Mutation Testing of Java Project Using PIT

Team MutationTesters

November 26, 2024

Abstract

This report presents the mutation testing of Java project codebase involving several solver classes implementing algorithms for standard computational problems. The objective is to evaluate the effectiveness of test cases in detecting faults introduced through mutation operators using the PIT (Pitest) framework.

1 Introduction

Mutation testing is a fault-based software testing technique used to assess the quality of test cases. By introducing small changes (mutations) to the code and observing if the existing tests can detect these changes, we can measure the effectiveness of the test suite.

The underlying principle is that a strong test suite should fail when the program is altered in any way that introduces faults. Therefore, if tests detect and fail due to these mutations, it indicates that they are effective at catching errors. Conversely, mutations that survive (i.e., do not cause test failures) reveal weaknesses in the test suite, highlighting areas where additional or more rigorous testing is needed. When a test case detects a mutant or causes it to fail or differ from the original code, then we say the mutant has been killed by the test case.

This report details the mutation testing process applied to a Java project comprising multiple solver classes, each solving a different algorithmic problem.

2 Codebase

The codebase consists of a main class, `Main.java`, and several (eleven) solver classes (`Solver1` to `Solver11`), each implementing the `Solver` interface. Each solver class addresses a specific common computational problem sourced from LeetCode or GeeksforGeeks. The problems include algorithms like topological sorting of Direct Acyclic Graph, Maximum Path Sum of a Binary Tree, Single Source Shortest Path (SSSP) problem, and others. Test cases are written using JUnit to verify the correctness of each solver. The Java project codebase contains roughly 1200+ lines of code.

3 Mutation Operators

Mutation testing was conducted using the PIT (Pitest) framework, which applies various mutation operators to the codebase to introduce small changes (mutants) and assess the

effectiveness of the test suite. The operators target different aspects of the code, such as conditional statements, arithmetic operations, and return values. The purpose is primarily unit testing, focusing on the internal logic of methods and functions to ensure robustness against faults.

The following mutation operators were used in the testing process:

3.1 Conditionals Boundary Mutator

This operator modifies the boundaries of conditional statements by changing relational operators (e.g., `<` to `<=`, `>` to `>=`). Its purpose is to test the correctness of conditional logic and ensure that tests cover boundary conditions effectively.

- **Mutations Generated**: 54
- **Mutations Killed**: 47 (87%)
- **Survived Mutations**: 7
- **Timed Out Mutations**: 1

3.2 Primitive Returns Mutator

This operator changes the return values of methods that return primitive types (e.g., replacing a return value with zero). It tests whether the test suite can detect incorrect return values from methods.

- **Mutations Generated**: 8
- **Mutations Killed**: 8 (100%)
- **Survived Mutations**: 0

3.3 Increments Mutator

This operator alters increment and decrement operations (e.g., changing `i++` to `i--`). It checks the test suite's ability to catch errors in loop counters and other arithmetic operations.

- **Mutations Generated**: 9
- **Mutations Killed**: 9 (100%)
- **Survived Mutations**: 0
- **Timed Out Mutations**: 2

3.4 Void Method Call Mutator

This operator removes calls to void methods, testing whether the test suite can detect the absence of side effects that these methods are supposed to produce.

- **Mutations Generated**: 70
- **Mutations Killed**: 67 (96%)
- **Survived Mutations**: 3

3.5 Boolean True Return Values Mutator

This operator changes boolean return values to `true`. It tests the test suite's ability to handle incorrect boolean logic.

- **Mutations Generated**: 1
- **Mutations Killed**: 1 (100%)
- **Survived Mutations**: 0

3.6 Null Return Values Mutator

This operator changes return values to `null`, checking if the test suite can handle unexpected null values and prevent `NullPointerException`s.

- **Mutations Generated**: 2
- **Mutations Killed**: 2 (100%)
- **Survived Mutations**: 0

3.7 Math Mutator

This operator alters mathematical operations (e.g., changing addition to subtraction). It tests whether the test suite can detect incorrect calculations.

- **Mutations Generated**: 42
- **Mutations Killed**: 42 (100%)
- **Survived Mutations**: 0
- **Timed Out Mutations**: 3

3.8 Empty Object Return Values Mutator

This operator changes return values to empty objects, such as empty arrays or collections. It checks if the test suite can handle and detect incorrect empty return values.

- **Mutations Generated**: 1
- **Mutations Killed**: 1 (100%)
- **Survived Mutations**: 0

3.9 Negate Conditionals Mutator

This operator negates logical conditions in control flow statements (e.g., changing `if (a > b)` to `if (a <= b)`). It is essential for testing the test suite's ability to detect incorrect branching logic.

- **Mutations Generated**: 106
- **Mutations Killed**: 105 (99%)
- **Survived Mutations**: 1
- **Timed Out Mutations**: 1
- **Memory Errors**: 1

3.10 Remove Method Call Mutator

This operator removes method calls between classes, focusing on the integration points within the application. It tests whether the integration tests can detect the absence of expected interactions between components. By simulating scenarios where method calls are omitted, we can verify the robustness of the system's architecture and the effectiveness of the integration tests.

- **Mutations Generated**: 10
- **Mutations Killed**: 9 (90%)
- **Survived Mutations**: 1

Summary of Mutation Testing Results

Overall, a total of 303 mutations were generated across all operators. The test suite was able to kill 291 mutations, resulting in a mutation score of approximately 96%. The surviving mutations indicate areas where the test suite could be improved, potentially by adding more comprehensive tests or addressing equivalent mutants that cannot be detected through testing.

Testing Approach

We applied mutation operators to both unit and integration tests, targeting individual methods and class interactions. By introducing faults at various codebase levels, we assessed the effectiveness of test cases in detecting functionality deviations. The high mutation scores across most operators indicate robust test coverage at multiple levels. Integration testing validated interactions between solve function and its helper function. The **Remove Method Call Mutator**, which removes method calls between classes, simulated failures at integration points. This helped assess whether integration tests could detect missing interactions and maintain overall functionality.

Mutation Operator	Generated	Killed	Mutation Score
Conditionals Boundary Mutator	54	47	87%
Primitive Returns Mutator	8	8	100%
Increments Mutator	9	9	100%
Void Method Call Mutator	70	67	96%
Boolean True Return Values Mutator	1	1	100%
Null Return Values Mutator	2	2	100%
Math Mutator	42	42	100%
Empty Object Return Values Mutator	1	1	100%
Negate Conditionals Mutator	106	105	99%
Remove Method Call Mutator	10	9	90%
Total	**293**	**282**	**96%**

Table 1: Mutation Testing Results by Operator

4 Test Case Design

Test cases were designed to cover a wide range of inputs, including edge cases and typical scenarios for each solver. For instance:

- **Solver1 (Topological Sort)**: Tested with acyclic graphs, graphs with multiple valid orders, no edges, and large graphs.
- **Solver8 (Binary Tree Maximum Path Sum)**: Tested balanced/skewed trees, trees with negative values, and single-node trees.
- **Solver9 (Word Ladder)**: Covered valid transformations, no-path cases, varying word list lengths, and identical start/end words.
- **Solver10 (Number of Islands)**: Tested grids with multiple islands, no land, single large islands, and varying dimensions.
- **Solver11 (Dijkstra's Algorithm)**: Checked graphs with positive weights, zero-weight edges, disconnected nodes, and multiple shortest paths.

The test cases aim to ensure that any mutation affecting the code logic would result in a test failure, thereby killing the mutant.

5 Team Members:

The team name as mentioned in the title section is MutationTesters. The team members are:

1. Kadaru Jashwanth Reddy - IMT2021095
2. Adithya Sunil - IMT2021068

6 Contributions

Broadly the contributions include:

1. Developed comprehensive large code base in Java, with a Main class, Solver interface, and 11 Solver classes. Collected hard computational problems focusing graphs, trees, binary, search and arrays.
2. Configured the PIT mutation testing framework for the project, including adjusting timeouts and mutators. Finally, we chose to not include timeouts, because, it was causing unexpected behaviour.
3. Analyzed mutation testing results to identify weaknesses in the test suite and improve test coverage and add additional testcases.
4. Documented the testing process and results, providing insights into the effectiveness of mutation testing in project.

All the above things were divided as per problems. For each problem, the member responsible implemented the Solver class methods, and all test cases. The problems were divided as follows:

- Jashwanth Kadaru : $[1, 2, 3, 4, 5, 11]$
- Adithya Sunilkumar : $[6, 7, 8, 9, 10]$

7 Results

The mutation testing process generated a total of 293 mutants, of which 282 were killed, resulting in a mutation score of 96%. Line coverage achieved was 510 out of 514 lines, corresponding to 99% line coverage. All mutations were covered by the test suite, with no uncovered mutants. A total of 499 test cases were executed, averaging 1.7 tests per mutation. These results demonstrate the extent of coverage and mutation detection achieved through the applied test cases.

Table 2: Mutation Testing Overall Statistics

Metric	**Value**
Line Coverage	510/514 (99%)
Mutations Generated	293
Mutations Killed	282 (96%)
Mutations with No Coverage	0
Test Strength	96%
Total Test Cases	161
Total Tests Executed	575
Tests per Mutation	1.96

8 Console Output of Mutation Coverage

Below are the console outputs from the Maven mutation coverage tool, showcasing the statistics and mutator details.

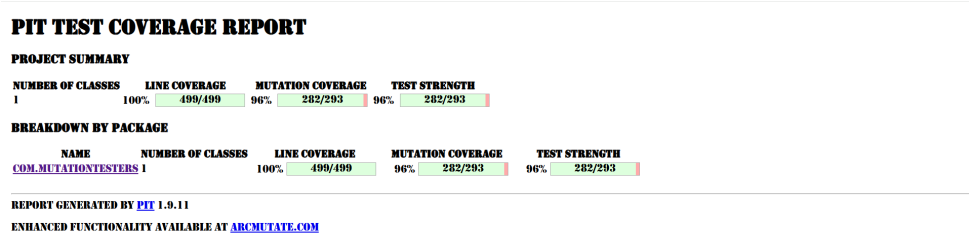
Figure 1: Part 1: Mutation Coverage Statistics and Mutator Details (1/2)

9 & PIT Report of Mutation Coverage

Below are the screenshots of PIT report (target/pit-reports/index.html) from the Maven mutation coverage tool, showcasing the statistics and mutator details.

```
> org.pitest.mutationtest.engine.gregor.mutators.returns.EmptyObjectReturnValsMutator
>> Generated 1 Killed 1 (100%)
> KILLED 1 SURVIVED 0 TIMED_OUT 0 NON_VIABLE 0
> MEMORY_ERROR 0 NOT_STARTED 0 STARTED 0 RUN_ERROR 0
> NO_COVERAGE 0
--------------------------------------------------------------------------------
> org.pitest.mutationtest.engine.gregor.mutators.NegateConditionalsMutator
>> Generated 106 Killed 105 (99%)
> KILLED 103 SURVIVED 1 TIMED_OUT 1 NON_VIABLE 0
> MEMORY_ERROR 1 NOT_STARTED 0 STARTED 0 RUN_ERROR 0
> NO_COVERAGE 0
--------------------------------------------------------------------------------
================================================================================
- Timings
================================================================================
> pre-scan for mutations : < 1 second
> scan classpath : < 1 second
> coverage and dependency analysis : < 1 second
> build mutation tests : < 1 second
> run mutation analysis : 36 seconds
--------------------------------------------------------------------------------
> Total  : 37 seconds
--------------------------------------------------------------------------------
================================================================================
- Statistics
================================================================================
>> Line Coverage: 510/514 (99%)
>> Generated 293 mutations Killed 282 (96%)
>> Mutations with no coverage 0. Test strength 96%
>> Ran 499 tests (1.7 tests per mutation)
Enhanced functionality available at https://www.arcmutate.com/
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  40.365 s
[INFO] Finished at: 2024-11-26T19:43:32+05:30
[INFO] ------------------------------------------------------------------------
PS C:\Users\91939\Desktop\ST\Final Project\mutationtestersfinalproject>
```

Figure 2: Part 2: Mutation Coverage Statistics and Mutator Details (2/2)



PIT TEST COVERAGE REPORT

PROJECT SUMMARY

NUMBER OF CLASSES	LINE COVERAGE		MUTATION COVERAGE		TEST STRENGTH	
1	100%	499/499	96%	282/293	96%	282/293

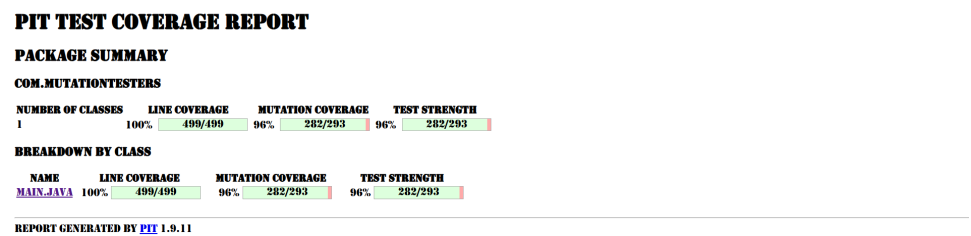
BREAKDOWN BY PACKAGE

NAME	NUMBER OF CLASSES	LINE COVERAGE		MUTATION COVERAGE		TEST STRENGTH	
COM.MUTATIONTESTERS	1	100%	499/499	96%	282/293	96%	282/293

REPORT GENERATED BY PIT 1.9.11

ENHANCED FUNCTIONALITY AVAILABLE AT ARCMUTATE.COM

Figure 3: Part 1: Mutation Coverage Statistics and Mutator Details (1/2)



PIT TEST COVERAGE REPORT

PACKAGE SUMMARY

COM.MUTATIONTESTERS

NUMBER OF CLASSES	LINE COVERAGE		MUTATION COVERAGE		TEST STRENGTH	
1	100%	499/499	96%	282/293	96%	282/293

BREAKDOWN BY CLASS

NAME	LINE COVERAGE		MUTATION COVERAGE		TEST STRENGTH	
MAIN.JAVA	100%	499/499	96%	282/293	96%	282/293

REPORT GENERATED BY PIT 1.9.11

Figure 4: Part 2: Mutation Coverage Statistics and Mutator Details (2/2)