Keenan Herbe

Dr. Zhao

Final Project Report

Dec 8, 2025

Evolving Test Cases with Genetic Algorithms for Improved Bug Detection

Github: https://github.com/Kee-nan/GeneticAlgorithmAdvSoft

Introduction:

Software testing has always been a critical and time-consuming component of software development. In crafting tests, traditional manual options struggles to reveal edge cases, leading to insufficient code coverage and undetected bugs that can propagate into production systems as technical debt. This has been an important point throughout the course, and has lead us to study different testing strategies and algorithms in order to help combat these limitations. One of these was simple random testing, which does provide an automated alternative, but randomly generated inputs rarely stress corner cases as well. As a result, both manual and random testing routinely fail to expose deeper logical faults in algorithmic implementations.

To continue searching for better methods, this project aims to address the limitations of manual and random testing by leveraging another strategy we covered in class: Genetic Algorithms. The central idea is that GA-generated inputs should be able to progressively improve their effectiveness over generations, producing test sets that achieve higher code coverage and increased bug-detection capability when compared to purely random test generation.

The tested dataset we will use focuses on a representative subset of algorithmic problems with both correct and buggy implementations. Having both of these implementations allows us to evaluate not only code coverage, but also the ability of the created GA to expose logical inconsistencies and bugs. Therefore, the project explores whether a GA can outperform random testing in exposing faults and improving test quality.

Problem Statement:

The precise problem addressed in this project is how can we automatically generate test inputs that maximize code coverage and reveal logical bugs more efficiently than random test generation?

Many algorithmic problems, like the ones seen on Leetcode, rely on intricate branching logic, boundary checks, arithmetic corner cases, and array manipulation patterns. Manually crafting sufficient tests for these problems is difficult and requires domain knowledge. Random tests, while easy to generate, rarely trigger deep execution paths or rare branching behavior.

Thus, the ultimate goal is to create a test-generation engine that searches efficiently through the input space, discovers diverse execution paths, and identifies inputs where the correct and buggy implementations behave differently.

To solve this problem, I integrate a Genetic Algorithm framework with both code coverage tracking and a bug detection mechanism. The GA evolves inputs by treating each input as a

chromosome, evaluating its fitness based on coverage and oracle disagreement, and applying selection, crossover, and mutation to iteratively improve the population. This was done based on the interest of the algorithm as described in earlier lectures. The hypothesis is that this evolutionary pressure leads the GA to explore deeper, rarer execution behaviors than naive random testing.

Experimental Evaluation

The core of the evaluation compares GA-generated tests to randomly generated tests along two axes: total coverage across all discovered inputs and number of detected logical bugs (oracle mismatches)

Using the experiment.py file in the root, I performed experiments across 6 LeetCode algorithmic problem answers, where each of these files had both a correct and buggy implementation. The experiment file runs both naïve random inputs for a set number of cases, and then calling the genetic algorithm I made, collecting these axes for each trial, and then saving them to a csv. In each trial, the GA was run for a fixed number of generations with population size held constant, and the random generator was allowed an equal number of input evaluations.

```
trial,problem,random_avg_coverage,random_failures,random_n_tests,ga_avg_coverage,ga_failures,ga_n_tests
1,reverse_pairs,45.56153846153843,66,250,51.60000000000004,205,250
2,reverse_pairs,45.60769230769228,90,250,51.83076923076926,187,250
3,reverse_pairs,44.56153846153845,66,250,51.83846153846157,203,250
4,reverse_pairs,45.3307692307692,72,250,51.79230769230774,187,250
5,reverse_pairs,45.569230769230714,66,250,51.48461538461542,204,250
```

Figure 1: Result of 5 trials of reverse_pairs problem

Across all 6 of the problems, we did see anywhere from a 3%-12% increase in code coverage between the random inputs to that of the ga algorithm. To provide one example of an output file, we can look above at *Figure 1. We can* see that, across repeated trials, random testing consistently achieved around 45% coverage on average. This behavior aligns with expectations because the problem's logic is highly branch-dependent, random arrays rarely hit the edge-case structural relationships needed to trigger deeper execution paths. By contrast, the genetic algorithm reliably achieved 51% coverage, demonstrating a stable and repeatable improvement in structural exploration. This suggests that evolutionary search carried out by the built genetic algorithm is in fact able to steer input distributions towards better code coverage.

We can see that for failures we also see a big improvement when moving into the genetic algorithm. Due to the weighted value in fitness for failures and detecting bugs, after the first few generations we are then able to see the vast majority of individuals in the population trigger bugs in the program. In Figure 1, for the reverse pairs proble, we can see that random generation tends to trigger anywhere from 66-90 bugs, while ga failures ranges between 187-205 bugs detected. This confirms that it is indeed possible to have the algorithm learn to suggest inputs that will trigger bugs in a given program.

Future Work

trial	problem	random_avg_coverage	random_failures	random_n_tests	ga_avg_coverage	ga_failures	ga_n_tests
1	median_two_arrays	31.69459962756054	91	250	27.67111111111111	202	250
2	median_two_arrays	30.862818125387964	87	250	24.41777777777778	226	250
3	median_two_arrays	30.6257982120051	96	250	22.595555555555556	232	250
4	median_two_arrays	30.57680631451125	85	250	27.448888888888895	208	250
5	median_two_arrays	29.73099415204681	89	250	28.595555555555553	205	250

Figure 2: The result of 5 trials of median_two_arrays problem

While for the most part we were able to get consistent results and significantly improve both code coverage and bug detection across the board, there were some instances in results that only chose one or the other. For instance, in *Figure 2*, for median_two_arrays, we can see that we are really good at finding inputs that find errors in the GA, but in these trials we actually got a worse coverage value for all the trials. This suggests that our evaluator function is imperfect and may not be the best when universally applied to different kinds of algorithms. Because we have a lot of success in our failures, but none in our coverage, this suggests that maybe we have weighed failure-finding too heavy in the fitness evaluator for this testing problem. Each generation rewards inputs that detect failures far more than it should, and is sacrifiing improving code coverage for reaching greater errors.

To combat this, we would need to continue to tweak and modify values like mutation likelihood and severity, crossover likelihood and severity, and fitness calculation in evaluate_indivudal. These may need to be actively tweaked depending on the kind of problem we are evaluating or to be more universally beneficial across the board.

With these improvements, the genetic algorithm is likely to surpass random testing in both coverage and bug detection more consistently, realizing its potential as a robust adaptive test generation method.