CS 731 - Software Testing

Submitted by:

Rishabh Rai (MT2023088)

Sreeganesh T S (MT2023162)

(Project Link)

Aim

The goal of this project is to apply Mutation Testing on a set of algorithms implemented in Java. The primary focus of the project is to assess the quality and correctness of algorithms by introducing small mutations in their code and testing if these changes are detected through unit tests.

Code Tested

In this project, we focused on testing algorithms implemented in Java.
The list of algorithms tested includes:
- Graph Algorithms
- Dynamic Programming Algorithms
- Basic Data structures
- Divide and Conquer Algorithms
- String Algorithms
- Tree Algorithms

Testing Strategy and Tools Used

Mutation Testing involves making small, controlled changes (mutants) to the source code and checking if the unit tests can detect these changes. The objective is to assess the test suite's ability to identify faults introduced by the mutants. Mutation testing evaluates both the correctness of the tests and the robustness of the system. The tests are said to 'kill' a mutant when they detect a difference in the output caused by the mutation.

There are two types of mutant killing strategies:
1. Weakly Kill a Mutant: The memory state of the program after mutation differs from the original state, but the program output might remain the same.
2. Strongly Kill a Mutant: The program's output differs when the mutant is present compared to when it is not, thereby confirming the detection of the fault.

We aimed to 'strongly kill' the mutants in this project, ensuring that the mutations would lead to observable changes in the program’s output.

Tools Used:
- Java: The programming language used to implement and test the algorithms.
- PIT Mutation Testing Tool: PIT is a mutation testing tool for Java. It automatically introduces mutations in the source code and allows us to evaluate whether the unit tests can detect those mutations.
- VSCode: We used VSCode as our Integrated Development Environment (IDE) for editing and running the Java code, along with the PIT mutation testing tool integrated via Maven.

Mutations Used

The PIT mutation testing tool comes with a set of mutation operators that apply different kinds of changes to the source code. These mutation operators are used to create mutants by altering the program's statements or logic. Below are the mutation operators used in this project:

- BOOLEAN_FALSE_RETURN: Mutates boolean expressions by returning `false` instead of the original expression.
- BOOLEAN_TRUE_RETURN: Mutates boolean expressions by returning `true` instead of the original expression.
- INCREMENTS_MUTATOR: Modifies increment operations (e.g., `i++` to `i--`).
- INVERT_NEGS_MUTATOR: Inverts negations in the code.
- CONDITIONALS_BOUNDARY_MUTATOR: Alters boundary conditions in conditionals (e.g., changing `<` to `<=`).
- MATH_MUTATOR: Introduces mathematical errors by modifying arithmetic operators.
- EMPTY_RETURN_VALUES: Changes the return value of methods to `null` or other empty values.
- NEGATE_CONDITIONALS_MUTATOR: Negates the conditions in the code (e.g., changes `if` conditions to `else`).
- PRIMITIVE_RETURN_VALS_MUTATOR: Modifies primitive return values (e.g., changing return values of integer functions).
- VOID_METHOD_CALL_MUTATOR: Changes method calls that return `void` to return a value or vice versa.
- NULL_RETURN_VALUES: Mutates return values to `null` or empty values.

The mutations introduced by PIT were designed to test whether the existing unit tests could detect these changes in the algorithm implementations.

Results

For each algorithm tested, we applied mutation testing using the PIT tool. The results of the mutation testing, including the effectiveness of the tests and the number of mutants 'killed,'

are presented below.

Pit Test Coverage Report

Project Summary

Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
28	93%	891/960	82%	743/907	87%	743/858

Breakdown by Package

Name	Number of Classes	Line Coverage	Mutation Coverage	Test Strength
com.example.BasicDS	12	91% 420/463	78% 277/356	84% 277/329
com.example.DP	6	98% 122/125	95% 121/127	95% 121/127
com.example.DivideAndCon	2	98% 49/50	81% 73/90	83% 73/88
com.example.Graph	1	95% 18/19	94% 15/16	94% 15/16
com.example.NumberTheory	1	90% 125/139	73% 124/170	83% 124/150
com.example.String	4	97% 141/146	89% 112/126	89% 112/126
com.example.Tree	2	89% 16/18	95% 21/22	95% 21/22

Report generated by PIT 1.9.0

Mathematical Algorithms

Pit Test Coverage Report

Package Summary

com.example.NumberTheory

Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
1	90%	125/139	73%	124/170	83%	124/150

Breakdown by Class

Name	Line Coverage		Mutation Coverage		Test Strength	
NumberTheory.java	90%	125/139	73%	124/170	83%	124/150

Report generated by PIT 1.9.0

String Algorithms

Pit Test Coverage Report

Package Summary

com.example.String

Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
4	97%	141/146	89%	112/126	89%	112/126

Breakdown by Class

Name	Line Coverage		Mutation Coverage		Test Strength	
Anagram.java	100%	42/42	100%	37/37	100%	37/37
KMPSearch.java	100%	33/33	75%	21/28	75%	21/28
MyAtoi.java	93%	52/56	88%	30/34	88%	30/34
Palindrome.java	93%	14/15	89%	24/27	89%	24/27

Report generated by PIT 1.9.0

Basic Data Structure

Pit Test Coverage Report

Package Summary

com.example.BasicDS

Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
12	91%	420/463	78%	277/356	84%	277/329

Breakdown by Class

Name	Line Coverage		Mutation Coverage		Test Strength	
AVLTree.java	81%	57/70	51%	36/71	62%	36/58
BIT.java	100%	14/14	100%	17/17	100%	17/17
DSU.java	95%	20/21	64%	9/14	64%	9/14
LL.java	98%	49/50	95%	20/21	95%	20/21
LRUCache.java	100%	13/13	86%	6/7	86%	6/7
MRUCache.java	92%	12/13	71%	5/7	100%	5/5
MinHeap.java	98%	49/50	79%	34/43	81%	34/42
Queue.java	97%	30/31	96%	22/23	100%	22/22
RedBlackTree.java	79%	91/115	65%	37/57	74%	37/50
SegmentTree.java	100%	31/31	98%	58/59	98%	58/59
Stack.java	96%	22/23	94%	16/17	100%	16/16
Trie.java	100%	32/32	85%	17/20	94%	17/18

Report generated by PIT 1.9.0

Dynamic Programming Algorithms

Pit Test Coverage Report

Package Summary

com.example.DP

Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
6	98%	122/125	95%	121/127	95%	121/127

Breakdown by Class

Name	Line Coverage		Mutation Coverage		Test Strength	
CoinChangeSource.java	100%	18/18	90%	19/21	90%	19/21
EditDistanceSource.java	100%	21/21	88%	22/25	88%	22/25
FibonacciSource.java	96%	27/28	100%	28/28	100%	28/28
KnapsackSource.java	92%	12/13	100%	21/21	100%	21/21
LongestPalindromicSubsequenceSource.java	100%	24/24	100%	14/14	100%	14/14
LongestPalindromicSubstringSource.java	95%	20/21	94%	17/18	94%	17/18

Report generated by PIT 1.9.0

Tree Algorithms

Pit Test Coverage Report

Package Summary

com.example.Tree

Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
2	89%	16/18	95%	21/22	95%	21/22

Breakdown by Class

Name	Line Coverage		Mutation Coverage		Test Strength	
CeilInBinarySearchTree.java	89%	8/9	88%	7/8	88%	7/8
CheckTreeIsSymmetric.java	89%	8/9	100%	14/14	100%	14/14

Report generated by PIT 1.9.0

Divide and Conquer Algorithms

Pit Test Coverage Report

Package Summary

com.example.DivideAndCon

Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
2	98%	49/50	81%	73/90	83%	73/88

Breakdown by Class

Name	Line Coverage		Mutation Coverage		Test Strength	
BinarySearch2dArray.java	96%	26/27	73%	38/52	76%	38/50
RabinKarp.java	100%	23/23	92%	35/38	92%	35/38

Report generated by PIT 1.9.0

Graph Algorithms

Pit Test Coverage Report

Package Summary

com.example.Graph

Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
1	95%	18/19	94%	15/16	94%	15/16

Breakdown by Class

Name	Line Coverage		Mutation Coverage		Test Strength	
Bipartite.java	95%	18/19	94%	15/16	94%	15/16

Conclusion

The project demonstrated the power of mutation testing in identifying weaknesses in the test suite of various algorithms. By introducing small mutations and checking the response of the unit tests, we were able to evaluate the effectiveness of the tests and assess the robustness of the algorithm implementations.