Analysis of PathCrawler Output for Function `Tritype`

PathCrawler successfully instrumented and analyzed the `Tritype` function using the all-path coverage criterion. While 100% branch coverage was achieved, none of the 11 generated test cases were categorized as successful or failed, leaving all as "unknown." This implies that the tool executed all branches in the code, yet could not conclusively determine the verdict for any specific path, possibly due to incomplete post-condition specifications or ambiguity in return value interpretation. The test harness was properly compiled, and the environment was initialized and cleaned up, suggesting that the tool functioned without interruptions. However, the presence of infeasible paths (8 of 19) emphasizes the complexity of the control flow. The inability to evaluate paths conclusively indicates a need to refine input constraints or further investigate symbolic execution limitations. Ultimately, this dataset offers deep insights into path feasibility without final verdict certainty.

The test generation summary reveals a comprehensive attempt by PathCrawler to traverse 19 total paths within the `Tritype` function. Of these, 11 were covered and 8 marked as infeasible. The success of achieving 100% branch coverage suggests an effective instrumentation and exploration of all conditional branches in the function. However, none of the generated test cases resulted in a definitive "success" or "failure" outcome, indicating that path postconditions or functional assertions may not have been defined or interpreted robustly enough for classification. The discrepancy between path coverage and test verdicts signals potential limitations in either the underlying test oracle or symbolic constraint solving. Nevertheless, all test cases completed without runtime errors or crashes, which strengthens the confidence in static path feasibility. This situation is common in symbolic execution where certain paths are executable in theory but do not yield assertable functional correctness outcomes.

This table demonstrates that subcondition 10b was explored most frequently, present in all test cases. The spread of paths across subconditions 12c through 17 shows an extensive reach through different logical segments of `Tritype`. Notably, the fewer occurrences in subcondition 12c imply stricter path constraints or deeper nesting. These insights help in prioritizing potential revisions or debugging efforts in the function.

Several test cases (such as 3, 4, and 7) represent crucial boundary classifications in triangle identification logic. Test Case 3, returning 0, indicates that the input values do not form a valid triangle—corroborated by the path predicate involving non-equal sides and valid sum conditions. Test Cases 4 to 7 exhibit values returning 1 or 2, which correspond to isosceles and equilateral triangle scenarios. These variations affirm the tool's capability to explore distinct triangle classifications based on the input side lengths. However, all cases have an "unknown" verdict. This suggests that while execution paths were valid, postconditions or specification assertions did not yield sufficient evidence for a conclusive outcome. It is also possible that comparison thresholds or floating-point

equality checks introduced uncertainty. The tool's symbolic reasoning did cover a wide input range, showcasing strength in data diversity.

This selection emphasizes the diversity in input ranges—both positive and negative, and also includes floating-point values. Negative and invalid inputs, such as in Case 1, correctly trace to a return of 3, denoting an error or invalid triangle. Equilateral triangles like in Case 7 return 2, consistent with the expected logic. The presence of precise floating values (e.g., 886.99) highlights the need for robust type handling in symbolic execution and comparison logic. The return values indicate that, functionally, the outputs align with expectations, even if PathCrawler's analysis labels the verdict as unknown.

Examining the path predicates reveals detailed symbolic constraints applied during test generation. For example, Case 3 checks all triangle inequality constraints and non-equality of sides. Case 7's predicate confirms all sides are equal, ideal for verifying the equilateral branch of logic. These conditions provide assurance that symbolic execution adheres to logical expectations derived from triangle properties. However, predicates like those in Cases 1, 10, and 11 reflect invalid or boundary-breaking inputs—some involving negative sides. While such inputs yield predictable output values, their classification remains "unknown" due to a lack of explicit functional assertions. The predicates affirm that PathCrawler applies comprehensive constraint exploration but might benefit from clearer postcondition verification to move cases from "unknown" to "success" or "failure."

The analysis indicates high internal code coverage but limited external verdict classification due to insufficient assertions or postcondition specifications. While symbolic execution succeeded in evaluating feasible and infeasible paths and fully traversing conditional logic, the lack of verdicts suggests an improvement opportunity in integrating output expectations with test oracles. PathCrawler's utility in this context lies more in code path analysis and constraint checking than in final correctness validation. Future work may involve explicitly encoding expected behavior (e.g., via assertion annotations or return value comparisons) to convert "unknown" results into definitive outcomes. Despite these limitations, the tool has proven its capability in surfacing hidden paths and exercising diverse execution flows. Particularly in legacy or critical systems like `Tritype`, such analysis can prevent unanticipated bugs or logic flaws through exhaustive test path enumeration.