Software Testing Report

M Srinivasan [1*†] and P Ramsai Koushik[2†]

[1]IMT2021058, IIIT-Bangalore, Bangalore, India.
[2]IMT2021072, IIIT-Bangalore, Bangalore, India.
[†]These authors contributed equally to this work.

1 Introduction

Mutation testing is a fault-based testing technique aimed at evaluating the quality of test cases by introducing artificial faults (mutations) into the code and checking whether the test cases can detect these faults. This document explores the fundamentals of mutation testing, its process, types of mutants, applications, and challenges. It also contains the evaluation results of applying PiTest in our Java Code for Graph Based problems from Leetcode.

Software testing is an essential phase of the software development lifecycle (SDLC), ensuring the correctness, reliability, and robustness of software systems. Traditional testing techniques primarily focus on verifying expected outputs for given inputs. However, these methods often fail to measure the effectiveness of test cases. Mutation testing addresses this gap by systematically introducing small changes to the program's code, known as *mutants*, to assess the capability of test suites to identify defects.

2 Concepts in Mutation Testing

Mutation testing is built on the premise of the **Competent Programmer Hypothesis**, which assumes that programmers write code that is "close to correct." Hence, faults introduced are likely minor, and test cases designed to detect them are expected to catch real-world defects.

2.1 Key Terminologies

- **Mutant**: A version of the original program with a small syntactic change.
- **Mutation Operator**: A rule for creating mutants. Examples include altering arithmetic operators (e.g., replacing `+` with `-`), logical operators (e.g., replacing `||` with `&&`), or relational operators (e.g., replacing `>` with `>=`).
- **Killed Mutant**: A mutant that is detected by a test case (i.e., the test case produces different outputs for the mutant and the original program).

- **Survived Mutant**: A mutant that is not detected by any test case, indicating potential weaknesses in the test suite.
- **Mutation Score**: The ratio of killed mutants to the total number of non-equivalent mutants, representing the effectiveness of the test suite.
- **Strong Killing** A mutant is said to be *strongly killed* if the mutation causes a visible effect on the program's behavior, and this effect is detected by the test suite through a failing test case. Mathematically, this can be expressed as:

$$\text{Strong Killing (S)} = \begin{cases} 1 & \text{if Mutation} \Rightarrow \text{Output Mismatch} \land \text{Test Failure}, \\ 0 & \text{otherwise.} \end{cases}$$

- **Weak Killing** A mutant is *weakly killed* if the mutation introduces a change in the program's internal state (e.g., a variable or a control flow) but does not propagate to cause an observable output mismatch or a test failure. This is defined mathematically as:

$$\text{Weak Killing (W)} = \begin{cases} 1 & \text{if Mutation} \Rightarrow \text{State Change} \land \neg\text{Output Mismatch}, \\ 0 & \text{otherwise.} \end{cases}$$

3 Pitest

Pitest is a mutation testing tool for Java, designed to assess the quality of unit tests by introducing small modifications (mutations) into the codebase and evaluating if the test suite detects them. The primary goal of mutation testing is to ensure the robustness and effectiveness of test cases.

Pitest operates by:

- Identifying code sections to mutate, such as arithmetic operators, conditional statements, or method calls.
- Generating mutants by introducing changes (e.g., replacing `+` with `-`, or changing `true` to `false`).
- Running the test suite against these mutants.

A mutant is said to be *killed* if the tests fail when the mutation is present, indicating that the test suite effectively captures the introduced faults. Conversely, a *survived* mutant implies a gap in the test coverage.

The metrics generated by Pitest include:

- **Mutation Coverage**: The percentage of mutants killed by the test suite.
- **Mutant Status**: A detailed report of killed and survived mutants.

Pitest integrates with popular build tools like Maven and Gradle, making it suitable for continuous integration workflows. Its ability to uncover subtle weaknesses in tests makes it an essential tool for developers aiming to enhance code reliability.

4 Graph Class and Algorithms

The `Graph` class is a comprehensive implementation of various graph algorithms and their applications in solving standard Data Structures and Algorithms (DSA) problems. This section outlines the key methods provided by the `Graph` class and their functionality.

4.1 Class Overview

- **Breadth-First Search (BFS)**: A graph traversal algorithm that explores vertices in the order of their distance from the source, using a queue to traverse level by level. It is often used to find the shortest path in an unweighted graph.
- **Depth-First Search (DFS)**: A graph traversal algorithm that explores as far as possible along each branch before backtracking. It uses a stack (explicit or implicit via recursion) and is useful for tasks like finding connected components or detecting cycles.
- **Graph Connectivity and Bipartite Checking**: Methods to determine whether a graph is connected (single component) and to check if the graph is bipartite, which means its vertices can be colored with two colors such that no two adjacent vertices share the same color.
- **Topological Sorting**: A linear ordering of vertices in a Directed Acyclic Graph (DAG), such that for every directed edge (u, v), vertex u appears before v. Useful for tasks like scheduling.
- **Cycle Detection in Graphs**: Algorithms to detect cycles in directed or undirected graphs, often using DFS or Union-Find methods. Cycle detection is critical for understanding graph structure and constraints.
- **Dijkstra's Algorithm**: A shortest-path algorithm for graphs with non-negative weights. It uses a priority queue to iteratively find the shortest path from a source node to all other nodes in the graph.
- **Bellman-Ford Algorithm**: A shortest-path algorithm that works on graphs with negative weights, detecting negative weight cycles if they exist. It uses relaxation to update distances from the source node.
- **Floyd-Warshall Algorithm**: An all-pairs shortest-path algorithm that computes the shortest paths between all pairs of nodes in a weighted graph. It uses dynamic programming to iteratively improve the shortest paths.
- **Disjoint Set Union (DSU)**: A data structure for managing and merging disjoint sets. It supports union and find operations and is commonly used in algorithms like Kruskal's for Minimum Spanning Tree.

The source code incorporates efficient data structures such as adjacency lists, queues, and sets for optimal performance. There is considerable potential for applying mutation testing to this code.

5 Mutation Operators Producing Valid Mutants

5.1 Unit-Level Mutation Operators

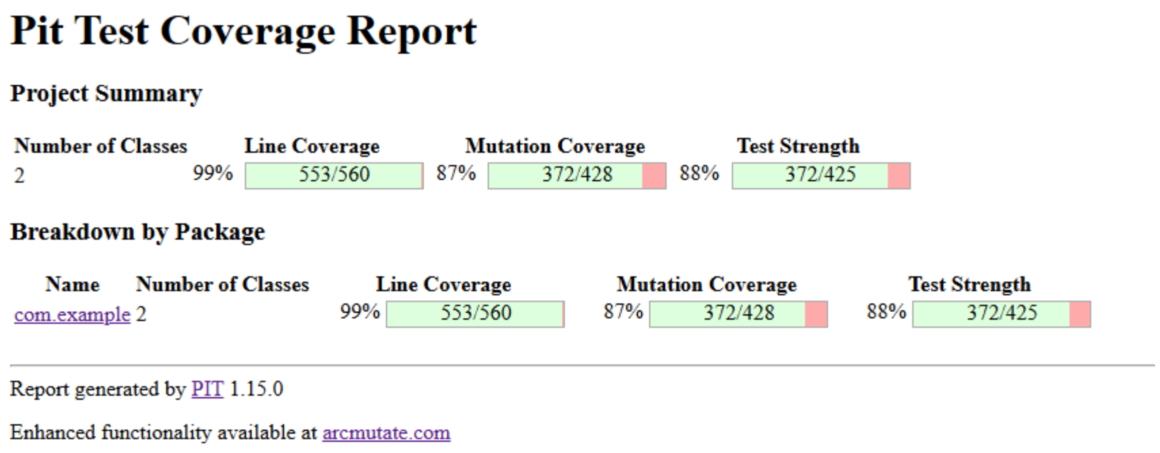
- **ConditionalsBoundaryMutator**: Alters conditional boundaries, such as changing $<$ to $\leq$ or vice versa. This tests edge-case handling in conditional statements.
- **IncrementsMutator**: Modifies increment and decrement operations, such as changing "i++" to "i–". This ensures loop or arithmetic correctness.
- **MathMutator**: Mutates mathematical operations, e.g., changing addition to subtraction, or multiplication to division. This ensures resilience to arithmetic errors.
- **NegateConditionalsMutator**: Negates logical conditions in "if" statements and loops, such as replacing if (a > b) with if ($a \leq b$). This validates logic correctness in decision-making.

5.2 Integration-Level Mutation Operators

- **PrimitiveReturnsMutator**: Mutates the return values of primitive types, e.g., replacing a return of "1" with "0" or other default values. This tests how effectively return values are used in logic.
- **VoidMethodCallMutator**: Removes calls to void methods, verifying if these calls are essential for program correctness.
- **BooleanTrueReturnValsMutator**: Changes boolean return values to always return "true". This evaluates robustness to unexpected logic behavior.
- **NullReturnValsMutator**: Changes object return values to "null", testing how the code handles null references.
- **EmptyObjectReturnValsMutator**: Modifies return values of collections or objects to return an empty value, such as an empty list or array. This checks how well code handles empty inputs.
- **BooleanFalseReturnValsMutator**: Modifies boolean return values to always return "false", testing robustness to unexpected false outcomes.

6 Test Case Design Strategy

To ensure correctness of the algorithms implemented in the `Graph` class, we followed a systematic test case design strategy. This section outlines the approach we adopted to create effective test cases, tailored to the unique requirements of graph algorithms, and discusses how we used mutation testing to refine our testing process.



Pit Test Coverage Report

Project Summary

Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
2	99%	553/560	87%	372/428	88%	372/425

Breakdown by Package

Name	Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
com.example	2	99%	553/560	87%	372/428	88%	372/425

Report generated by PIT 1.15.0

Enhanced functionality available at arcmutate.com

Figure 1: The Evaluation of the PIT result in our code.

6.1 Initial Test Case Design

Our initial set of test cases was designed to cover:

- **Basic Functionality:** Test cases were created to validate the correctness of fundamental operations such as graph traversal, pathfinding, and graph property checks.
- **Common Scenarios:** Typical scenarios like fully connected graphs, sparse graphs, and disconnected graphs were tested to ensure general applicability.

6.2 Mutation Testing and Coverage Analysis

After executing the initial test cases, we performed mutation testing to evaluate their effectiveness. Mutation operators were applied to simulate potential errors in the code, and the test cases were executed to identify mutants that survived. The mutation testing revealed a coverage of 83%, indicating areas where additional test cases were required to kill surviving mutants.

6.3 Refined Test Case Design

To enhance mutation coverage and address surviving mutants, we adopted the following refined test case design principles:

- **Handling Edge Cases:** Test cases were designed for scenarios like empty graphs, graphs with a single node or edge, and graphs with disconnected components. These ensured the algorithms handled degenerate inputs gracefully.
- **Targeting Key Mutations:** Specific mutants, such as those created by removing sort function calls in Kruskal's algorithm, were addressed with test cases that validated correct Minimum Spanning Tree (MST) generation even with unordered edges. Boolean-returning functions were also tested thoroughly to ensure both 'true' and 'false' outcomes were validated.

- **Boundary and Special Conditions:** Test cases included extreme conditions, such as graphs with cycles of odd lengths to test bipartite detection, and varied path scenarios to validate algorithms like `surroundingRegions`.

After refining the test cases, we achieved a mutation coverage of 87%. Here are some specific reasons why achieving more than 87% mutation coverage was challenging:

- **Matrix Traversal Mutant:** When performing BFS on a matrix, directional offsets such as $(1, 0), (-1, 0)$, etc., are added to the current coordinates to determine the next move. If a mathematical mutation operator changes addition to subtraction in these offsets, the algorithm's functionality remains unaffected, leaving the mutant undetected.
- **Minimum Spanning Tree Mutant:** In MST construction, we terminate the process upon forming a tree with $n - 1$ edges. If a mutation changes this condition to $n + 1$, the algorithm continues without breaking early. However, the final correctness of the MST remains intact since the edge comparison to $n - 1$ is primarily for optimization, not correctness validation.
- **Disjoint Set Union Mutant:** In the `union` function of a disjoint set, the smaller tree is typically attached to the bigger tree to optimize the height of the resulting tree. However, if a mutation alters this condition and the bigger tree is instead attached to the smaller tree, it does not affect the correctness of the algorithm. The change only impacts the efficiency of tree height reduction but not the final result of the union operation.

These scenarios illustrate cases where mutants can survive despite robust test coverage, as the changes do not influence the algorithm's correctness or output.

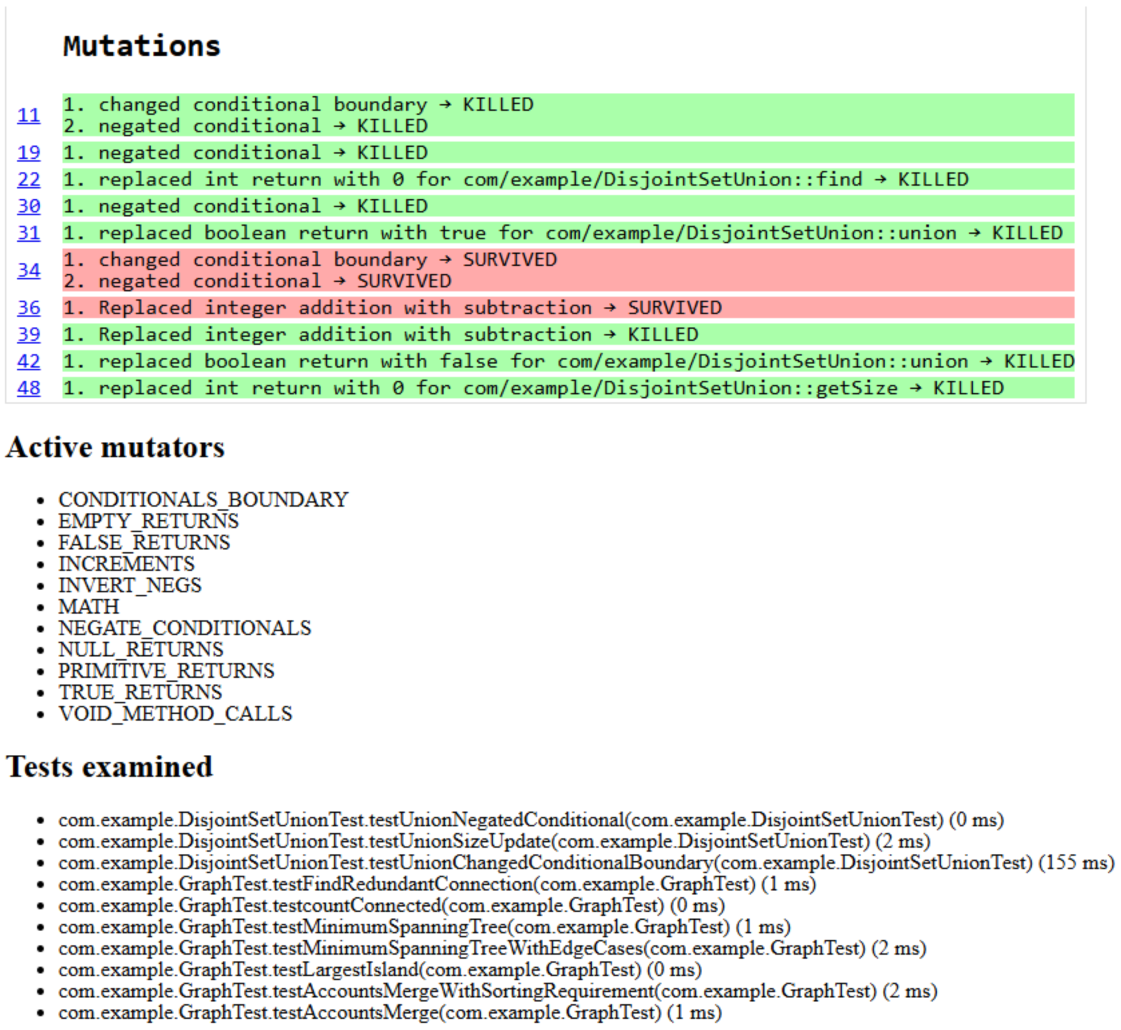
7 Use of Automation Tool for Mutation Testing

To evaluate the effectiveness of our manually designed test cases, we utilized **PIT Mutation Testing** as the automation tool. The process involved integrating PIT into our Maven project and running mutation tests on the `Graph` class and its associated algorithms. PIT systematically introduced mutations (changes) into the code and executed the test cases to determine how many of these mutants were effectively *killed* by the existing test suite.

This automated approach provided the following benefits:

- **Assessment of Test Case Strength:** It measured the capability of the test cases to identify faults in the code by analyzing killed vs. survived mutants.
- **Identification of Weak Spots:** Surviving mutants revealed areas where test cases needed improvement, enabling targeted refinements to increase coverage and robustness.
- **Comprehensive Feedback:** The PIT reports highlighted mutation operators used, the number of mutants generated, and the mutation coverage achieved.

By leveraging PIT, we ensured that our manually designed test cases were systematically validated and strengthened, meeting the project's requirements for automation in testing.



Mutations

```
11  1. changed conditional boundary → KILLED
    2. negated conditional → KILLED
19  1. negated conditional → KILLED
22  1. replaced int return with 0 for com/example/DisjointSetUnion::find → KILLED
30  1. negated conditional → KILLED
31  1. replaced boolean return with true for com/example/DisjointSetUnion::union → KILLED
34  1. changed conditional boundary → SURVIVED
    2. negated conditional → SURVIVED
36  1. Replaced integer addition with subtraction → SURVIVED
39  1. Replaced integer addition with subtraction → KILLED
42  1. replaced boolean return with false for com/example/DisjointSetUnion::union → KILLED
48  1. replaced int return with 0 for com/example/DisjointSetUnion::getSize → KILLED
```

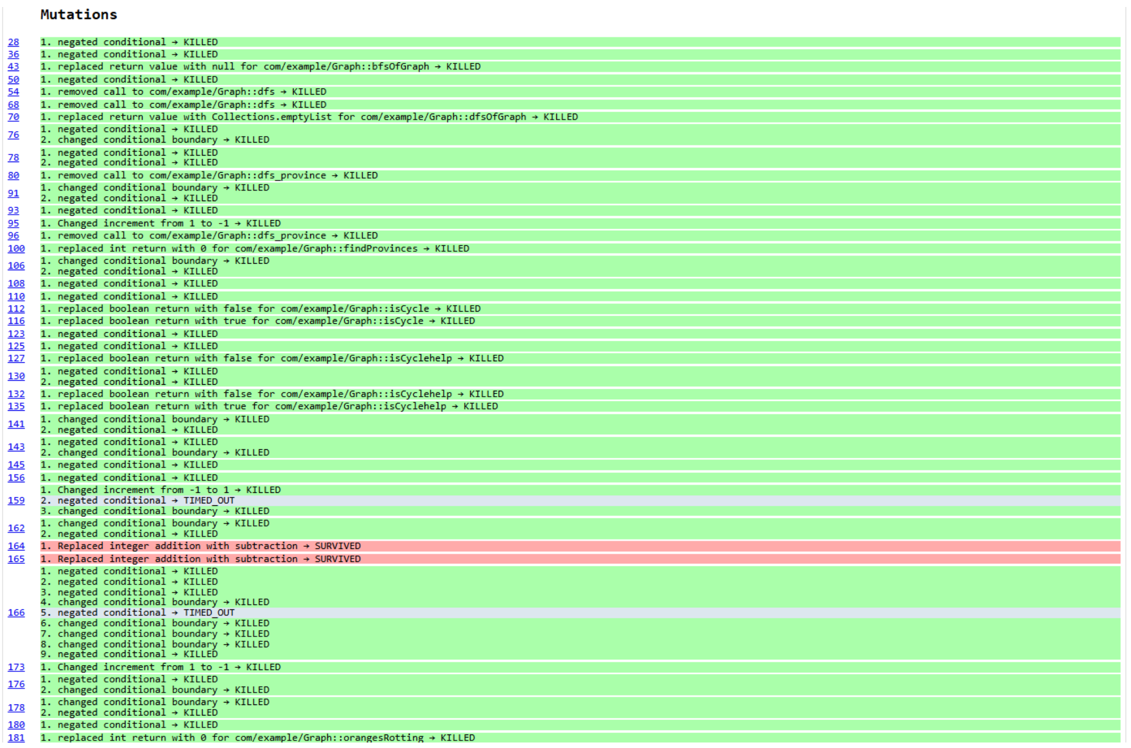
Active mutators

- CONDITIONALS_BOUNDARY
- EMPTY_RETURNS
- FALSE_RETURNS
- INCREMENTS
- INVERT_NEGS
- MATH
- NEGATE_CONDITIONALS
- NULL_RETURNS
- PRIMITIVE_RETURNS
- TRUE_RETURNS
- VOID_METHOD_CALLS

Tests examined

- com.example.DisjointSetUnionTest.testUnionNegatedConditional(com.example.DisjointSetUnionTest) (0 ms)
- com.example.DisjointSetUnionTest.testUnionSizeUpdate(com.example.DisjointSetUnionTest) (2 ms)
- com.example.DisjointSetUnionTest.testUnionChangedConditionalBoundary(com.example.DisjointSetUnionTest) (155 ms)
- com.example.GraphTest.testFindRedundantConnection(com.example.GraphTest) (1 ms)
- com.example.GraphTest.testcountConnected(com.example.GraphTest) (0 ms)
- com.example.GraphTest.testMinimumSpanningTree(com.example.GraphTest) (1 ms)
- com.example.GraphTest.testMinimumSpanningTreeWithEdgeCases(com.example.GraphTest) (2 ms)
- com.example.GraphTest.testLargestIsland(com.example.GraphTest) (0 ms)
- com.example.GraphTest.testAccountsMergeWithSortingRequirement(com.example.GraphTest) (2 ms)
- com.example.GraphTest.testAccountsMerge(com.example.GraphTest) (1 ms)

Figure 2: Pitest Output for the DSU class code.



```
Mutations

28   1. negated conditional → KILLED
36   1. negated conditional → KILLED
43   1. replaced return value with null for com/example/Graph::bfsOfGraph → KILLED
50   1. negated conditional → KILLED
54   1. removed call to com/example/Graph::dfs → KILLED
68   1. removed call to com/example/Graph::dfs → KILLED
70   1. replaced return value with Collections.emptyList for com/example/Graph::dfsOfGraph → KILLED
76   1. negated conditional → KILLED
     2. changed conditional boundary → KILLED
78   1. negated conditional → KILLED
     2. negated conditional → KILLED
80   1. removed call to com/example/Graph::dfs_province → KILLED
91   1. changed conditional boundary → KILLED
     2. negated conditional → KILLED
93   1. negated conditional → KILLED
95   1. Changed increment from 1 to -1 → KILLED
96   1. removed call to com/example/Graph::dfs_province → KILLED
100  1. replaced int return with 0 for com/example/Graph::findProvinces → KILLED
106  1. changed conditional boundary → KILLED
     2. negated conditional → KILLED
108  1. negated conditional → KILLED
110  1. negated conditional → KILLED
112  1. replaced boolean return with false for com/example/Graph::isCycle → KILLED
116  1. replaced boolean return with true for com/example/Graph::isCycle → KILLED
123  1. negated conditional → KILLED
125  1. negated conditional → KILLED
127  1. replaced boolean return with false for com/example/Graph::isCyclehelp → KILLED
130  1. negated conditional → KILLED
     2. negated conditional → KILLED
132  1. replaced boolean return with false for com/example/Graph::isCyclehelp → KILLED
135  1. replaced boolean return with true for com/example/Graph::isCyclehelp → KILLED
141  1. changed conditional boundary → KILLED
     2. negated conditional → KILLED
143  1. negated conditional → KILLED
     2. changed conditional boundary → KILLED
145  1. negated conditional → KILLED
156  1. negated conditional → KILLED
     1. Changed increment from -1 to 1 → KILLED
159  2. negated conditional → TIMED_OUT
     3. changed conditional boundary → KILLED
162  1. changed conditional boundary → KILLED
     2. negated conditional → KILLED
164  1. Replaced integer addition with subtraction → SURVIVED
165  1. Replaced integer addition with subtraction → SURVIVED
     1. negated conditional → KILLED
     2. negated conditional → KILLED
     3. negated conditional → KILLED
     4. changed conditional boundary → KILLED
166  5. negated conditional → TIMED_OUT
     6. changed conditional boundary → KILLED
     7. changed conditional boundary → KILLED
     8. changed conditional boundary → KILLED
     9. negated conditional → KILLED
173  1. Changed increment from 1 to -1 → KILLED
176  1. negated conditional → KILLED
     2. changed conditional boundary → KILLED
178  1. changed conditional boundary → KILLED
     2. negated conditional → KILLED
180  1. negated conditional → KILLED
181  1. replaced int return with 0 for com/example/Graph::orangesRotting → KILLED
```

Figure 3: Pitest Output for the Graph class code

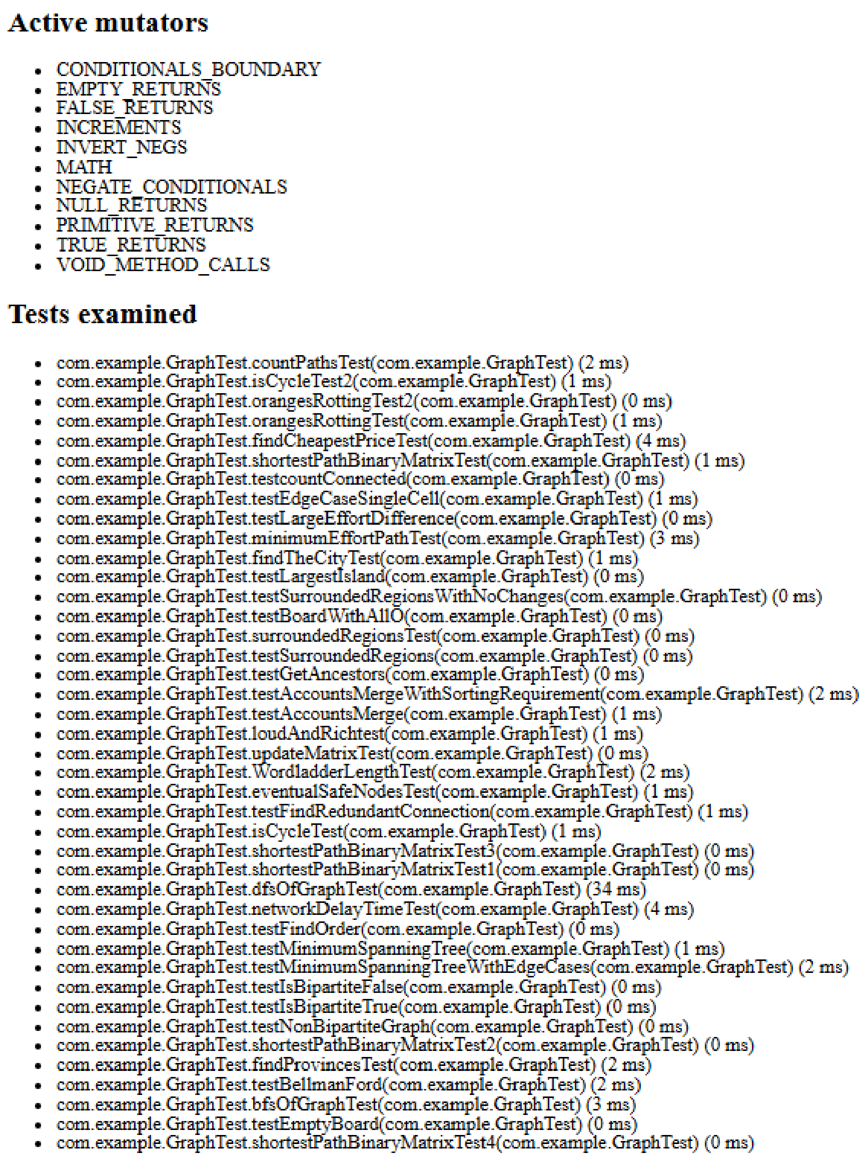
8 Contributions

The development and testing of the `Graph` class were collaboratively carried out, with contributions divided between the team members based on specific algorithms and functionalities. Each team member focused on implementing and testing our assigned subset of algorithms to ensure correctness and robustness. The contributions are detailed below:

8.1 Srinivasan's Contributions

Srinivasan was responsible for implementing and testing the following algorithms and functionalities:

- **Breadth-First Search (BFS):** Implemented BFS for graph traversal and verified its utility in finding shortest paths in unweighted graphs.
- **Depth-First Search (DFS):** Developed DFS for graph traversal, with a focus on applications like connected components and cycle detection.
- **Graph Connectivity and Bipartite Checking:** Designed methods to verify graph connectivity and determine whether a graph is bipartite.
- **Topological Sorting:** Implemented a method for topological sorting in Directed Acyclic Graphs (DAGs), ensuring correct linear ordering of vertices.



Active mutators

- CONDITIONALS_BOUNDARY
- EMPTY_RETURNS
- FALSE_RETURNS
- INCREMENTS
- INVERT_NEGS
- MATH
- NEGATE_CONDITIONALS
- NULL_RETURNS
- PRIMITIVE_RETURNS
- TRUE_RETURNS
- VOID_METHOD_CALLS

Tests examined

- com.example.GraphTest.countPathsTest(com.example.GraphTest) (2 ms)
- com.example.GraphTest.isCycleTest2(com.example.GraphTest) (1 ms)
- com.example.GraphTest.orangesRottingTest2(com.example.GraphTest) (0 ms)
- com.example.GraphTest.orangesRottingTest(com.example.GraphTest) (1 ms)
- com.example.GraphTest.findCheapestPriceTest(com.example.GraphTest) (4 ms)
- com.example.GraphTest.shortestPathBinaryMatrixTest(com.example.GraphTest) (1 ms)
- com.example.GraphTest.testcountConnected(com.example.GraphTest) (0 ms)
- com.example.GraphTest.testEdgeCaseSingleCell(com.example.GraphTest) (1 ms)
- com.example.GraphTest.testLargeEffortDifference(com.example.GraphTest) (0 ms)
- com.example.GraphTest.minimumEffortPathTest(com.example.GraphTest) (3 ms)
- com.example.GraphTest.findTheCityTest(com.example.GraphTest) (1 ms)
- com.example.GraphTest.testLargestIsland(com.example.GraphTest) (0 ms)
- com.example.GraphTest.testSurroundedRegionsWithNoChanges(com.example.GraphTest) (0 ms)
- com.example.GraphTest.testBoardWithAllO(com.example.GraphTest) (0 ms)
- com.example.GraphTest.surroundedRegionsTest(com.example.GraphTest) (0 ms)
- com.example.GraphTest.testSurroundedRegions(com.example.GraphTest) (0 ms)
- com.example.GraphTest.testGetAncestors(com.example.GraphTest) (0 ms)
- com.example.GraphTest.testAccountsMergeWithSortingRequirement(com.example.GraphTest) (2 ms)
- com.example.GraphTest.testAccountsMerge(com.example.GraphTest) (1 ms)
- com.example.GraphTest.loudAndRichtest(com.example.GraphTest) (1 ms)
- com.example.GraphTest.updateMatrixTest(com.example.GraphTest) (0 ms)
- com.example.GraphTest.WordladderLengthTest(com.example.GraphTest) (2 ms)
- com.example.GraphTest.eventualSafeNodesTest(com.example.GraphTest) (1 ms)
- com.example.GraphTest.testFindRedundantConnection(com.example.GraphTest) (1 ms)
- com.example.GraphTest.isCycleTest(com.example.GraphTest) (1 ms)
- com.example.GraphTest.shortestPathBinaryMatrixTest3(com.example.GraphTest) (0 ms)
- com.example.GraphTest.shortestPathBinaryMatrixTest1(com.example.GraphTest) (0 ms)
- com.example.GraphTest.dfsOfGraphTest(com.example.GraphTest) (34 ms)
- com.example.GraphTest.networkDelayTimeTest(com.example.GraphTest) (4 ms)
- com.example.GraphTest.testFindOrder(com.example.GraphTest) (0 ms)
- com.example.GraphTest.testMinimumSpanningTree(com.example.GraphTest) (1 ms)
- com.example.GraphTest.testMinimumSpanningTreeWithEdgeCases(com.example.GraphTest) (2 ms)
- com.example.GraphTest.testIsBipartiteFalse(com.example.GraphTest) (0 ms)
- com.example.GraphTest.testIsBipartiteTrue(com.example.GraphTest) (0 ms)
- com.example.GraphTest.testNonBipartiteGraph(com.example.GraphTest) (0 ms)
- com.example.GraphTest.shortestPathBinaryMatrixTest2(com.example.GraphTest) (0 ms)
- com.example.GraphTest.findProvincesTest(com.example.GraphTest) (2 ms)
- com.example.GraphTest.testBellmanFord(com.example.GraphTest) (2 ms)
- com.example.GraphTest.bfsOfGraphTest(com.example.GraphTest) (3 ms)
- com.example.GraphTest.testEmptyBoard(com.example.GraphTest) (0 ms)
- com.example.GraphTest.shortestPathBinaryMatrixTest4(com.example.GraphTest) (0 ms)

Figure 4: Pitest mutators for Graph Class code.

- **Cycle Detection in Graphs:** Developed and tested algorithms for detecting cycles in both directed and undirected graphs using DFS and Union-Find methods.

8.2 Koushik's Contributions

Koushik worked on implementing and testing the following algorithms and functionalities:

- **Dijkstra's Algorithm:** Implemented Dijkstra's shortest-path algorithm, ensuring efficiency and correctness for graphs with non-negative edge weights.
- **Bellman-Ford Algorithm:** Developed and tested the Bellman-Ford algorithm, including the detection of negative weight cycles.
- **Floyd-Warshall Algorithm:** Designed the Floyd-Warshall algorithm for computing all-pairs shortest paths in weighted graphs and verified its correctness.
- **Disjoint Set Union (DSU):** Implemented the DSU data structure with efficient union and find operations, validating its application in Kruskal's MST algorithm.

9 Coverage Statistics

Mutation testing statistics are detailed in Table 1, showcasing the effectiveness of our refined test cases.

Metric	**Value**
Line Coverage (Mutated Classes)	553/560 (99%)
Generated Mutations	428
Killed Mutations	372 (87%)
Mutations with No Coverage	3
Test Strength	88%
Tests Executed	522 (1.22 tests per mutation)

Table 1: Mutation Testing Coverage and Statistics

This approach demonstrates how focused refinements in test case design can significantly enhance mutation testing results, providing comprehensive coverage and targeting critical code paths to improve test quality and robustness. For detailed description regarding the effect of each mutator, refer to the Output.txt file.

The codebase can be found GitHub.

10 Conclusion

Mutation testing is a powerful technique for assessing and improving the quality of test suites. By introducing artificial faults, it uncovers weaknesses in the test cases and enhances their ability to detect real-world bugs. While challenges like computational cost and equivalent mutants exist, advancements in automation and optimization techniques are making mutation testing increasingly practical and effective for modern software systems.

References

1. Jia, Y., & Harman, M. (2011). An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, 37(5), 649–678.

2. Ammann, P., & Offutt, J. (2016). *Introduction to Software Testing*. Cambridge University Press.

3. PITest Documentation.

4. LeetCode