Software Testing Project
Mutation Testing

Jainav Sanghvi(IMT2020098)
Arin Awasthi(IMT2020081)

Project Aim

The aim of this project is to use Mutation testing to test a real-world software project with the help of open-source tools.

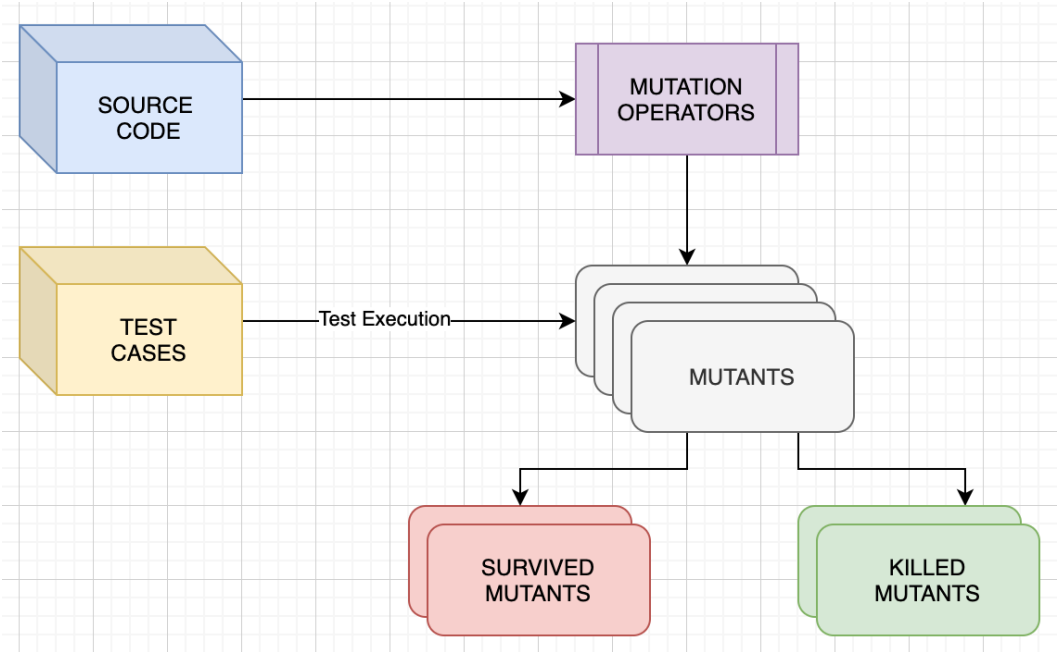
Github Repository Link (Click here)

About the Software

The software is a collection of different string algorithms that we come across while we do study data structures and algorithm. It consists of different types of string matching algorithm, string problems involving dynamic programming and various other hard string problems.
Given the importance of this software, it was decided to apply mutation testing to the project. Our proprietary codebase served as the testing framework's reference during the testing process.

What is Mutation Testing?

Mutation testing is a software testing technique that assesses the effectiveness of a test suite by introducing small, intentional changes (mutations) into the source code and determining whether the existing tests can detect these changes. Each mutation represents a potential bug, and the goal is to ensure that the test suite is sensitive enough to identify such alterations. If a mutation is not detected by the tests, it implies a weakness in the test suite, indicating that the code coverage or the quality of the tests may be insufficient. Mutation testing helps developers identify areas of improvement in their test suites and enhances overall software reliability by providing insights into the robustness of the testing process.



Mutation testing involves two paradigms for evaluating the effectiveness of test cases: Weak Mutation Killing and Strong Mutation Killing.

In the Weak Mutation Killing paradigm, the memory state of the program after the execution of a mutated statement differs from the memory state when the statement remains unaltered and is executed. Interestingly, in this scenario, the program's output on a given test case may remain unchanged, regardless of whether a program statement has undergone mutation.

Conversely, in the Strong Mutation Killing paradigm, the output of the program on a given test case must demonstrate a noticeable change when a statement undergoes mutation compared to its unaltered counterpart. Importantly, strong killing of mutants indicates that the error introduced by mutation propagates through the program, leading to distinct outputs in the presence and absence of the mutant.

Our adoption of mutation testing as a testing strategy is characterized by a deliberate emphasis on the robust neutralization of mutants through strong mutation killing. This strategic approach aims to thoroughly validate the resilience and accuracy of the software under examination, ensuring that mutations result in discernible changes in the program's behavior and output.

Testing Strategies and Tools used

Chosen testing strategy mainly focuses on strong mutation killing.Strong Mutation Killing: Killing a mutant strongly requires that the program's output on a test case changes when a statement is mutated compared to when it's not mutated. This type of killing indicates that the error introduced by the mutation propagates through the program.
Tools used -

- VS Code IDE
- Pitest Java Library
- Maven
- JUnit

What is PIT ?

PiTest, or Pit, is an open-source mutation testing tool for Java that helps developers evaluate the effectiveness of their test suites. It works by introducing various small, syntactic changes (mutations) to the source code and then running the existing test suite to identify which mutations are detected and which are not. If a mutation goes undetected, it suggests a potential weakness in the test coverage. Pit generates comprehensive mutation reports, highlighting the mutation score and specific areas

in the code where the test suite may be improved. By providing insights into the thoroughness and fault-detection capabilities of tests, Pit encourages developers to enhance the reliability and effectiveness of their software testing strategies.

Why PIT?

There are other mutation testing systems for Java, but they are not widely used. They are mostly slow, difficult to use and written to meet the needs of academic research rather than real development teams.
PIT is different. It's

- fast - can analyse in minutes what would take earlier systems days
- easy to use - works with ant, maven, gradle and others
- actively developed
- actively supported

The reports produced by PIT are in an easy to read format combining line coverage and mutation coverage information.

Testing Approach

Given below is the list of algorithms implemented in the codebase -

1. Rabin Karp Algorithm
2. Z Algorithm
3. Knuth Morris Pratt Algorithm
4. Longest Common Subsequence
5. Longest Palindromic Subsequence
6. Super Sequence
7. Longest Substring that can form a palindrome
8. Longest Common Prefix
9. Longest Valid Parenthesis
10. Edit Distance
11. Manachers Algorithm
12. Boyer Moore Algorithm
13. Sequence Alignment
14. WildCard Pattern
15. Minimum Palindrome Partition
16. Longest Repeating Subsequence
17. Longest Prefix Suffix
18. KVowel Words
19. Left and Right Rotate
20. Reverse Vowel
21. Horspool BM Algo (RepeatedStringMatch)
22. Find All concatenated Words in a Dictionary
23. Words Break 1
24. Words Break 2

25. At most n given digit set (Total count of numbers that are smaller than n and can be formed using given digits)
26. Palindrome Pairs (Find i, j such that word[i] + word[j] form a palindrome)
27. Minimum Stickers required to make a word
28. KSimilarity (Swap 2 chars k times to make a target word)

Mutation Testing includes both unit and integration testing.

Unit Testing:
Unit testing is a software testing approach where individual units or components of a software application are isolated and tested in isolation. The purpose of unit testing is to validate that each unit of the software performs as intended by checking if the individual functions or methods produce the correct output for a given set of inputs. Unit tests are typically automated, enabling developers to identify and fix issues early in the development process. By isolating and verifying the smallest testable parts of the codebase, unit testing contributes to the overall reliability, maintainability, and efficiency of software systems.

Integration Testing:
Integration testing is a software testing phase that focuses on evaluating the interactions and interfaces between different components or modules of a system to ensure they function seamlessly as a unified whole. The goal of integration testing is to detect defects that may arise when integrating individual units or modules and to verify that the integrated components collaborate correctly, exchanging data and services as expected. It aims to identify issues such as communication errors, data inconsistencies, and functional mismatches that may arise when combining diverse parts of a software application. Integration testing plays a crucial role in validating the overall functionality, reliability, and performance of a software system by examining the collaborative behaviour of its integrated components.

Mutation Operators:
Mutation operators in PITest are specific types of changes introduced into the source code to simulate potential faults or bugs. The purpose of applying mutation operators is to assess the effectiveness of a test suite by determining if it can detect these artificially introduced changes. Each operator represents a potential fault that could occur in the code.
PIT currently provides some built-in mutators, of which most are activated by default. The default set can be overridden, and different operators selected, by passing the names of the required operators to the mutators parameter. To make configuration easier, some mutators are put together in groups. Passing the name of a group in the mutators parameter will activate all mutators of the group.

Name	Description
DEFAULTS	The Default group of Mutation Operators applied by PIT when no extra configuration was provided.
NEW_DEFAULTS	The Group contains the same set of Mutation Operators like DEFAULTS group, where Return Values group is different. In the future, maybe it would become the default group.
STRONGER	Provides a few more Mutation Operators in addition to the default group.
RETURNS	Provides Mutation Operators related to the return values.
REMOVE_CONDITIONAL	Provides Mutation Operators related to the 'if, else' statements.
ALL	Provides all mutators defined by PIT.

Here are some of the mutation operators(unit + integration) provided by PITest:

- Arithmetic Operator Replacement: Like the math mutator, this mutator replaces binary arithmetic operations for either integer or floating-point arithmetic with another operation.
- Relational Operator Replacement: This mutator replaces a relational operator with another one.
- Conditionals Boundary Mutator: The conditionals boundary mutator replaces the relational operators <, <=, >, >=
- Increments Mutator: The increments mutator will mutate increments, decrements and assignment increments and decrements of local variables (stack variables). It will replace increments with decrements and vice versa.
- Invert Negatives Mutator: The invert negatives mutator inverts negation of integer and floating point variables.
- Math Mutator: The math mutator replaces binary arithmetic operations for either integer or floating-point arithmetic with another operation.
- Negate Conditionals Mutator: The conditionals boundary mutator replaces the relational operators <, <=, >, >=
- Constructor Call Mutator: Optional mutator that replaces constructor calls with null values.
- Empty returns Mutator: Replaces return values with an 'empty' value for that type
- False returns Mutator : Replaces primitive and boxed boolean return values with false. Pitest will filter out equivalent mutations to methods that are already hard coded to return false.
- Void Method Call Mutator: The void method call mutator removes method calls to void methods.

Active mutators

- CONDITIONALS_BOUNDARY
- CONSTRUCTOR_CALLS
- EMPTY_RETURNS
- FALSE_RETURNS
- INCREMENTS
- INVERT_NEGS
- MATH
- NON_VOID_METHOD_CALLS
- NULL_RETURNS
- PRIMITIVE_RETURNS

Results

Total Lines in the code= 1440 (App.java) + 520(AppTest.java)=1960
Our final test case design produced excellent results for us i.e., Line Coverage - 97% and Mutation Coverage - 81%.
Initially, the scores were a bit low, but by examining the mutants that survived and creating test cases to raise the mutation coverage percentage, we attempted to increase this mutation coverage. We examined the lines that the test cases did not cover and added test cases accordingly.

Pit Test Coverage Report

Project Summary

Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
1	97%	691/712	81%	940/1160	82%	940/1141

Breakdown by Package

Name	Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
src	1	97%	691/712	81%	940/1160	82%	940/1141

Report generated by PIT 1.15.3

Enhanced functionality available at arcmutate.com

App.java

```
1    package src;
2
3    import java.util.*;
4
5    public class App {
6
7            // EDIT DISTANCE
8            int EditDistance(String s1, String s2, int n, int m, int[][] dp) {
9
10                   // If any String is empty,
11                   // return the remaining characters of other String
12 1                 if (n == 0)
13 1                         return m;
14 1                 if (m == 0)
15 1                         return n;
16
17                   // To check if the recursive tree
18                   // for given n & m has already been executed
19 1                 if (dp[n][m] != -1)
20 1                         return dp[n][m];
21
22                   // If characters are equal, execute
23                   // recursive function for n-1, m-1
24 5                 if (s1.charAt(n - 1) == s2.charAt(m - 1)) {
25 4                         return dp[n][m] = EditDistance(s1, s2, n - 1, m - 1, dp);
26                   }
27                   // If characters are not equal, we need to
28                   // find the minimum cost out of all 3 operations.
29                   else {
30
31                           int insert, del, replace; // temp variables
32
33 2                         insert = EditDistance(s1, s2, n, m - 1, dp);
34 2                         del = EditDistance(s1, s2, n - 1, m, dp);
35 3                         replace = EditDistance(s1, s2, n - 1, m - 1, dp);
36 4                         return dp[n][m] = 1 + Math.min(insert, Math.min(del, replace));
37                   }
38           }
39
40           // KMP ALGORITHM
41           int KMPSearch(String pat, String txt) {
42 1                 int M = pat.length();
43 1                 int N = txt.length();
44
```

```
public int minStickers(String[] stickers, String target) {
        // Optimization 1: Maintain frequency only for characters present in target
        int[] targetNaiveCount = new int[26];
        for (char c : target.toCharArray())
                targetNaiveCount[c - 'a']++;
        int[] index = new int[26];
        int N = 0; // no of distinct characters in target
        for (int i = 0; i < 26; i++)
                index[i] = targetNaiveCount[i] > 0 ? N++ : -1;
        int[] targetCount = new int[N];
        int t = 0;
        for (int c : targetNaiveCount)
                if (c > 0) {
                        targetCount[t++] = c;
                }
        int[][] stickersCount = new int[stickers.length][N];
        for (int i = 0; i < stickers.length; i++) {
                for (char c : stickers[i].toCharArray()) {
                        int j = index[c - 'a'];
                        if (j >= 0)
                                stickersCount[i][j]++;
                }
        }
        // Optimization 2: Remove stickers dominated by some other sticker
        int start = 0;
        for (int i = 0; i < stickers.length; i++) {
                for (int j = start; j < stickers.length; j++)
                        if (j != i) {
                                int k = 0;
                                while (k < N && stickersCount[i][k] <= stickersCount[j][k])
                                        k++;
                                if (k == N) {
                                        int[] tmp = stickersCount[i];
                                        stickersCount[i] = stickersCount[start];
                                        stickersCount[start++] = tmp;
                                        break;
                                }
                        }
        }
        // Perform BFS with target as source and an empty string as destination
        Queue<int[]> Q = new LinkedList<>();
        Set<String> visited = new HashSet<>();
        Q.add(targetCount);
        int steps = 0;
```

Mutations

12	1. removed conditional - replaced equality check with false → KILLED
13	1. replaced int return with 0 for src/App::EditDistance → SURVIVED
14	1. removed conditional - replaced equality check with false → KILLED
15	1. replaced int return with 0 for src/App::EditDistance → KILLED
19	1. removed conditional - replaced equality check with false → SURVIVED
20	1. replaced int return with 0 for src/App::EditDistance → KILLED
24	1. removed call to java/lang/String::charAt → KILLED 2. Replaced integer subtraction with addition → KILLED 3. Replaced integer subtraction with addition → KILLED 4. removed call to java/lang/String::charAt → KILLED 5. removed conditional - replaced equality check with false → KILLED
25	1. Replaced integer subtraction with addition → KILLED 2. Replaced integer subtraction with addition → KILLED 3. replaced int return with 0 for src/App::EditDistance → KILLED 4. removed call to src/App::EditDistance → KILLED
33	1. removed call to src/App::EditDistance → KILLED 2. Replaced integer subtraction with addition → KILLED
34	1. removed call to src/App::EditDistance → KILLED 2. Replaced integer subtraction with addition → KILLED
35	1. removed call to src/App::EditDistance → KILLED 2. Replaced integer subtraction with addition → KILLED 3. Replaced integer subtraction with addition → KILLED
36	1. removed call to java/lang/Math::min → KILLED 2. Replaced integer addition with subtraction → KILLED 3. removed call to java/lang/Math::min → KILLED 4. replaced int return with 0 for src/App::EditDistance → KILLED

Challenges

The mutation test report revealed sections of the code where mutants persisted, prompting us to delve into the algorithms to discern potential test cases capable of eliminating these mutants. Crafting test scenarios that adequately addressed the surviving mutants proved to be a challenging task, particularly in the context of string algorithms with numerous conditional and logical operators. This process served as a valuable lesson in the art of test case design and underscored the critical role of effective testing in thoroughly assessing a program's functionality.

Contributions

We had a long discussion regarding the project before deciding to perform mutation testing on the string-algorithms codebase. Given our mutual familiarity with Java, we chose to write the algorithm implementations in that language. After dividing the functions equally among ourselves, we eventually combined the code. We additionally provided test cases that would eliminate the mutants for each implementation of the function.

1. Analysis and Testcases for Algo 1-14 - Arin Awasthi
2. Analysis and Testcases for Algo 15-28 - Jainav Sanghvi