Project Report

By Prajit Banerjee(MT2023004) under Dr Meenakshi D'Souza

guided by Anshul Jindal(TA).

Abstract

This report presents an analysis of mutation testing performed on Java code functions using PITclipse, a plugin designed to execute mutation testing tools against existing unit test cases. The document provides a comprehensive explanation of the methodology applied, the functions utilized in the process, and a detailed overview of the PIT tool and its outcomes.

Introduction

Software testing involves assessing and verifying that a software application behaves as intended. It provides numerous benefits, including bug detection, reduced development costs, and enhanced performance.

Mutation testing is a white-box testing approach that examines the internal structure of the code, such as data structures, logic, and overall design. Unlike black-box testing, which focuses on functionality, mutation testing dives deeper into the code to evaluate its quality and robustness.

In mutation testing, specific changes, or mutations, are introduced into the source code to create altered versions called mutants. These changes simulate potential errors in the code, and the objective is to ensure that the test cases can detect these errors. A mutant is considered "killed" if the test cases fail when executed against the altered code, proving their effectiveness. If both the original and mutant code produce identical outputs, the mutant is "survived," indicating potential gaps in the test cases. For instance, mutations might involve altering operators (e.g., changing < to >). The efficiency of the test suite is quantified through the mutation score, which reflects the percentage of killed mutants. The ultimate aim of mutation testing is to strengthen test cases so that they can effectively identify issues in modified code.

Mutation testing verifies the coverage and robustness of a test suite, uncovering subtle bugs and ambiguities in the code. However, manually creating mutants is complex and time-intensive. To simplify this process, automation tools are employed. In this study, we utilized PITclipse to generate mutants, apply mutations, and evaluate the strength of our test cases systematically.

How Mutation Testing is Done

Mutation testing is a software testing technique used to evaluate the effectiveness of a test suite by introducing small changes, or "mutants," into the source code to simulate potential bugs. These mutants are generated by modifying operators, conditions, or return values within the code. The test suite is then executed against the mutants, and its strength is determined based on whether the tests can detect and fail due to these changes. If a mutant causes a test case to fail, it is considered "killed," indicating the test suite's effectiveness. Surviving mutants highlight gaps in the test coverage, prompting improvements to the tests. The quality of the test suite is measured using the mutation score, calculated as the percentage of killed mutants. Mutation testing ensures robust test coverage by uncovering subtle defects and enhancing the overall reliability of the software. Tools like PIT and MuJava automate this process, simplifying mutant generation and evaluation.

Algorithms on which Mutation Testing is done

1. Bellman Ford algorithm.
2. BFS with Unit Weight.
3. Finding whether a graph is bipartite or not.
4. Bridge finding algorithm.
5. Dijkstra algorithm.
6. Find total number of components using DSU.
7. Graph Traversal algorithm(DFS and BFS).
8. Kosaraju’s algorithm.
9. Loop Detection algorithm.
10. Prims algorithm.

11.Topological Sorting.

Mutation Testing Tool used: PIT

PIT (short for **PITest**) is a powerful, open-source mutation testing tool designed for Java applications. It evaluates the quality of unit tests by introducing small modifications, or "mutants," into the code and checking if the test suite detects these changes. By identifying gaps in test coverage, PIT ensures the robustness and reliability of software testing practices. It supports popular testing frameworks like JUnit and TestNG and seamlessly integrates with build tools like Maven and Gradle, making it easy to incorporate into continuous integration workflows. PIT's efficiency lies in its selective mutation strategy, which targets only the most impactful parts of the code, reducing the computational overhead compared to traditional mutation testing tools. It provides detailed reports highlighting survived and killed mutants, helping developers improve their test cases and achieve higher mutation scores.

Results

Pit Test Coverage Report

Package Summary

org.example

Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
13	97%	448/461	77%	257/335	81%	257/316

Breakdown by Class

Name	Line Coverage		Mutation Coverage		Test Strength	
BFSUnitWeight.java	100%	23/23	90%	9/10	90%	9/10
BellmanFord.java	92%	11/12	78%	14/18	78%	14/18
BipartiteGraph.java	100%	30/30	91%	20/22	91%	20/22
BridgeFinding.java	100%	33/33	85%	17/20	85%	17/20
DSUComponents.java	97%	34/35	50%	10/20	50%	10/20
DijkstraAlgorithm.java	100%	28/28	71%	10/14	71%	10/14
FloydWarshall.java	75%	21/28	73%	29/40	100%	29/29
GraphTraversal.java	100%	42/42	93%	14/15	93%	14/15
KosarajuAlgorithm.java	100%	44/44	96%	26/27	96%	26/27
Kruskal.java	96%	47/49	54%	22/41	59%	22/37
LoopDetection.java	100%	50/50	85%	35/41	85%	35/41
Prims.java	95%	39/41	57%	20/35	65%	20/31
TopologicalSort.java	100%	46/46	97%	31/32	97%	31/32

Report generated by PIT 1.9.11

The **PIT Test Coverage Report** provides an analysis of the testing effectiveness for the Java project under the `org.example` package. The project consists of 13 classes, achieving an overall **Line Coverage** of 97% (448 out of 461 lines covered), **Mutation Coverage** of 77% (257 out of 335 mutants killed), and an overall **Test Strength** of 81% (257 out of 316).

A class-wise breakdown reveals that several classes, such as `BFSUnitWeight.java` and `TopologicalSort.java`, demonstrate excellent test coverage, with 100% line coverage and high mutation coverage at 90% and 97%, respectively. Similarly, classes like `BipartiteGraph.java` and `KosarajuAlgorithm.java` maintain robust mutation coverage above 90%. However, areas for improvement are evident in classes like `Kruskal.java` and `Prims.java`, which show lower mutation coverage of 54% and 65%, respectively, despite achieving high line coverage.

This report highlights the project's overall testing robustness while identifying specific areas where the test suite can be improved to enhance reliability and mutation resilience.

References

https://www.geeksforgeeks.org/

https://www.javatpoint.com/

https://pitest.org/

Thank You