CS731: Software Testing Final Project

Ishan Shanware IMT2019037
ishan.shanware@iiitb.ac.in

Ghazi Shazan Ahmad IMT2019033
ghazishazan.ahmad@iiitb.ac.in

Abstract

The following report of testcases built for the purpose of mutation testing on a codebase. The codebase consists of several string algorithm functions which perform tasks like string matching, spare search, prefix search, substring search etc. There are 21 string algorithm functions in the Java Codebase. The goal of this project work was to self-learn and understand the practical aspects of testing, through use of open source tools and the test case design strategies learnt in the course. To perform mutation testing we have used the open source tool called PITclipse. PITclipse provides mutation coverage for your Java programs within the Eclipse IDE.

Introduction

The purpose of mutation testing, a sort of white box testing, is to ensure that a software test suite will be able to detect changes by changing specific components of the source code for an application. Modifications made to the software are intended to break the application. To ensure the calibre of a software testing suite, mutation testing is done.

Traditional test coverage (i.e line, statement, branch, etc.) measures only which code is executed by the provided tests. It does not check that your tests are actually able to detect faults in the executed code. It is therefore only able to identify code that is definitely not tested.

Unit tests are frequently performed using mutation testing. All mutated codes are to be found by the software testing. Then, by changing an existing line of code with a different value, mutations might be carried out. For instance, you may add or remove a sentence, change the true or false expression, or change other variables.

Codebase

The codebase contains java implementations of popular string algorithms. The algorithms consist of different pattern matching, string search, as well as dynamic programming strategies. Given below is the list of algorithms implemented in the codebase.

1. Edit distance: Given two strings *str1*, *str2* and a list of operations which could be performed on the *str1*. Find the minimum number of operations required to convert *str1* tp *str2*. The following operations could be performed on *str1*:
   (a) Insertion of a character.
   (b) Removal of a character.
   (c) Replacement of a character.
2. Knuth Morris Pratt algorithm: KMP algorithm is used to find a ”Pattern” in a ”Text”. This algorithm compares character by character from left to right. But whenever a mismatch occurs, it uses a preprocessed table called ”Prefix Table” to skip characters comparison while matching.
3. Rabin Karp Algorithm: Rabin-Karp algorithm is an algorithm used for searching/matching patterns in the text using a hash function. Unlike Naive string matching algorithm, it does not travel through every character in the initial phase rather it filters the characters that do not match and then performs the comparison.
4. Z-algorithm: This algorithm finds all occurrences of a pattern in a text in linear time. Let length of text be n and of pattern be m, then total time taken is O(m + n) with linear space complexity. For applying Z algorithm, we require a string and a pattern that is to be searched. Z-algorithm is an algorithm used for finding occurrences of a pattern in a string. Z-algorithm works by maintaining an auxiliary array called the Z-array. This Z-array stores the length of the longest substring, starting from the current index, that also it’s prefix. This means that each index stores the number of characters, matching the starting characters, starting from this index. This implies that if Z-array has a value k for any index, it means that k characters after this index match the first k characters of the string. This is the fundamental part of Z-algorithm.
5. Longest Common Subsequence: Given two sequences, find the length of longest subsequence present in both of them. A subsequence is a sequence that appears in the same relative order, but not necessarily contiguous. For example, “abc”, “abg”, “bdf”, “aeg”, ‘”acefg”, .. etc are subsequences of “abcdefg”.
6. Longest Substring to form a Palindrome: Given a string S which only contains lowercase alphabets. Find the length of the longest substring of S such that the characters in it can be rearranged to form a palindrome.
7. Longest Valid Paranthesis: Given a string containing just the characters ’(’ and ’)’, find the length of the longest valid (well-formed) parentheses substring. For example, given string ”)()())”, the length of the longest valid parenthesis will be 4.
8. Longest common prefix: longest common prefix string amongst an array of strings. If there is no common prefix, return an empty string . Example: Input: strs = [”flower”,”flow”,”flight”] Output: ”fl”
9. Longest palindromic subsequence: Given a string (say s), find the length of the longest palindromic subsequence of s. Example: Input: s = ”abbab”, Output: 4, Explanation: The

subsequence abba is the longest palindromic subsequence.

10. Manacher's Algorithm: It is used to find the Longest Palindromic Sub-string in any string. This algorithm is required to solve sub-problems of some very hard problems. Manacher's algorithm is used to find the longest palindromic substring in any given string. This algorithm is faster than the brute force approach, as it exploits the idea of a palindrome happening inside another palindrome. Manacher's algorithm is designed to find the palindromic substrings with odd lengths only. To use it for even lengths also, we tweak the input string by inserting the character "#" at the beginning and each alternate position after that (changing "abcaac" to "#a#b#c#a#a#c#"). In the case of an odd length palindrome, the middle character will be a character of the original string, surrounded by "#".

11. Boyer Moore algorithm: The algorithm is very effective for the various pattern-searching problems. It has served as the benchmark for pattern-searching algorithms ever since. Unlike the traditional way of pattern searching where we try to match the two strings in a forward manner, the Boyer-Moore advances the concept by beginning to match the last character of the string to be searched. In this way, the time complexity is reduced significantly. This algorithm takes the backward approach by aligning the pattern string P with text string T and thereafter, comparing the characters from right to left, starting with the rightmost character. It works on the principle that if a mismatched character is found, then there is no use in matching the other characters in the two strings.

12. Sequence Alignment problem: Given as an input two strings, $X = x_1x_2...x_m$, and $Y = y_1y_2...y_m$, output the alignment of the strings, character by character, so that the net penalty is minimized. The penalty is calculated as:
    (a) A penalty of p_{gap} occurs if a gap is inserted between the string.
    (b) A penalty of p_{xy} occurs for mis-matching the characters of X and Y.

13. Wildcard Pattern matching: Given a text and a wildcard pattern, implement wildcard pattern matching algorithm that finds if wildcard pattern is matched with text. The matching should cover the entire text (not partial text). The wildcard pattern can include the characters '?' and '*'
    (a) '?' – matches any single character
    (b) '*' – Matches any sequence of characters (including the empty sequence)

14. Palindrome partitioning: Given a string s, partition s such that every substring of the partition is a palindrome. Find all possible ways of palindromic partitioning. For example, "aba—b—bbabb—a—b—aba" is a palindrome partitioning of "ababbbabbababa". Determine the fewest cuts needed for a palindrome partitioning of a given string. For example, minimum of 3 cuts are needed for "ababbbabbababa". The three cuts are "a—babbbab—b—ababa". If a string is a palindrome, then minimum 0 cuts are needed. If a string of length n containing all different characters, then minimum n-1 cuts are needed.

15. Sparse search: Given a sorted array of strings that is interspersed with empty strings, write a method to find the location of a given string. Input: ball, ["at", "", "", "", "ball", "", "", "car", "", "", "dad", "", ""] Output: 4

16. Longest repeating subsequence: Given a string, find the length of the longest repeating subsequence, such that the two subsequences don't have same string character at the same position, i.e. any ith character in the two subsequences shouldn't have the same index in the original string.
17. Longest Prefix which is also a Suffix: Given a string 'S', find the longest proper prefix that is also a suffix of 'S'. You have to print the size of the longest proper prefix. A string's proper prefix includes all prefixes except the entire string itself. Example: Input: 'S' = xyxyxy Output: 4.
18. Number of distinct words of size N with at most K contiguous vowels: Given two integers N and K, the task is to find the number of distinct strings consisting of lowercase alphabets of length N that can be formed with at-most K contiguous vowels. As the answer may be too large, function returns answer%1000000007.
19. Left and Right rotation of a string: Given a string of size n, write functions to perform the following operations on a string.
    (a) Left (Or anticlockwise) rotate the given string by d elements (where d ¡= n).
    (b) Right (Or clockwise) rotate the given string by d elements (where d ¡= n).
20. Reverse vowels in a given string: Given a string s, reverse only all the vowels in the string and return it. The vowels are 'a', 'e', 'i', 'o', and 'u', and they can appear in both lower and upper cases, more than once.
21. Horspool BM algorithm pattern searching: Horspool BM algorithm is an algorithm for finding substrings into strings. This algorithm compares each characters of substring to find a word or the same characters into the string. When characters do not match, the search jumps to the next matching position in the pattern by the value indicated in the Bad Match Table. The Bad Match Table indicates how many jumps should it move from the current position to the next.

Pitclipse

PIT runs unit tests against automatically modified versions of application code. When the application code changes, it should produce different results and cause the unit tests to fail. If a unit test does not fail in this situation, it may indicate an issue with the test suite. We chose PIT because it is fast, easy to use, actively developed and actively supported.

Pitclipse is a plugin which relies on PIT and runs directly on the Eclipse IDE. The usage of Pitclipse is quite simple;

- Right-click on a Java project defining unit tests
- Run As > PIT Mutation Test

Approach to Testing

We used the following different mutators and applied to the whole codebase.

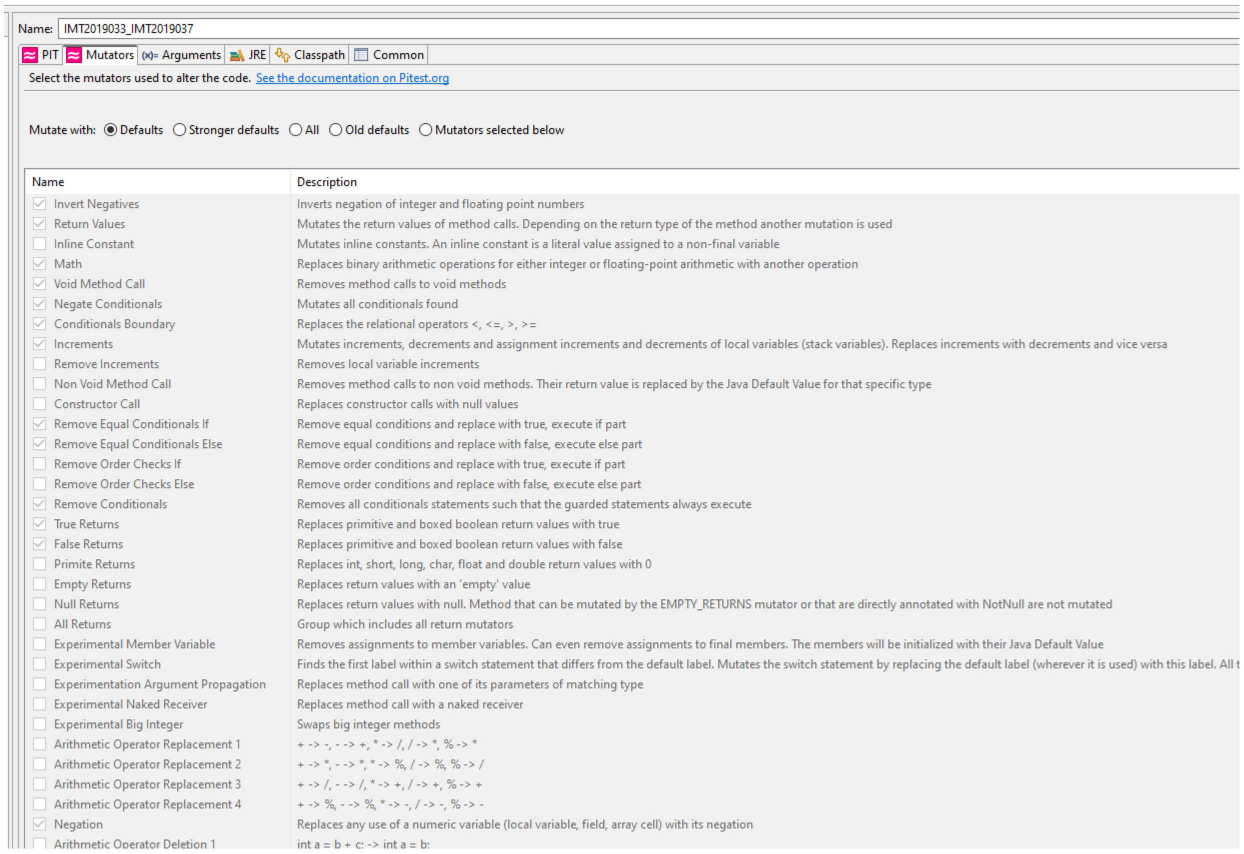
- $BOOLEAN\_FALSE\_RETURN$
- $BOOLEAN\_TRUE\_RETURN$
- $BOOLEAN\_TRUE\_RETURN$
- $CONDITIONALS\_BOUNDARY\_MUTATOR$
- $EMPTY_RETURN\_VALUES$
- $INCREMENTS\_MUTATOR$
- $INVERT\_NEGS\_MUTATOR$
- $MATH\_MUTATOR$
- $NEGATE\_CONDITIONALS\_MUTATOR$
- $NULL\_RETURN\_VALUES$
- $PRIMITIVE\_RETURN\_VALS\_MUTATOR$
- $VOID\_METHOD_CALL\_MUTATOR$

Active mutators

- BOOLEAN_FALSE_RETURN
- BOOLEAN_TRUE_RETURN
- CONDITIONALS_BOUNDARY_MUTATOR
- EMPTY_RETURN_VALUES
- INCREMENTS_MUTATOR
- INVERT_NEGS_MUTATOR
- MATH_MUTATOR
- NEGATE_CONDITIONALS_MUTATOR
- NULL_RETURN_VALUES
- PRIMITIVE_RETURN_VALS_MUTATOR
- VOID_METHOD_CALL_MUTATOR

Tests examined

- src.StringAlgorithmsTest.[engine:junit-vintage]/[runner:src.StringAlgorithmsTest]/[test:TestLongestPrefixSuffix(src.StringAlgorithmsTest)] (11 ms)
- src.StringAlgorithmsTest.[engine:junit-vintage]/[runner:src.StringAlgorithmsTest]/[test:TestLRR(src.StringAlgorithmsTest)] (13 ms)
- src.StringAlgorithmsTest.[engine:junit-vintage]/[runner:src.StringAlgorithmsTest]/[test:TestLongestPalindromicSubsequence(src.StringAlgorithmsTest)] (1669717 ms)
- src.StringAlgorithmsTest.[engine:junit-vintage]/[runner:src.StringAlgorithmsTest]/[test:TestLCS(src.StringAlgorithmsTest)] (12 ms)
- src.StringAlgorithmsTest.[engine:junit-vintage]/[runner:src.StringAlgorithmsTest]/[test:TestLongestCommonPrefix(src.StringAlgorithmsTest)] (13 ms)
- src.StringAlgorithmsTest.[engine:junit-vintage]/[runner:src.StringAlgorithmsTest]/[test:TestSequenceAlignment(src.StringAlgorithmsTest)] (25 ms)
- src.StringAlgorithmsTest.[engine:junit-vintage]/[runner:src.StringAlgorithmsTest]/[test:TestReverseVowel(src.StringAlgorithmsTest)] (10 ms)
- src.StringAlgorithmsTest.[engine:junit-vintage]/[runner:src.StringAlgorithmsTest]/[test:TestKMPAlgorithm(src.StringAlgorithmsTest)] (57 ms)
- src.StringAlgorithmsTest.[engine:junit-vintage]/[runner:src.StringAlgorithmsTest]/[test:TestminPalPartition(src.StringAlgorithmsTest)] (31 ms)
- src.StringAlgorithmsTest.[engine:junit-vintage]/[runner:src.StringAlgorithmsTest]/[test:TestLVP(src.StringAlgorithmsTest)] (11 ms)
- src.StringAlgorithmsTest.[engine:junit-vintage]/[runner:src.StringAlgorithmsTest]/[test:TestEditDistance(src.StringAlgorithmsTest)] (38 ms)
- src.StringAlgorithmsTest.[engine:junit-vintage]/[runner:src.StringAlgorithmsTest]/[test:TestLongestRepeatingSubSeq(src.StringAlgorithmsTest)] (11 ms)
- src.StringAlgorithmsTest.[engine:junit-vintage]/[runner:src.StringAlgorithmsTest]/[test:TestShortestCommonSequence(src.StringAlgorithmsTest)] (13 ms)
- src.StringAlgorithmsTest.[engine:junit-vintage]/[runner:src.StringAlgorithmsTest]/[test:TestBoyerMoore(src.StringAlgorithmsTest)] (29 ms)
- src.StringAlgorithmsTest.[engine:junit-vintage]/[runner:src.StringAlgorithmsTest]/[test:TestRepeatedStringMatch(src.StringAlgorithmsTest)] (11 ms)
- src.StringAlgorithmsTest.[engine:junit-vintage]/[runner:src.StringAlgorithmsTest]/[test:TestRabinKarp(src.StringAlgorithmsTest)] (11 ms)
- src.StringAlgorithmsTest.[engine:junit-vintage]/[runner:src.StringAlgorithmsTest]/[test:TestWildcardPattern(src.StringAlgorithmsTest)] (16 ms)
- src.StringAlgorithmsTest.[engine:junit-vintage]/[runner:src.StringAlgorithmsTest]/[test:TestZAlgorithm(src.StringAlgorithmsTest)] (28 ms)
- src.StringAlgorithmsTest.[engine:junit-vintage]/[runner:src.StringAlgorithmsTest]/[test:TestKVowelWords(src.StringAlgorithmsTest)] (21 ms)
- src.StringAlgorithmsTest.[engine:junit-vintage]/[runner:src.StringAlgorithmsTest]/[test:TestSparseSearch(src.StringAlgorithmsTest)] (11 ms)
- src.StringAlgorithmsTest.[engine:junit-vintage]/[runner:src.StringAlgorithmsTest]/[test:TestManachersAlgorithm(src.StringAlgorithmsTest)] (10 ms)
- src.StringAlgorithmsTest.[engine:junit-vintage]/[runner:src.StringAlgorithmsTest]/[test:TestLSFP(src.StringAlgorithmsTest)] (12 ms)



Name: IMT2019033_IMT2019037

PIT | Mutators | Arguments | JRE | Classpath | Common

Select the mutators used to alter the code. See the documentation on Pitest.org

Mutate with: [✓ Defaults] Stronger defaults All Old defaults Mutators selected below

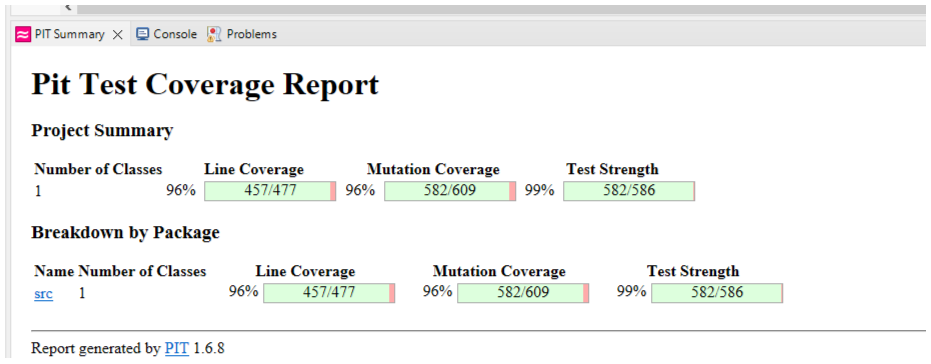
	Name	Description
✓	Invert Negatives	Inverts negation of integer and floating point numbers
✓	Return Values	Mutates the return values of method calls. Depending on the return type of the method another mutation is used
	Inline Constant	Mutates inline constants. An inline constant is a literal value assigned to a non-final variable
✓	Math	Replaces binary arithmetic operations for either integer or floating-point arithmetic with another operation
✓	Void Method Call	Removes method calls to void methods
✓	Negate Conditionals	Mutates all conditionals found
✓	Conditionals Boundary	Replaces the relational operators <, <=, >, >=
✓	Increments	Mutates increments, decrements and assignment increments and decrements of local variables (stack variables). Replaces increments with decrements and vice versa
	Remove Increments	Removes local variable increments
	Non Void Method Call	Removes method calls to non void methods. Their return value is replaced by the Java Default Value for that specific type
	Constructor Call	Replaces constructor calls with null values
✓	Remove Equal Conditionals If	Remove equal conditions and replace with true, execute if part
✓	Remove Equal Conditionals Else	Remove equal conditions and replace with false, execute else part
	Remove Order Checks If	Remove order conditions and replace with true, execute if part
	Remove Order Checks Else	Remove order conditions and replace with false, execute else part
✓	Remove Conditionals	Removes all conditionals statements such that the guarded statements always execute
✓	True Returns	Replaces primitive and boxed boolean return values with true
✓	False Returns	Replaces primitive and boxed boolean return values with false
	Primite Returns	Replaces int, short, long, char, float and double return values with 0
	Empty Returns	Replaces return values with an 'empty' value
	Null Returns	Replaces return values with null. Method that can be mutated by the EMPTY_RETURNS mutator or that are directly annotated with NotNull are not mutated
	All Returns	Group which includes all return mutators
	Experimental Member Variable	Removes assignments to member variables. Can even remove assignments to final members. The members will be initialized with their Java Default Value
	Experimental Switch	Finds the first label within a switch statement that differs from the default label. Mutates the switch statement by replacing the default label (wherever it is used) with this label. All t
	Experimentation Argument Propagation	Replaces method call with one of its parameters of matching type
	Experimental Naked Receiver	Replaces method call with a naked receiver
	Experimental Big Integer	Swaps big integer methods
	Arithmetic Operator Replacement 1	+ -> -, - -> +, * -> /, / -> *, % -> *
	Arithmetic Operator Replacement 2	+ -> *, - -> *, * -> %, / -> %, % -> /
	Arithmetic Operator Replacement 3	+ -> /, - -> /, * -> +, / -> +, % -> +
	Arithmetic Operator Replacement 4	+ -> %, - -> %, * -> -, / -> -, % -> -
✓	Negation	Replaces any use of a numeric variable (local variable, field, array cell) with its negation
	Arithmetic Operator Deletion 1	int a = b + c; -> int a = b;

Results

Our final test cases design gave us very good results. Initially, our line coverage and mutation coverage was 80% and 69% respectively.

We also experimented by changing the mutators and trying out different combinations of mutators. On average, our mutation score varied between 75% and 80%. The conditional boundary constraint was the most challenging mutator and was difficult to kill them.

We tried to improve this mutation coverage by analysing the mutants that survived and then designed test cases to improve the mutation coverage percentage. We looked at the lines which were not covered by the test cases, boundary values and negative conditional constraints were the major mutators. We added new test cases and also tried to randomize some of the test case generations to improve the coverage.


PIT Summary Console Problems

Pit Test Coverage Report

Project Summary

Number of Classes	Line Coverage	Mutation Coverage	Test Strength
1	96% 457/477	96% 582/609	99% 582/586

Breakdown by Package

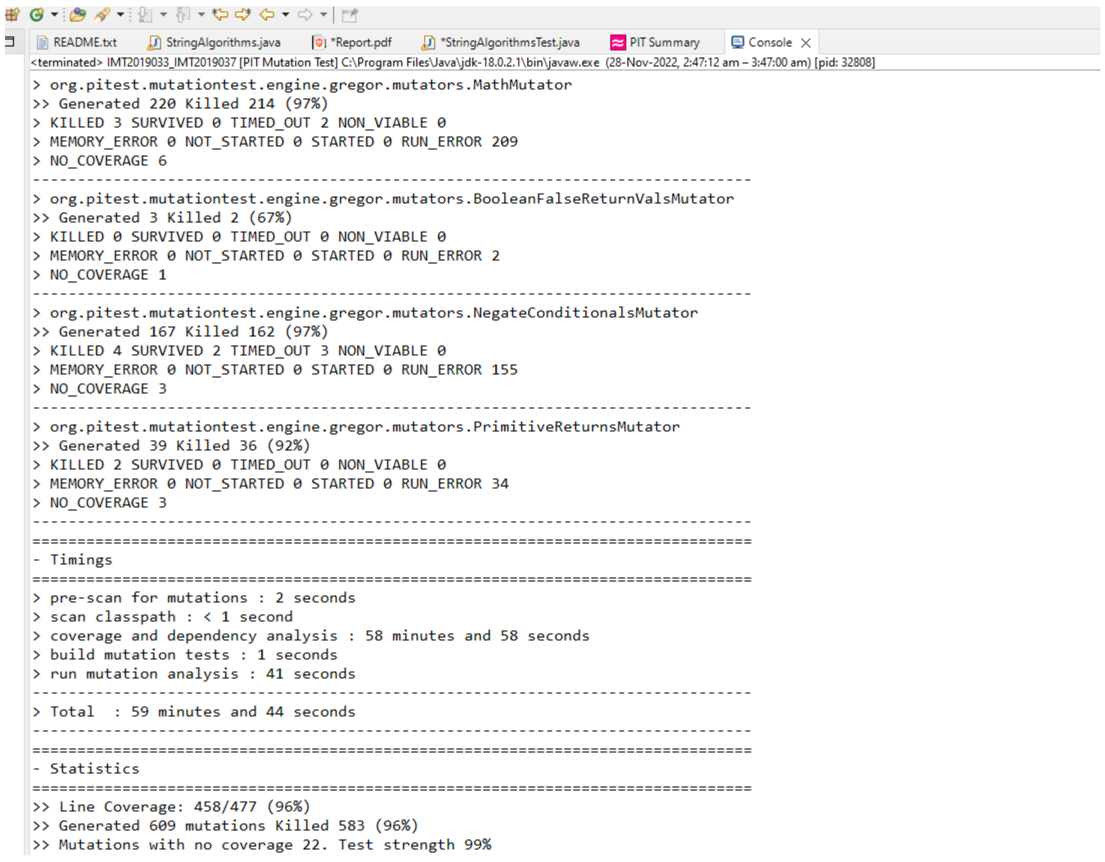
Name	Number of Classes	Line Coverage	Mutation Coverage	Test Strength
src	1	96% 457/477	96% 582/609	99% 582/586

Report generated by PIT 1.6.8

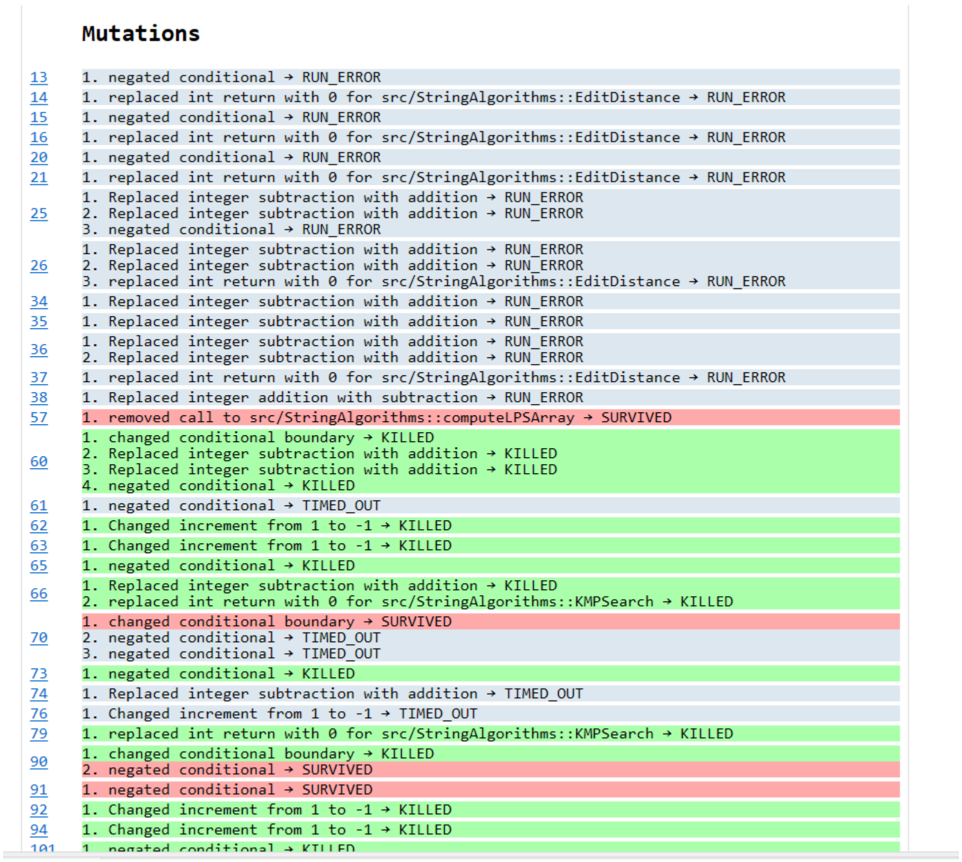
Finally, after analyzing the PIT Summary, we were able to achieve a mutation coverage of 96% and line coverage of 96%, and the mutation strength was 99% which is quite remarkable.

Challenges

The mutation test report showed us lines of code where the mutants survived. Using this information, we started understanding the algorithms to identify test cases that might kill the mutants. It was difficult to design test cases which would cover the mutants which had survived. As string algorithms have a lot of conditional and logical operators, it was a challenging to map out test cases. This exercise taught us how to design test cases and the importance of effective test cases in testing a program.



```
<terminated> IMT2019033_IMT2019037 [PIT Mutation Test] C:\Program Files\Java\jdk-18.0.2.1\bin\javaw.exe (28-Nov-2022, 2:47:12 am – 3:47:00 am) [pid: 32808]
> org.pitest.mutationtest.engine.gregor.mutators.MathMutator
>> Generated 220 Killed 214 (97%)
> KILLED 3 SURVIVED 0 TIMED_OUT 2 NON_VIABLE 0
> MEMORY_ERROR 0 NOT_STARTED 0 STARTED 0 RUN_ERROR 209
> NO_COVERAGE 6
--------------------------------------------------------------------------------
> org.pitest.mutationtest.engine.gregor.mutators.BooleanFalseReturnValsMutator
>> Generated 3 Killed 2 (67%)
> KILLED 0 SURVIVED 0 TIMED_OUT 0 NON_VIABLE 0
> MEMORY_ERROR 0 NOT_STARTED 0 STARTED 0 RUN_ERROR 2
> NO_COVERAGE 1
--------------------------------------------------------------------------------
> org.pitest.mutationtest.engine.gregor.mutators.NegateConditionalsMutator
>> Generated 167 Killed 162 (97%)
> KILLED 4 SURVIVED 2 TIMED_OUT 3 NON_VIABLE 0
> MEMORY_ERROR 0 NOT_STARTED 0 STARTED 0 RUN_ERROR 155
> NO_COVERAGE 3
--------------------------------------------------------------------------------
> org.pitest.mutationtest.engine.gregor.mutators.PrimitiveReturnsMutator
>> Generated 39 Killed 36 (92%)
> KILLED 2 SURVIVED 0 TIMED_OUT 0 NON_VIABLE 0
> MEMORY_ERROR 0 NOT_STARTED 0 STARTED 0 RUN_ERROR 34
> NO_COVERAGE 3
--------------------------------------------------------------------------------
================================================================================
- Timings
================================================================================
> pre-scan for mutations : 2 seconds
> scan classpath : < 1 second
> coverage and dependency analysis : 58 minutes and 58 seconds
> build mutation tests : 1 seconds
> run mutation analysis : 41 seconds
--------------------------------------------------------------------------------
> Total  : 59 minutes and 44 seconds
--------------------------------------------------------------------------------
================================================================================
- Statistics
================================================================================
>> Line Coverage: 458/477 (96%)
>> Generated 609 mutations Killed 583 (96%)
>> Mutations with no coverage 22. Test strength 99%
```



```
Mutations

13   1. negated conditional → RUN_ERROR
14   1. replaced int return with 0 for src/StringAlgorithms::EditDistance → RUN_ERROR
15   1. negated conditional → RUN_ERROR
16   1. replaced int return with 0 for src/StringAlgorithms::EditDistance → RUN_ERROR
20   1. negated conditional → RUN_ERROR
21   1. replaced int return with 0 for src/StringAlgorithms::EditDistance → RUN_ERROR
     1. Replaced integer subtraction with addition → RUN_ERROR
25   2. Replaced integer subtraction with addition → RUN_ERROR
     3. negated conditional → RUN_ERROR
     1. Replaced integer subtraction with addition → RUN_ERROR
26   2. Replaced integer subtraction with addition → RUN_ERROR
     3. replaced int return with 0 for src/StringAlgorithms::EditDistance → RUN_ERROR
34   1. Replaced integer subtraction with addition → RUN_ERROR
35   1. Replaced integer subtraction with addition → RUN_ERROR
36   1. Replaced integer subtraction with addition → RUN_ERROR
     2. Replaced integer subtraction with addition → RUN_ERROR
37   1. replaced int return with 0 for src/StringAlgorithms::EditDistance → RUN_ERROR
38   1. Replaced integer addition with subtraction → RUN_ERROR
57   1. removed call to src/StringAlgorithms::computeLPSArray → SURVIVED
     1. changed conditional boundary → KILLED
60   2. Replaced integer subtraction with addition → KILLED
     3. Replaced integer subtraction with addition → KILLED
     4. negated conditional → KILLED
61   1. negated conditional → TIMED_OUT
62   1. Changed increment from 1 to -1 → KILLED
63   1. Changed increment from 1 to -1 → KILLED
65   1. negated conditional → KILLED
66   1. Replaced integer subtraction with addition → KILLED
     2. replaced int return with 0 for src/StringAlgorithms::KMPSearch → KILLED
     1. changed conditional boundary → SURVIVED
70   2. negated conditional → TIMED_OUT
     3. negated conditional → TIMED_OUT
73   1. negated conditional → KILLED
74   1. Replaced integer subtraction with addition → TIMED_OUT
76   1. Changed increment from 1 to -1 → TIMED_OUT
79   1. replaced int return with 0 for src/StringAlgorithms::KMPSearch → KILLED
90   1. changed conditional boundary → KILLED
     2. negated conditional → SURVIVED
91   1. negated conditional → SURVIVED
92   1. Changed increment from 1 to -1 → KILLED
94   1. Changed increment from 1 to -1 → KILLED
101  1. negated conditional → KILLED
```

Contributions

For the project, we had elaborate discussions before we decided to do mutation testing on the string-algorithms codebase. Since both of us are familiar with java, we decided to code the algorithm implementations in java. We divided the functions equally between us, and then we merged the code later. For each function implementation, we also provided test cases that would kill the mutants.

1. Analysis and TestCases for Pattern Searching: Ghazi
2. Analysis and TestCases for String DP problems: Ishan.

Once, we ran the PITest, we together designed a test case of complex algorithms like the Z-algorithm and Horspool algorithms to improve mutation coverage. We also picked some popular string-based problems like edit distance and wildcard-matching which use dynamic programming.

References

- `https://github.com/pitest/pitclipse`
- `https://pitest.org/`

- https://dev.to/silviobuss/increase-the-quality-of-unit-tests-using-mutation-with-pitest-3b27
- https://www.researchgate.net/publication/310773886_Mutation_Testing_Techniques_A_Comparative_Study
- https://junit.org/junit5/
- https://github.com/topics/string-algorithms