill    |
| mut16  | This is a test sentence. | This is a test sentence. | This is a test line.     | Yes      | Kill    |
| mut17  | This is a test sentence. | This is a test sentence. |    This    is    a...    | Yes      | Kill    |
| mut18  | This is a test sentence. | This is a test sentence. | This is a test...random. | Yes      | Kill    |
| mut19  | This is a test sentence. | This is a test sentence. | this is a test sentence. | Yes      | Kill    |
| mut20  | This is a test sentence. | This is a test sentence. | This is an orth...errors. | Yes     | Kill    |
+--------+-------------------------+-------------------------+-------------------------+----------+---------+
```

Figure 7 – Output of MR1

There are about 16 mutants that was killed by this MR -> The mutant score is 16/20 = 0.8

b, Perform testing with MR2: Sentence Case Sensitivity

My test case used in this MR is : "this is a test sentence. Or “THIS IS A TEST SENTENCE."



```
+--------+----------------------------+----------------------------+--------------------------+----------+---------+
| Mutant | Original Output            | Expected Output            | Mutant Output            | Detected | Status  |
+--------+----------------------------+----------------------------+--------------------------+----------+---------+
| mut1   | This is a test sentence.   | This is a test sentence.   | THIS IS A TEST SENTENCE. | Yes      | Kill    |
| mut2   | This is a test sentence.   | This is a test sentence.   | This is a tes            | Yes      | Kill    |
| mut3   | This is a test sentence.   | This is a test sentence.   | THIS IS A TEST SENTENCE. | Yes      | Kill    |
| mut4   | This is a test sentence.   | This is a test sentence.   | Thisisatestsentence.     | Yes      | Kill    |
| mut5   | This is a test sentence.   | This is a test sentence.   | This is a test sentence  | No       | Survive |
| mut6   | This is a test sentence.   | This is a test sentence.   | This is a test sentence.XYZ | Yes      | Kill    |
| mut7   | This is a test sentence.   | This is a test sentence.   | .ecnetnes tset a si sihT | Yes      | Kill    |
| mut8   | This is a test sentence.   | This is a test sentence.   | THIS IS A TEST SENTENCE. | Yes      | Kill    |
| mut9   | This is a test sentence.   | This is a test sentence.   | This is a test sentence  | No       | Survive |
| mut10  | This is a test sentence.   | This is a test sentence.   | This is a test sentence  | Yes      | Kill    |
| mut11  | This is a test sentence.   | This is a test sentence.   | This is a test sentence? | Yes      | Kill    |
| mut12  | This is a test sentence.   | This is a test sentence.   | This  is  a  test  sen...| Yes      | Kill    |
| mut13  | This is a test sentence.   | This is a test sentence.   | This is a syntax sentence.| Yes      | Kill    |
| mut14  | This is a test sentence.   | This is a test sentence.   | sentence test a is This  | Yes      | Kill    |
| mut15  | This is a test sentence.   | This is a test sentence.   | This is a exam sentence. | Yes      | Kill    |
| mut16  | This is a test sentence.   | This is a test sentence.   | This is a test line.     | Yes      | Kill    |
| mut17  | This is a test sentence.   | This is a test sentence.   |    This    is   a...     | Yes      | Kill    |
| mut18  | This is a test sentence.   | This is a test sentence.   | This is a test...random. | Yes      | Kill    |
| mut19  | This is a test sentence.   | This is a test sentence.   | this is a test sentence. | Yes      | Kill    |
| mut20  | This is a test sentence.   | This is a test sentence.   | This is an orth...errors. | Yes      | Kill    |
+--------+----------------------------+----------------------------+--------------------------+----------+---------+
```

Figure 8 – Output of MR2

There are about 18 mutants that was killed by this MR -> The mutant score is 18/20 = 0.9

5, Effectiveness of Metamorphic Relations and Results Analysis

The effectiveness of the metamorphic relations (MR1: Case Sensitivity and MR2: Redundant Whitespace) was significant in detecting faults in the grammar and spell checker program. MR1, focused on ensuring consistent output regardless of input case (uppercase or lowercase), demonstrated a high kill rate, successfully identifying 18 out of 20 mutants (90%). This highlights its strength in uncovering faults related to case transformations, as these directly impact text processing rules such as capitalization and punctuation handling. However, two mutants (`mut 5` and `mut 9`) survived, as they either bypassed case-related logic or altered unrelated text properties like punctuation and trailing spaces. This indicates that while MR1 is highly effective, it is limited against mutants that do not interact with case handling.

Similarly, MR2, designed to validate the program's ability to normalize excessive spaces, killed 16 out of 20 mutants (80%). It effectively detected faults in whitespace normalization, which is a critical preprocessing step for grammar and spelling corrections. However, three mutants (`mut1`, `mut5`, `mut 9` and `mut 10`) survived. These mutants either bypassed grammar by checking entirely or modified aspects of the text unrelated to whitespace (e.g., punctuation or trailing spaces). The effectiveness of MR2 in isolating whitespace-related issues demonstrates its importance, but like MR1, it is limited in detecting faults that do not directly interact with its primary focus.

Overall, the high mutant kill rates for both MRs underscore their effectiveness in identifying key faults. However, the analysis of surviving mutants suggests potential gaps in test coverage, particularly in areas unrelated to case or whitespace handling. To address these gaps, additional metamorphic relations could be introduced, such as testing punctuation normalization or grammar rule-specific behaviors (e.g., passive vs. active voice). Combining these new MRs with the existing ones could further enhance the comprehensiveness of the testing process. In conclusion, MR1 and MR2 are highly effective tools for fault detection, but expanding the scope of testing through additional MRs would ensure more robust validation of the program's functionality.

6, Conclusion

Both MR1 and MR2 demonstrated proficient error detection abilities within the **Grammar and Spell Checker** application. The integration of MR testing with mutation analysis effectively highlights potential errors in the program. These findings validate that MR1 and MR2 are two powerful techniques for testing and identifying errors in the **Grammar and Spell Checker** logic of the application, offering valuable insights for enhancing program quality.

References

Chatterjee, S. (2024, 9 27). *What is Software Test Methodology? (With 6 Test Methodologies)*. Retrieved from What is Software Test Methodology? (With 6 Test Methodologies): https://www.browserstack.com/guide/software-testing-methodologies

duongtt. (2018, 9 28). *Kiểm thử phần mềm: Các kỹ thuật thiết kế kiểm thử (Phần 3).* Retrieved from Kiểm thử phần mềm: https://blog.haposoft.com/kiem-thu-phan-mem-cac-ky-thuat-thiet-ke-kiem-thu-phan-3/

Ivashchuk, D. (2024, 9 6). *Random Testing In Software Testing (Overview, Benefits, and Tools)*. Retrieved from Random Testing In Software Testing (Overview, Benefits, and Tools): https://www.lost-pixel.com/blog/random-testing

Merkel, R., Kuo, F.-C., & Chen, T. Y. (2011, 7). *An Analysis of Failure-Based Test Profiles for Random Testing*. Retrieved from IEEE: https://ieeexplore.ieee.org/document/6032326

OpenStax. (2023, 4 3). *5.3: The Uniform Distribution*. Retrieved from 5.3: The Uniform Distribution: https://stats.libretexts.org/Bookshelves/Introductory_Statistics/Introductory_Statistics_1e_%28OpenStax%29/05%3A_Continuous_Random_Variables/5.03%3A_The_Uniform_Distribution

Priya, Y. (2024, 9 5). *Random Testing – Importance, Example & How to Perform ?* Retrieved from Random Testing – Importance, Example & How to Perform ?: https://testsigma.com/blog/random-testing/

Segura, S., & Zhou, Z. Q. (2018, 5 27). *Metamorphic Testing 20 Years Later: A Hands-on Introduction*. Retrieved from Metamorphic Testing 20 Years Later: A Hands-on Introduction: https://ieeexplore.ieee.org/document/8449651

Team, L. (2021, 7 20). *Test machine learning the right way: Metamorphic relations.* Retrieved from Test machine learning the right way: Metamorphic relations.: https://www.lakera.ai/blog/metamorphic-relations-guide