PyGET: Genetic Algorithm for Automatic Test Input Generation

By: Jordan Shaheen and Gareth Fultz

Introduction

- Problem
  - Creating comprehensive test inputs is an important part of software engineering that manually takes significant time and effort.
  - We address this with a genetic algorithm for automating test input generation that maximizes code coverage.
- Motivation
  - Automating test input generation accelerates the software engineering process, saving time and effort.
  - This course made our group was interested in genetic algorithms. We wanted to understand how we could implement and optimize genetic testing techniques.

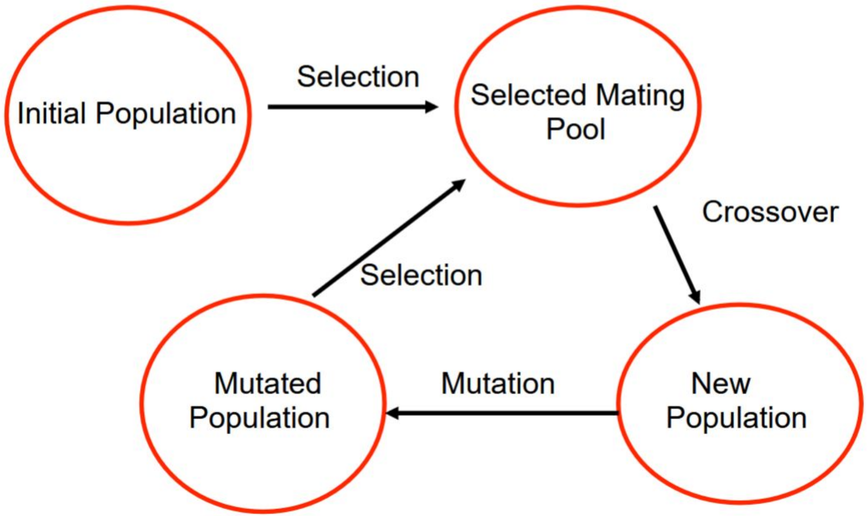
Our Tool - PyGET (Python GEnetic Testing)

```python
def fibonacci(n: int):
    a = 0
    b = 1
    if n < 0:
        print("Incorrect input")
    elif n == 0:
        return a
    elif n == 1:
        return b
    else:
        for i in range(2, n):
            c = a + b
            a = b
            b = c
        return b


gen = GeneticTestGenerator(SharedStatementFitness(), TournamentSelection())
gen.run_until(fibonacci, max_generations(10))
```

```
Testing fibonacci...
generations=1, fitness=12.999999999999995,
coverage=1.0, pop_size=64, missed_lines=[]

fibonacci(92047)
fibonacci(1)
fibonacci(0)
fibonacci(-18254)
```

Brief Overview: Genetic Testing Algorithm

- **Our Techniques:**
  - **Fitness Function:** Shared Statement Coverage instead of Traditional Statement Coverage.
  - **Selection Algorithms:** Tournament and Roulette.
  - **Crossover:** Uniform Crossover.
  - **Mutation:** Random Chance Mutation.



Flow Diagram of Genetic Testing Algorithm

Techniques and Implementation

```python
fitness_algorithm: IFitness,  # Fitness Function
selection_strategy: ISelection,  # Selection Method
pop_size: int = 64,  # Number of candidates in the population
elite_count: int = 0,  # Number of elite candidates preserved each generation
percent_candidates_preserved: float = 0.5,  # Percent of candidates preserved each generation
mutation_rate: float = 0.25,  # Mutation chance per argument
init_population_strat: InitialPopulationStrategy = InitialPopulationStrategy.INTERESTING_FIRST,  # how the initial population is chosen
pop_random_percent: float = 0.5,  # Min percent of population to be randomly created, only used if init_population_strat is MIN_PERCENT_RANDOM
interesting_chance: float = 0.5,  # Chance that a random value is an interesting value
mutate_to_new_value_chance: float = 0.025,  # On mutation, chance that the value gets reset to a random value
dynamic_interesting_values: bool = True,  # Whether to scan the function to find interesting values
type_registry: TypeRegistry = default_registry,  # The supported types
```

Interesting Values

- Some values are more likely to cause different paths to execute. We call these *interesting values*.
- Our program makes interesting values more likely to occur in the initial population and certain mutations.
- There are 2 types:
    - *Simple interesting values* are universal to every function. This includes values like 0 and the empty list.
    - *Dynamic interesting values* are different for each function. They are determined by analyzing the source code for constant values.

Interesting Values

```python
def classify(salary: float):
    if salary <= 10000:
        return "low"
    elif salary <= 30000:
        return "medium"
    else:
        return "high"
```

Interesting Values

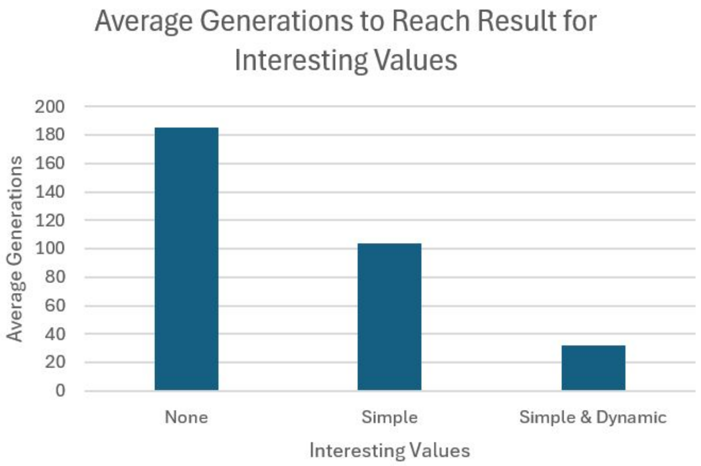
Simple Interesting Values: `0`

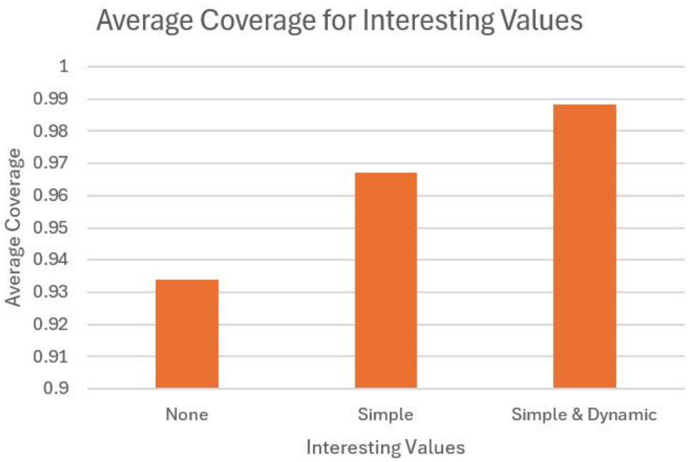
Dynamic Interesting Values: `10000`, `30000`

Initial Population: `classify(0), classify(10000), classify(30000), …`

Evaluation

- Our tool achieved an average of approximately 98.8% statement coverage on our dataset.
  - Our data was taken from a Python Instruct dataset, and adapted to use type signatures.





Q&A

Extra Slides

Shared Statement Coverage

```python
def sum(lst: list[int]):
    if len(lst) == 0:   5
        raise Exception("List is empty!") 1

    sum = lst[0]   4
    for i in range(1, len(lst)):   4
        sum += lst[i]   4
    return sum   4
```

Encourages more diverse populations.

Statement Coverage

- `sum([])`
  - fitness = 2
- `sum([1])`
  - fitness = 5
- `sum([2, 3])`
  - fitness = 5
- `sum([0])`
  - fitness = 5
- `sum([-7, 3, 1])`
  - fitness = 5

Shared Statement Coverage

- `sum([])`
  - fitness = 1.2
- `sum([1])`
  - fitness = 1.2
- `sum([2, 3])`
  - fitness = 1.2
- `sum([0])`
  - fitness = 1.2
- `sum([-7, 3, 1])`
  - fitness = 1.2

Limitations

- All tested functions must have type hints on its parameters.
- We only have built-in support for certain types.
  - Float, int, str, list, dict, bool, etc.
  - Users can add support for other types.
- The algorithm struggles when an input is needed in a very specific format.

```python
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)
```