Mutation Testing of Java Data Structures using PITest

Software testing is vital and plays a critical role in ensuring program correctness, reliability, and maintainability. Traditionally, testing techniques often rely on structural coverage metric**s**, such as line or branch coverage, to estimate test quality. These metrics are used to indicate which parts of the program were executed, but they do not guarantee that the test suite can detect faults. A test suite may execute every line of code yet fail to assert correct behavior.

Mutation testing addresses this limitation by evaluating the fault detection capability of a test suite rather than its execution coverage. I revolve this project around applying mutation testing using **PITest**, a mutation testing framework for Java, to real world use. I looked at the data structure implementations sourced from an open-source GitHub repository. The goal is to compare mutation testing with line coverage, analyze results across a few programs, and study how mutation testing scales with program size and complexity.

1. Background

1.1 Line Coverage

Line coverage is a structural testing metric that measures the percentile of executable lines run during testing. It is easy to compute and is widely used, but it has a crucial critical weakness; executing a line does not imply that incorrect behavior would be detected. For example, a conditional statement may be exercised only in one direction, leaving boundary faults undetected.

1.2 Mutation Testing

Mutation testing evaluates test adequacy by introducing small syntactic changes, called **mutants**, into the program. These mutants simulate common programmer mistakes, such as:

- Changing conditional boundaries
- Replacing arithmetic operators
- Altering return values
- Negating logical expressions

If a test suite fails when run against a mutant, that mutant is considered to be *killed*. If the test suite passes, the mutant *survives*. This is how we indicate a weakness in the tests. Mutation coverage is the percentage of mutants killed by the test suite. Mutation testing is considered a **stronger adequacy criterion** than coverage because it directly evaluates fault detection rather than mere execution.

2. Repository Selection and Initial Setup

To ensure the analysis was realistic and meaningful, I searched GitHub for a repository containing; Pure Java implementations, algorithmic or data structure code, Rich control flow (branches, loops, conditions), Self-contained modules/classes suitable for isolated testing

2.1 Selection

I selected the *William Fiset / Algorithms* repository, which is widely used for education and interview preparation. It contains a variety of data structures implemented with nontrivial logic.

2.2 Setup

After cloning the repository, I imported it into a JDE using VScode and inspected the directory structure to identify suitable classes for mutation testing.

3. Programs

3.1First Program: Union-Find

The **Union-Find** data structure was selected as the first program under test because it:

- Implements path compression and union-by-size
- Contains multiple loops and conditionals
- Has subtle correctness properties
- Is compact (~107 lines of core logic)

These characteristics make Union-Find ideal for mutation testing, as small mutations can easily introduce incorrect behavior.

3.2 Test Development

JUnit 5 tests were written to verify:

- Connectivity between elements

- Correct merging of components
- Proper tracking of component size
- Edge cases involving invalid input

The goal was to design tests that asserted *correct behavior*, not simply execution.

4. Mutation Testing with PITest

4.1 Tool Installation and Configuration

Mutation testing was performed using **PITest**, integrated via the Gradle build system. The Gradle plugin allowed automated mutation generation, execution, and reporting.
In the configuration worked on:

- Targeting specific packages
- Enabling JUnit 5 support
- Restricting mutation scope to the class under test
- Generating HTML reports

4.2 Running into Environment Limitations

The experiments were initially run on the university's **Potter Box** environment. While PITest executed correctly, the generated HTML report could not be viewed directly due to environment restrictions on launching browsers or opening files.
To resolve this, the report directory was transferred to my local machine, where the HTML output could be viewed in a browser. This step shows that mutation testing tools often require specific approaches to access to fully interpret results.

5. Results: Union-Find

The Union-Find has the following results:

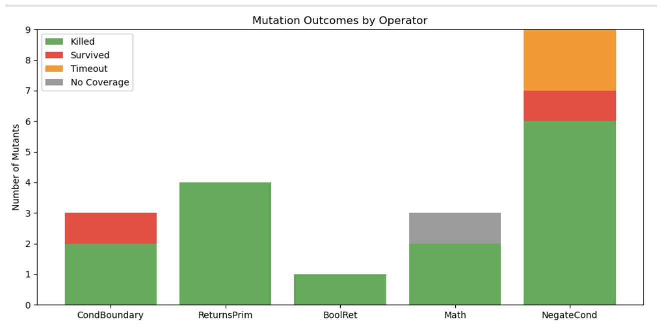
- Line Coverage: 91%
- Mutation Coverage: 83%
- Test Strength: 88%

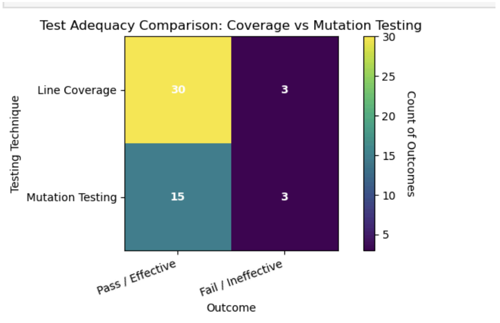
Despite high line coverage, some mutants survived—especially those related to conditional boundaries and control-flow negations. This confirmed that line coverage alone overestimates test adequacy.

6. Visualization and Interpretation

To better understand results, the numerical output from PITest was converted into visualizations using Python. Heatmaps and stacked bar charts were generated to show:

- Coverage vs mutation effectiveness
- Mutation operators vs outcomes
- Distribution of killed, survived, timed-out, and uncovered mutant





These visualizations made the inadequacy of coverage-only metrics immediately visible and helped identify operator-specific weaknesses in the test suite.

7.1 Extending the Evaluation: Larger Program

The Union-Find class alone did not satisfy the project's line-of-code requirement. To evaluate scalability and test mutation testing on a larger codebase, I had to look for a good second program. I initially explored stack and queue implementations but it was confusing and was limited additional logic. As a result, the **DoublyLinkedList** implementation was selected.

7.2 DoublyLinkedList

The DoublyLinkedList implementation contains approximately **300 lines of code** and includes; Bidirectional node traversal, multiple insertion and removal paths, index-based and object-based operations, extensive branching logic

JUnit tests were written to validate:

- Insertions at head, tail, and arbitrary positions
- Removals from head, tail, and middle
- Index bounds checking
- Empty list behavior

8. Comparative Results

Mutation testing on DoublyLinkedList revealed results similar to Union-Find with high line coverage, slightly lower mutation coverage, surviving mutants primarily related to boundary conditions

Despite increased program size, mutation scores remained consistent, suggesting that test adequacy scaled reasonably with code complexity.

9. Future Work

Possible extensions include:

If time permitted, I would have loved to keep developing past the checkpoint to do things like; Strengthening boundary and conditional tests, applying mutation testing to additional data structures, and comparing mutation testing results with coverage-only tools like JaCoCo. Writing the programs myself was a bit hard and took a while