Genetic Testing Software Report

Jordan Shaheen and Gareth Fultz

December 2024

1 Introduction & Problem Description

Software testing is an important aspect of software engineering. However, creating comprehensive test inputs can be a challenging and time-consuming task. Automatic test input generation allows developers to accelerate and improve this process by reducing manual effort and improving testing efficiency. In this project, we addressed the problem of automated test input generation specifically for Python, a programming language lacking existing tools for this purpose. Our tool, PyGET (Python GEnetic Testing), utilizes a genetic algorithm to generate comprehensive test inputs (genetic testing) aimed at maximizing code coverage. This simplifies the creation of strong test suites.

2 Techniques

Genetic algorithms comprise four key components: a fitness function, a selection strategy, a crossover process, and mutation algorithms. We developed specialized and novel approaches for each step, along with enhancements to other aspects of the algorithm. The implementations are detailed below.

2.1 Fitness Function

We have developed a custom fitness function known as shared statement coverage. It is an adaptation of the traditional statement coverage metric where the score for each statement is distributed among all test inputs that execute it. This promotes diversity within the population and prevents it from optimizing solely for the longest path. To visualize this, consider the example code below:

```python
def func(a: int, b: int, c: int):
    if a > b and a > c:
        return 0
    a  = a + b
    c  = c + 1
    if c > b + 2:
        b = b * 2
    return 1
```

A traditional statement coverage fitness function would aim to maximize the number of statements executed by each test input. This would likely lead all test inputs to avoid the '**if a > b and a > c:**' branch, since it only covers a small portion of the whole function func(). With the shared statement coverage fitness function, test inputs are encouraged to explore this branch because the score for executing these statements is higher when fewer candidates have covered them.

2.2 Selection

Our tool has a number of parameters that can be used to configure the selection process. Most notably, we have implemented two selection strategies: roulette selection and tournament selection. Roulette selection randomly chooses test inputs from the population weighted by their fitness score. Tournament selection randomly groups test inputs into "tournament brackets" where the test input with the highest fitness is selected. Additionally, we provide an option for elitism, which ensures that the test inputs with the highest fitness score are preserved. Specifically, it allows the user to specify the number of top candidates that should move on to the next generation.

2.3 Crossover

PyGET uses the uniform crossover technique to combine values from parent test inputs. In this method, each child value is independently chosen from one of the parents with a 50% probability. For example, given Parent 1 with values (a, b, c) and Parent 2 with values (d, e, f), their resulting offspring could be C1 = (a, e, f) and C2 = (d, b, c). This technique allows parents to pass on all of their genes, while also ensuring a balanced mix of traits from both parents.

2.4 Mutation

The mutation step follows the same principles used by most genetic testing algorithms. In our tool, we have defined specific types of mutations can occur on each supported data type. Examples of mutations include adding or removing characters in a string, incrementing an numeric value, and appending or deleting a list element. Additionally, users can configure the mutation rate for our tool.

2.5 Interesting Values

A novel technique implemented in our tool is the use of *interesting values*. Essentially, these are values that are likely to trigger unique behavior. For example, consider the empty list ([]). Many functions have explicit checks to handle empty lists, making this value is more likely to execute untested paths. Our tool leverages two types of interesting values:

- Simple Interesting Values: These values are universally interesting to all functions. The empty list is an example of a simple interesting value because any function that takes in a list will likely benefit from testing the behavior when the list is empty.
- Dynamic Interesting Values: These values change from function to function, and are obtained by analyzing the source code of the function under test. For example, if a function contains a condition that checks whether a numeric input equals a certain value (e.g., if x == 42), 42 would be considered a dynamic interesting value for that function.

PyGET (optionally) utilizes these values by making them more likely to occur in the initial population and in certain mutations, allowing it to achieve higher code coverage with fewer generations.

3 Technical Details

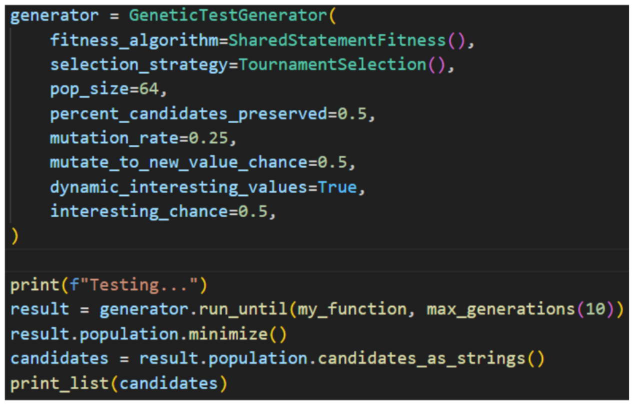
3.1 Interface

Users primarily interact with PyGET through the 'GeneticTestGenerator' class. This class can be configured through its constructor parameters and used by calling the generator's 'run_until(func, end_condition)' method. Figure 1 demonstrates an example of how the tool is used.

3.2 Supported Types

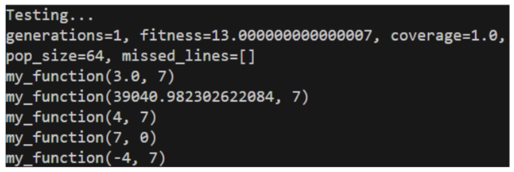
By default, PyGET supports the following types: int, float, str, bool, list, and dict. Lists and dictionaries are special because they are generic types, which means they have type arguments. They are only supported if the inner types are also supported. For example, PyGET can natively handle lists of integers, but not lists of an unsupported class. If necessary, users can add support for their own types by providing a creation function and mutation functions.

PyGET determines the type of function parameters by inspecting their type hints. It then uses that information when generating values for each parameter. This means that PyGET only works with functions that contain type hints.



```
generator = GeneticTestGenerator(
    fitness_algorithm=SharedStatementFitness(),
    selection_strategy=TournamentSelection(),
    pop_size=64,
    percent_candidates_preserved=0.5,
    mutation_rate=0.25,
    mutate_to_new_value_chance=0.5,
    dynamic_interesting_values=True,
    interesting_chance=0.5,
)

print(f"Testing...")
result = generator.run_until(my_function, max_generations(10))
result.population.minimize()
candidates = result.population.candidates_as_strings()
print_list(candidates)
```



```
Testing...
generations=1, fitness=13.000000000000007, coverage=1.0,
pop_size=64, missed_lines=[]
my_function(3.0, 7)
my_function(39040.982302622084, 7)
my_function(4, 7)
my_function(7, 0)
my_function(-4, 7)
```

Figure 1: Code using PyGET to generate test inputs, and its corresponding console output

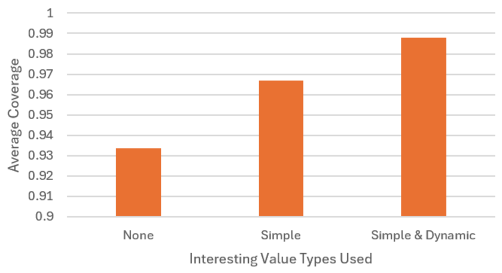


Figure 2: Average code coverage by interesting value usage

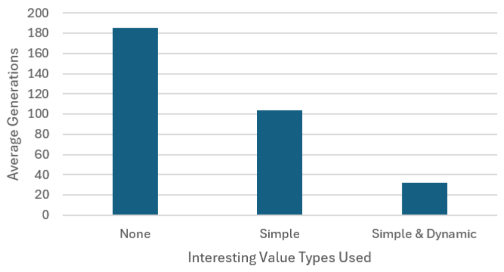


Figure 3: Average generations to reach result by interesting value usage

4 Evaluation & Results

4.1 Dataset

We evaluated PyGET on 54 functions from a Python Instruct dataset[1]. The functions were manually selected and modified to include type hints. Functions without branching logic were excluded, as these functions would have 100% coverage with any input.

4.2 Overall

We performed our evaluation with a population size of 64, a mutation rate of 0.25, simple & dynamic interesting values, and a maximum execution time of 5 minutes per function. Using these settings, PyGET achieved ∼98.8% statement coverage in an average of 32.2 generations. By examining the functions where we did not achieve 100% coverage, we discovered that PyGET struggles when inputs are needed in a very specific format (i.e. a date-time string).

4.3 By Interesting Value Usage

We also measured the impact of interesting values on the performance of our tool. The results from these tests are shown in Figure 2 and Figure 3. We found that including interesting values significantly increased the resulting code coverage (∼93.4% vs ∼98.8%) and decreased the number of average generations necessary (∼32.2 vs ∼185.7).

[1]Found at https://www.kaggle.com/datasets/thedevastator/python-code-instruction-dataset.