lancecho@umich.edu

I decided to focus on a set of fundamental mutation operators to ensure a comprehensive evaluation of the test suite adequacy. I implemented mutation operators such as changing arithmetic operators (+, -, *, /) to their counterparts, negating conditionals (if-else), and replacing literals with different values. These operators aim to simulate potential faults in the codebase, allowing me to assess the effectiveness of the test suite.

To determine which operators to apply, my program randomly selects mutation operators based on predefined probabilities. I assigned probabilities to each operator, considering their impact on the code and the likelihood of introducing faults. By randomly selecting operators, I ensure diversity in the generated mutants, contributing to a robust assessment of the test suite's effectiveness.

Developing the mutation testing tool was more challenging than anticipated, particularly in designing mutation operators that effectively simulate real-world faults. Understanding the intricacies of Python's abstract syntax tree (AST) and implementing mutations without breaking the code's syntactic integrity required careful consideration and testing. Additionally, ensuring the determinism of the tool was crucial to guarantee reproducibility across executions.

Mutation analysis, unlike statement coverage, provides a more nuanced evaluation of the test suite's quality by assessing its ability to detect subtle faults in the code. While statement coverage measures the proportion of executed code, mutation analysis examines the test suite's capability to detect changes in the program's behavior. By generating mutants and evaluating how many are killed by the test suite, mutation analysis provides insights into the effectiveness of the testing efforts in identifying potential faults, complementing traditional coverage metrics.