Mutation Testing Analysis

Overview

Mutation testing is a software testing method that introduces small changes, known as mutations, to the source code to assess the quality of the test suite. The purpose of this process is to evaluate whether the current tests can detect these modifications, highlighting any gaps or weaknesses in the test coverage.

Mutation Operators Used

The following mutation operators were applied during the testing process:

1. **Coefficient Modification**: Adjusted the coefficients of the polynomial.
2. **Arithmetic Operation Replacement**: Replaced one arithmetic operator with another (e.g., + to -).
3. **Comparison Operator Adjustment**: Modified comparison logic (e.g., < to <=).
4. **Zero Coefficient Addition**: Added a zero coefficient to the polynomial representation.
5. **Method Elimination**: Removed specific methods from the class.

Details of Mutations and Their Effects

1. **Coefficient Modification**
    - Example: self.coefficients = coefficients changed to self.coefficients = [c + 1 for c in coefficients].
    - Effect: Altered the polynomial's internal representation, impacting all related computations.
2. **Arithmetic Operation Replacement**
    - Example: Replaced a + b with a - b in the add method.
    - Effect: Reversed addition behavior, disrupting polynomial arithmetic functionality.
3. **Comparison Operator Adjustment**
    - Example: Changed abs(self.evaluate(c)) < epsilon to abs(self.evaluate(c)) <= epsilon in find_root_bisection.
    - Effect: Risked premature termination of the root-finding algorithm due to altered stopping conditions.
4. **Zero Coefficient Addition**
    - Example: Modified self.coefficients = coefficients to self.coefficients = [0] + coefficients.

    - Effect: Increased the polynomial's degree by one, affecting arithmetic operations and its string representation.
5. **Method Elimination**
    - Example: Removed the evaluate method entirely.
    - Effect: Disabled functionality relying on polynomial evaluation, such as root finding.

Mutation Outcomes
All introduced mutations were detected by the test suite, demonstrating its effectiveness. Below is a summary of test results:

- **Tests Passed**: test_init, test_add.
- **Tests Failed**: test_str, test_sub, test_mul, test_first_degree_polynomial, test_second_degree_polynomial, test_third_degree_polynomial.

Failures in test_str suggest issues with the string representation, while failures in test_sub and test_mul highlight problems with subtraction and multiplication operations. The absence of find_root_bisection or the evaluate method caused failures in degree-specific tests. These results indicate that the test suite adequately identifies significant changes to the class's core functionalities.

Assessment of Test Suite Strength
The mutation testing results show that the test suite is generally robust, successfully identifying all code alterations. However, the failures suggest areas where the tests could be enhanced:

- The consistent failure of test_str indicates a strong reliance on this method's correctness.
- Failures in test_sub and test_mul point to opportunities to improve test coverage for these operations.
- Passing results for test_init and test_add suggest these methods are less vulnerable to the introduced changes, but additional edge case testing could further verify their stability.

Suggestions for Improvement

1. **Expand Coefficient Tests**
    - Add more cases to validate polynomial evaluations with slightly altered coefficients.
2. **Strengthen Arithmetic Tests**

    - Test a broader range of scenarios for addition, subtraction, and multiplication, including edge cases (e.g., adding a polynomial to itself).
3. **Cover Edge Cases**
    - Incorporate tests for zero polynomials, single-term polynomials, and cases with leading zeros in the coefficients.

Conclusion

The mutation testing process indicates that the current test suite for the Polynomial class is effective in identifying code modifications. However, incorporating additional edge cases and enhancing the coverage of arithmetic operations and coefficients can further improve its reliability. This iterative approach to mutation testing and test suite refinement is critical for maintaining high code quality and ensuring the class functions as intended through future changes.