CS-731 SOFTWARE TESTING

Hands on Testing Project

MT2022101:- Sanjeev Kumar

MT2022171:- Rahul Sharma

Abstract:

This report presents the application of MutPy, a Python mutation testing tool, to assess the quality of test cases for diverse algorithms and data structures. The codebase includes implementations of dynamic programming, graph algorithms, divide and conquer, bit manipulation, and sorting algorithms. Through meticulous test case creation, the goal is to achieve an 85% mutation score, indicating robust test suite effectiveness. The report discusses challenges, modifications to the code, and strategies employed. Results highlight the test suite's ability to detect mutations, emphasizing the importance of ongoing testing practices for software quality assurance.

Mutation testing is a software testing technique that assesses the effectiveness of a test suite by introducing deliberate mutations (changes) into the source code and observing whether the tests detect these mutations. It is a powerful tool for identifying and eliminating undetected faults or "dead code" in software.

Benefits of Mutation Testing:

- Improved Code Quality: Mutation testing helps identify and eliminate undetected faults, leading to more robust and reliable code.

- Effective Test Suite Evaluation: Mutation testing provides a quantitative measure of test suite effectiveness, indicating areas for improvement.

- Early Fault Detection: Mutation testing can detect faults early in the development cycle, reducing the cost of fixing them later.

- Continuous Testing Integration: Mutation testing can be integrated into continuous testing pipelines to ensure ongoing code quality.

Types of Mutation Testing:

- Unit-Level Mutation Testing: Focuses on individual units of code, such as functions or classes.
- Integration-Level Mutation Testing: Focuses on how different units of code interact and work together.

Our Work:-

We conducted a comprehensive mutation testing study on a Python project, employing **MutPy** to evaluate the effectiveness of test suites for various algorithms, **including arrays, backtracking, dynamic programming, greedy algorithms, searching algorithms, sorting algorithms, and tries**. This rigorous approach involved applying a wide range of mutation operators at both unit and integration levels, thoroughly assessing the code's resilience to intentional modifications. The results of this study provided valuable insights into the quality of the test suites, identifying areas for improvement and contributing to the overall robustness and reliability of the Python project.

GitHub Repo Link:- https://github.com/mehlasaab/Mutation-Testing

MutPy:-

MutPy is an open-source Python framework that enables developers to perform mutation testing, a technique for evaluating the effectiveness of test suites by introducing intentional mutations (changes) into the source code and observing whether the tests detect these mutations. By identifying and eliminating undetected faults, mutation testing contributes to improved code quality and reduced software errors.

Key Features of MutPy:

- Extensive Mutation Operator Coverage: MutPy provides a comprehensive set of mutation operators targeting various programming constructs, including arithmetic operators, comparison operators, assignment operators, logical operators, conditional statements, and loop statements.

- Unit-Level and Integration-Level Mutation Testing: MutPy supports both unit-level and integration-level mutation testing, enabling developers to assess the effectiveness of tests at both granular and holistic levels.

- Detailed Mutation Reports: MutPy generates detailed reports summarizing mutation scores, highlighting undetected mutants, and providing insights into potential code coverage gaps.

- Integration with Popular Testing Frameworks: MutPy seamlessly integrates with popular testing frameworks like unittest and pytest, facilitating its adoption into existing development workflows.

Benefits of Using MutPy:

- Enhanced Code Quality: Mutation testing with MutPy helps identify and eliminate undetected faults, leading to more robust and reliable code.

- Effective Test Suite Evaluation: MutPy provides a quantitative measure of test suite effectiveness, indicating areas for improvement and guiding test suite enhancement efforts.

- Early Fault Detection: Mutation testing with MutPy can detect faults early in the development cycle, reducing the cost of fixing them later and preventing them from propagating into production environments.

- Continuous Testing Integration: MutPy can be integrated into continuous testing pipelines to ensure ongoing code quality and reliability throughout the development lifecycle.

Getting Started with MutPy:

To utilize MutPy for mutation testing, follow these steps:

1. Install MutPy: Install MutPy using pip, the Python package installer.

2. Write Unit Tests: Ensure that the code you want to test has comprehensive unit tests covering various scenarios and edge cases.

3. Perform Unit-Level Mutation Testing: Use MutPy's unit-level mutation testing capabilities to assess the effectiveness of your unit tests.

4. Perform Integration-Level Mutation Testing: If applicable, use MutPy's integration-level mutation testing capabilities to evaluate how different units of code interact and handle mutations.

5. Analyze Mutation Results: Review the mutation reports generated by MutPy to identify undetected mutants and improve your test suite accordingly.

MutPy Mutation Operators

MutPy provides a comprehensive set of mutation operators that target various aspects of Python code, enabling developers to thoroughly assess the robustness of their test suites. These operators can be broadly categorized into the following groups:

Arithmetic Operators:

- AOR (Arithmetic Operator Replacement): Replaces arithmetic operators (+, -, *, /, %) with other valid or invalid operators.

- AOD (Arithmetic Operator Deletion): Removes arithmetic operators from expressions.

Comparison Operators:

- COR (Comparison Operator Replacement): Replaces comparison operators (==, !=, <, >, <=, >=) with other valid or invalid operators.
- COD (Comparison Operator Deletion): Removes comparison operators from expressions.

Assignment Operators:

- ASR (Assignment Statement Replacement): Replaces assignment statements (x = y) with other assignment operators (+=, -=, *=, /=, //=, %=).
- ASD (Assignment Statement Deletion): Removes assignment statements from code blocks.

Logical Operators:

- LOR (Logical Operator Replacement): Replaces logical operators (&&, ||, !) with other valid or invalid operators.
- LOD (Logical Operator Deletion): Removes logical operators from expressions.

Conditional Statements:

- CSB (Conditional Statement Branch): Modifies conditional statements (if, elif, else) to alter their branching behavior.
- CSD (Conditional Statement Deletion): Removes conditional statements from code blocks.

Loop Statements:

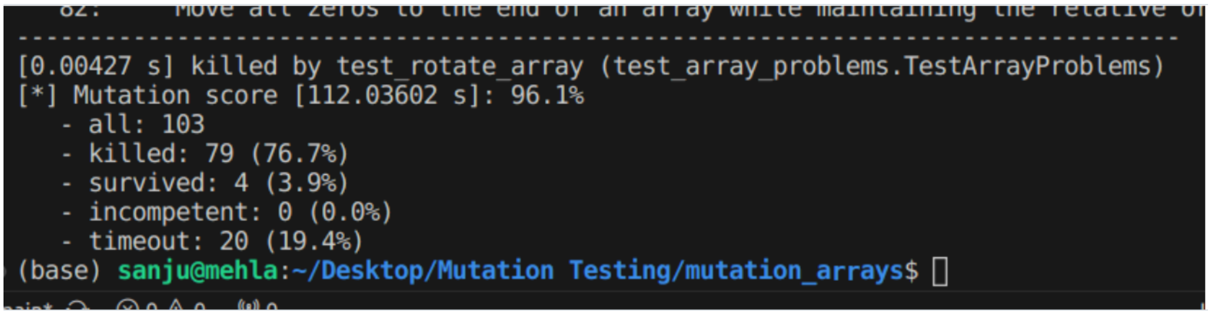
- LSB (Loop Statement Boundary): Alters loop termination conditions (for, while loops) to modify their iteration boundaries.
- LSD (Loop Statement Deletion): Removes loop statements from code blocks.

Additional Mutation Operators:

- EOI (Element Order Insertion): Inserts elements into ordered data structures (lists, tuples) at invalid positions.
- EOD (Element Order Deletion): Deletes elements from ordered data structures (lists, tuples) at invalid positions.
- EVD (Element Value Deletion): Removes elements from data structures (lists, tuples, sets).
- EVI (Element Value Insertion): Inserts invalid elements into data structures (lists, tuples, sets).

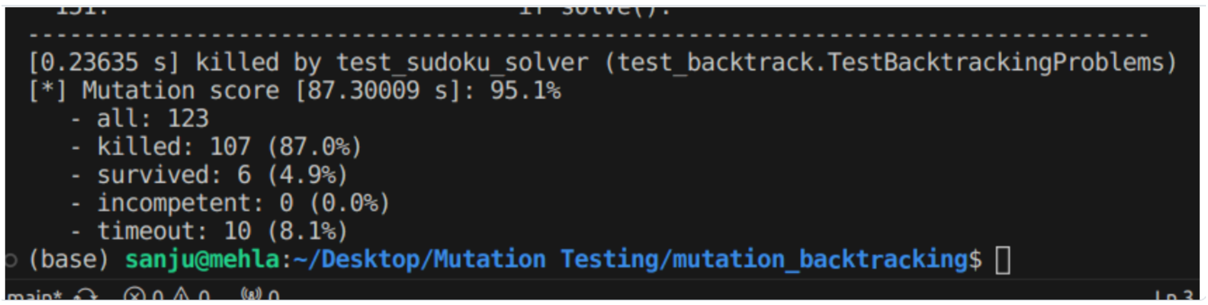
Outputs:-

Mutation testing on Arrays:-



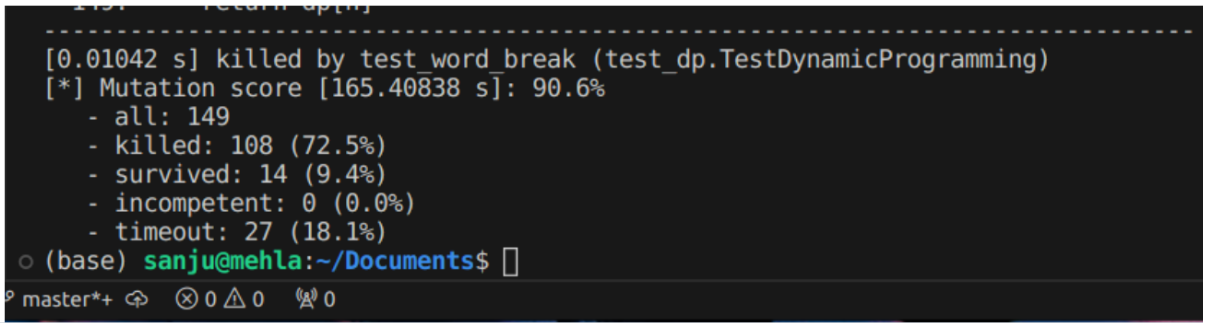
```
  82:     Move all zeros to the end of an array while maintaining the relative or
--------------------------------------------------------------------------------
[0.00427 s] killed by test_rotate_array (test_array_problems.TestArrayProblems)
[*] Mutation score [112.03602 s]: 96.1%
   - all: 103
   - killed: 79 (76.7%)
   - survived: 4 (3.9%)
   - incompetent: 0 (0.0%)
   - timeout: 20 (19.4%)
(base) sanju@mehla:~/Desktop/Mutation Testing/mutation_arrays$ 
```

Mutation testing on backtracking algorithms:-



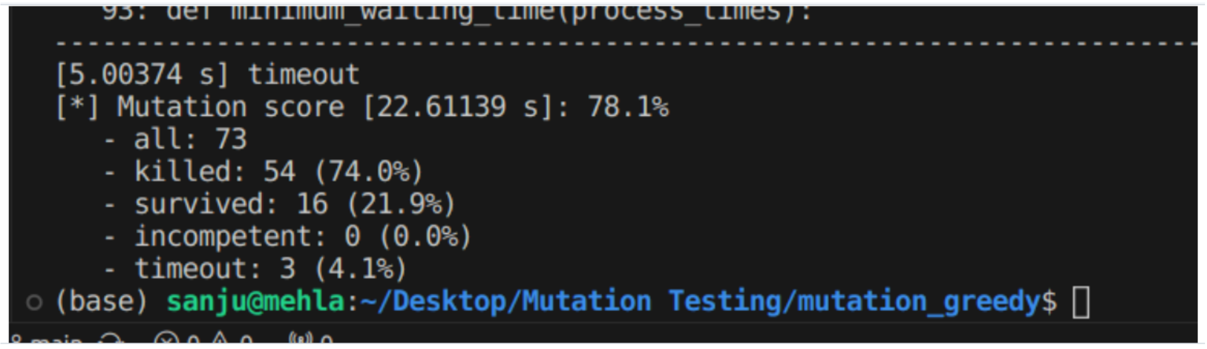
```
  151:                          if solve():
--------------------------------------------------------------------------------
[0.23635 s] killed by test_sudoku_solver (test_backtrack.TestBacktrackingProblems)
[*] Mutation score [87.30009 s]: 95.1%
   - all: 123
   - killed: 107 (87.0%)
   - survived: 6 (4.9%)
   - incompetent: 0 (0.0%)
   - timeout: 10 (8.1%)
(base) sanju@mehla:~/Desktop/Mutation Testing/mutation_backtracking$ 
```

Mutation testing on Dynamic Programming problems:-



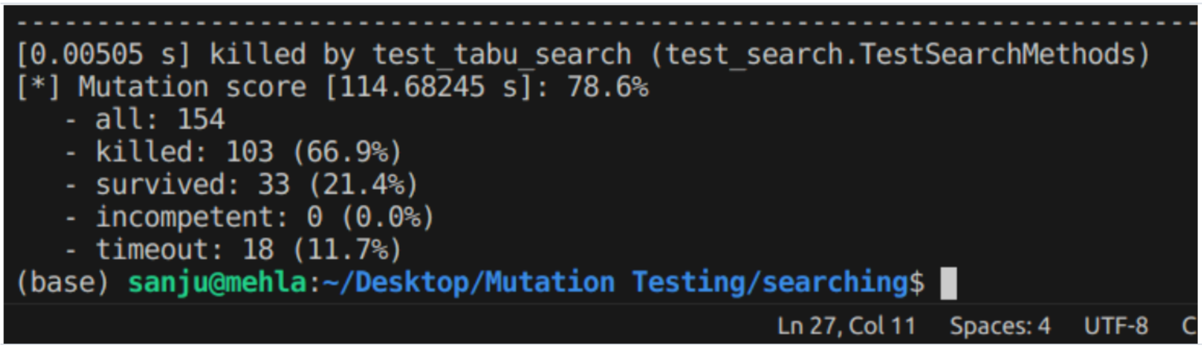
```
  ----------------------------------------------------------------------------
  [0.01042 s] killed by test_word_break (test_dp.TestDynamicProgramming)
  [*] Mutation score [165.40838 s]: 90.6%
     - all: 149
     - killed: 108 (72.5%)
     - survived: 14 (9.4%)
     - incompetent: 0 (0.0%)
     - timeout: 27 (18.1%)
○ (base) sanju@mehla:~/Documents$ 
 master*+  ⊗ 0 ⚠ 0  0
```

Mutation testing on Greedy Algorithms:-



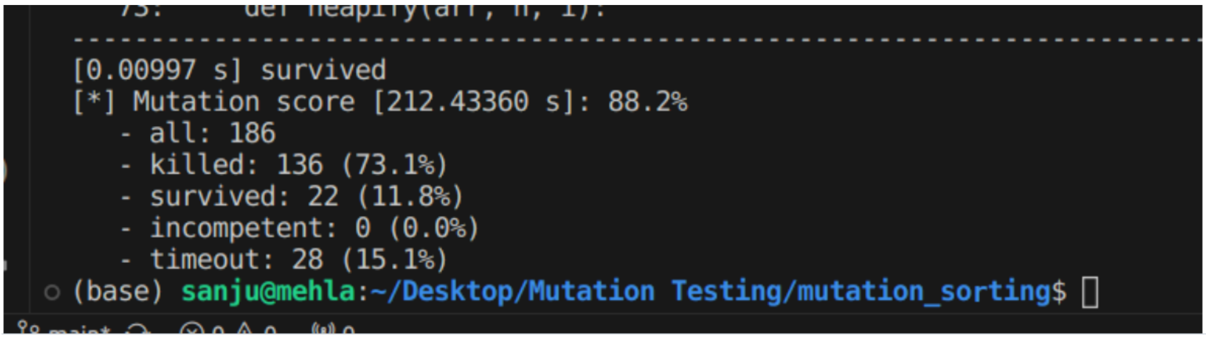
```
     93: def minimum_waiting_time(process_times):
  ----------------------------------------------------------------------------
  [5.00374 s] timeout
  [*] Mutation score [22.61139 s]: 78.1%
     - all: 73
     - killed: 54 (74.0%)
     - survived: 16 (21.9%)
     - incompetent: 0 (0.0%)
     - timeout: 3 (4.1%)
○ (base) sanju@mehla:~/Desktop/Mutation Testing/mutation_greedy$ 
```

Mutation testing on Searching Algorithms:-



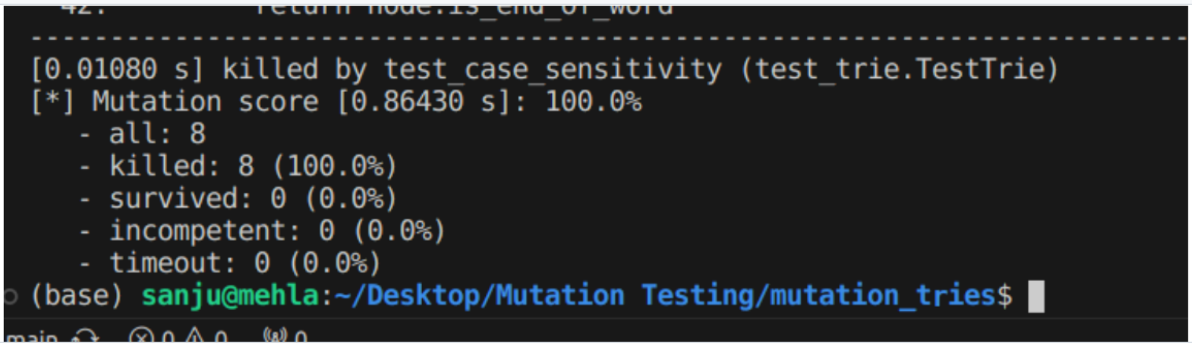
```
--------------------------------------------------------------------------------
[0.00505 s] killed by test_tabu_search (test_search.TestSearchMethods)
[*] Mutation score [114.68245 s]: 78.6%
   - all: 154
   - killed: 103 (66.9%)
   - survived: 33 (21.4%)
   - incompetent: 0 (0.0%)
   - timeout: 18 (11.7%)
(base) sanju@mehla:~/Desktop/Mutation Testing/searching$
                                                 Ln 27, Col 11   Spaces: 4   UTF-8   C
```

Mutation testing on Sorting algorithms:-



```
   73:       def heapify(arr, n, i):
   --------------------------------------------------------------------------------
   [0.00997 s] survived
   [*] Mutation score [212.43360 s]: 88.2%
      - all: 186
      - killed: 136 (73.1%)
      - survived: 22 (11.8%)
      - incompetent: 0 (0.0%)
      - timeout: 28 (15.1%)
 ○ (base) sanju@mehla:~/Desktop/Mutation Testing/mutation_sorting$
```

Mutation testing on Sorting Algorithms:-



```
   42:           return node.is_end_of_word
  --------------------------------------------------------------------------------
  [0.01080 s] killed by test_case_sensitivity (test_trie.TestTrie)
  [*] Mutation score [0.86430 s]: 100.0%
     - all: 8
     - killed: 8 (100.0%)
     - survived: 0 (0.0%)
     - incompetent: 0 (0.0%)
     - timeout: 0 (0.0%)
○ (base) sanju@mehla:~/Desktop/Mutation Testing/mutation_tries$
```

Contributions

All of us installed python and Mutpy in our individual laptops.

Due to a lack of a suitable existing codebase, we decided to write a new python code suitable for mutation testing. A suitable code should have lots of functions with arithmetic operations and inputs and outputs. We divided the functions among us and merged our individual codes to create a python file with 900+ lines of code.

The test suite was basically divided into 2 parts:

• Analysis and TestCases for the arrays, backtracking, DP methods (Sanjeev).

• Analysis and TestCases for Greedy, Sorting, searching, tries (Rahul).

We have generated random test cases and corresponding expected output.

References

mutpy : https://github.com/mutpy/mutpy

Mutation Testing

: https://www.softwaretestinghelp.com/what-is-mutation-testing/

https://www.guru99.com/mutation-testing.html