CS 520 Homework 2 - Aryan Tipnis

Automated Testing and Coverage

Github Link: https://github.com/aryantipnis/CS520_HW2

Part 1 - Baseline Coverage:

For this part, we analyze 10 problems and their corresponding solutions from Assignment 1. Using the benchmark test suites provided in the datasets, we run automated tests with **pytest** and measure both line and branch coverage to establish a baseline. The results include the number of tests passed, line and branch coverage for each problem.

Problem	Test cases passed	Line Coverage	Branch Coverage	Summary
Problem 1	100%	100%	100%	Fully covered
Problem 2	100%	59%	83%	Low coverage; lines 22-26 untested
Problem 3	100%	71%	62%	Some branches not tested: lines 19, 21, 26
Problem 4	100%	100%	100%	Fully covered
Problem 5	100%	82%	75%	Line 17 untested reduces coverage
Problem 6	100%	67%	62%	Lines 10, 14, 18 not executed
Problem 7	100%	100%	100%	Fully covered
Problem 8	100%	86%	83%	Lines 10, 14 untested
Problem 9	100%	64%	50%	Lines 11, 13 untested, low branch coverage
Problem 10	100%	90%	83%	Line 82 missed in tests

For the following parts, we will work with Problem 3 and Problem 9 because they have the largest gap between tests passed and branch coverage, indicating room for coverage improvement with LLM-generated tests.

Part 2 - LLM Assisted Test Generation and Coverage Improvement:

Problem 3

Iteration 1:

Prompt: *"Generate unit tests for this function that cover all branches, including edge cases [Paste code here]"*
Summary:

Problem	Test cases passed	Line Coverage	Branch Coverage	What changed?
Before				
Problem 3	100%	71%	62%	Some branches not tested: lines 19, 21, 26
After				
Problem 3	100%	100%	100%	The generated test cases covered all possible range of inputs like input not an int, input = 0, input < 0, input > 0, single digit and double digit

How did I handle duplicates?

The LLM did produce some tests that were similar to our baseline. To avoid redundant coverage, we kept only one representative input from each category or range. For example, one test case each for invalid input types when input is string, float, etc. and one test case each for input = 0, input < 0, input is single digit, input is multiple digits, etc. This ensured that coverage improvement was meaningful and not inflated by duplicate cases, while maintaining a diverse set of inputs that target all important branches.

Iteration 2:

Prompt: "Now that all main branches are covered, suggest stress tests or corner cases that might reveal hidden issues."

[Results can be seen in final summary table below]

Iteration 3:

Prompt: *"*Generate additional unit tests that could potentially reveal hidden edge cases or corner conditions not yet covered by the existing test suite. Avoid repeating inputs similar to previous tests.*"*

[Results can be seen in final summary table below]

Final Summary:

Iteration	Test cases passed	Line Coverage	Branch Coverage	Notes
Baseline	100%	64%	50%	Lines 11, 13 untested, low branch coverage
Iteration 1	100%	100%	100%	All branches covered
Iteration 2	100%	100%	100%	No increase convergence <= 3%
Iteration 3	100%	100%	100%	No increase convergence <= 3%

Problem 9

Iteration 1:
Prompt: *"Generate unit tests for this function that cover all branches, including edge cases [Paste code here]"*

Summary:

Problem	Test cases passed	Line Coverage	Branch Coverage	What changed?
Before				
Problem 9	100%	64%	50%	Lines 11, 13 untested, low branch coverage
After				
Problem 9	100%	100%	100%	The generated test cases covered all possible range of inputs including valid and invalid inputs rather than random test cases that provided low coverage previously.

How did I handle duplicates?

The LLM did produce some tests that were similar to our baseline. To avoid redundant coverage, we replace the previous tests with new tests that keep only one representative input from each category or range. For example, few inputs where t is not an int, when some variables are equal to 0, less than 0, large numbers, etc. This ensured that coverage improvement was meaningful and not inflated by duplicate cases, while maintaining a diverse set of inputs that target all important branches.

Iteration 2:

Prompt: *"*Now that all main branches are covered, suggest stress tests or corner cases that might reveal hidden issues.*"*

[Results can be seen in final summary table below]

Iteration 3:

Prompt: *"*Generate additional unit tests that could potentially reveal hidden edge cases or corner conditions not yet covered by the existing test suite. Avoid repeating inputs similar to previous tests.*"*

[Results can be seen in final summary table below]

Final Summary:

Iteration	Test cases passed	Line Coverage	Branch Coverage	Notes
Baseline	100%	64%	50%	Lines 11, 13 untested, low branch coverage
Iteration 1	100%	100%	100%	All branches covered
Iteration 2	100%	100%	100%	Lines 11, 13 untested, low branch coverage
Iteration 3	100%	100%	100%	Lines 11, 13 untested, low branch coverage

Why is branch coverage a better metric than line coverage?
Branch coverage is a better metric because it looks at whether every possible path in your code has been tested and not just whether each line ran once. Line coverage might show 100% even if some conditions like the "else" part of an "if " statement were never actually tested. But branch coverage checks all decision outcomes, making it more effective at revealing untested logic or hidden bugs.

Part 3 - Fault Detection:

Problem 3

I performed a seeded bug check by introducing a small, realistic bug into the A1 solution and re-ran the tests. The tests failed, meaning they catched the bug.

1. **What bug you injected/compared against and why it's realistic?**
   The problem asks to return the total sum of its digits in binary as a string, given a positive integer N. I intentionally introduced a small, realistic bug by modifying the return statement in the "if" block in the function to return 0 instead of the correct return "0". The special case N=0 breaks since we add a bug where the function incorrectly returns an integer 0 instead of a string.

   This is a **type-related bug**, commonly seen when developers mix up return types. For example, returning a numeric zero instead of its string form. It's realistic because it doesn't affect most numeric computations, but breaks interface consistency and can cause subtle downstream errors.

2. **Whether your tests failed as expected and which tests caught it?**
   After running all test cases, **2 out of 24 tests failed**.

   Yes, the tests failed as expected. Example of failed Tests:
   - test_zero_input → Expected "0", got 0.
   - test_input_as_boolean_true_false → For input False, expected "0", got 0.

3. **A short conclusion linking coverage - fault detection?**
   This test failure demonstrates that branch coverage and type-aware assertions are both crucial for effective fault detection because the tests exercised the if n == 0 path, the bug in that branch was immediately exposed. Even though this was a small, single branch issue, the test suite's wide range ensured the bug did not go unnoticed.

   This concludes that high branch and input coverage improved fault detection by ensuring that even small, realistic errors caused multiple test failures across different scenarios.

Problem 9

I performed a seeded bug check by introducing a small, realistic bug into the A1 solution and re-ran the tests. The tests failed, meaning they catched the bug.

1. **What bug you injected/compared against and why it's realistic?**
   A realistic bug was introduced in the formula for computing the *t-th term* of a geometric series. The correct formula is **term=a×r(t−1)** we changed it to **term=a×r(t+1)**

   This causes an exponent error. It's realistic because such errors often occur when adjusting between zero-based and one-based indexing in formulas.

2. **Whether your tests failed as expected and which tests caught it?**
   After running all test cases, **9 out of 29 tests failed**.

   Yes, the tests failed as expected. Example of failed Tests:

   - test_first_term (expected 5, got 45.0)
   - test_typical_geometric_term, test_fractional_ratio, and test_negative_ratio : all failed due to incorrect exponentiation.
   - Test_t_large_near_overflow_boundary : triggered an OverflowError, revealing the compounding effect of the wrong exponent.

   This shows that the test suite thoroughly checks different magnitudes, ratios, and edge cases.

3. **A short conclusion linking coverage - fault detection?**

   The results show that a wide input coverage including positive, fractional, negative, and large ratio values, allowed the test suite to immediately detect the off-by-one exponent bug. Since high branch coverage included validation checks and all numerical paths, the functional logic error was exposed across multiple tests.

   This again concludes that high branch and input coverage improved fault detection by ensuring that even small, realistic errors were caught.