Group Report for Team 8: PyTeG: Test Data Generation for Python

20193008 Sungmin Kang, 20150657 Sujin Jang, 20160662 Seungho Choi, 20160602 Jaehwang Jung

Project Information:

Project github repository link: https://github.com/priv-purpose/cs453-project
Account information of each team member:
Sungmin Kang is associated with the github id `priv-purpose`.
Sujin Jang is associated with the github id `sooj-j`.
Seungho Choi is associated with the github id `MoonlightRiver`.
Jaehwang Jung is associated with the github id `tomtomjhj`.

Definition of the Problem

[Written by Seungho] SBST has been successfully applied to automatic testing. However, SBST often involves simple input types, and functions with typed parameters. Therefore, it is difficult to apply SBST to dynamic languages such as Python. Python is the 4th most used programming language, and it dynamically typed and OOP, making SBST application to Python difficult. Despite this, we want to borrow the success of SBST techniques to dynamic languages, which use is become frequent recently. Therefore, we present a way to automatically generate test cases for Python through SBST, PyTeG.

Related Work

We have 4 related works.

The first one is "PP. Tonella.: Evolutionary testing of classes (2004)". It randomly generates parameters, and targets Java programs. The part related to our work is 'using genetic algorithms to generate a sequence of object instantiation & method calls that maximize the coverage criterion'.

The second one is "Ducasse et al.: Challenges to support automated random testing for dynamically typed languages (2011)". It has instance generation and execution, identifying errors, and understanding results ability. The part related to our work is 'automated random testing'.

The third one is "Mairhofer, S. et al.: Search-based Software Testing and Test Data Generation for a Dynamic Programming Language (2011)". This paper presents "Ruby Test case Generator (RuTeG)" tool that finds test scenarios that maximize the coverage criterion using genetic algorithms, and utilizes previous research on OOP test generation. This one is mainly related to our work.

The last one is "Bihel, S.: Challenges in Automated Test Data Generation for Dynamically Typed Programming Languages (2017)". It has literature review after RuTeG, and other related researches. It tried to come up with better tools and refined practical challenges.

Methodology

[Written by Jaehwang] Following the architecture of RuTeG, our implementation consists of 4 parts: analyzer, data generator, test case executor, and test case generator. The analyzer (`class Analyzer`) collects classes and methods from the input file. The data generator (`class RandomTestGenerator`) has 3 functionalities. First, it generates random values of simple primitive types `int`, `float`, and `str`. With the primitive values, it generates a simple test

case (`class Individual`) which can be used as a seed population for the genetic algorithm. Finally, it stores a set of combination a method and argument types that can lead to `TypeError` and `AttributeError`. When generating method arguments, it will try to avoid these erroneous type combinations. The test case executor is integrated into the `Individual` and `GeneticEnvironment` classes. It translates an individual test case to a Python program. The translated program is run under the `coverage.py` branch coverage tracker. If the program raises an error, the executor analyzes the type and the possible cause of the error in the test program.

[Written by Sujin] The test case test case generator is the main engine of PyTeG. It i) finds the appropriate input value of the constructor and methods of class under test and ii) generates a valid sequence of method calls. A genetic algorithm is used to perform these two tasks. The algorithm randomly generates a population and then evolves to find an individual that represents a valid test case. The individual consists of the information about the constructor to create an object of class under test, method call sequence, and method under test. The constructor contains information about the argument list. The method call sequence consists of the method name and the argument list of each method to be called. The method under test has information about the method name and the argument list. Each individual is evaluated by transforming it into an executable test case. Fitness is evaluated by the branch coverage of each individual, measured through `coverage.py`.

The single-point operator is used to crossover the argument lists of two individuals. The list after a randomly selected single position is exchanged between individuals. For method call sequences, the cut-and-splice crossover operator is used. The position is selected randomly for each individual, and the sequence after the selected position is replaced with each other. It allows dynamic change, i.e. growing and shrinking of the length of the sequence.

For mutation of the argument list, the operator generates a new type pattern or generates a new input value of the same type for one of the existing arguments. For the mutation applied to the method call sequence, the operator selects a position randomly. Then the method is removed, or a randomly selected method of the class under test is added at the position.

Evaluation

[Written by Sungmin] The objective of our work is to implement the RuTeG algorithm to work on Python and evaluate its performance relative to random inputs. Based on this objective, we formulate two research questions, RQ1 being whether our PyTeG algorithm can achieve higher branch coverage relative to random testing, and RQ2 being an examination of how crucial the type maintainer is to performance of the system.

Genetic Search Parameter Setting.

[Written by Sujin] Tournament selection method (size is 4) is used for individual selection. The crossover is applied with a probability of 1.0 and the mutation ratio is 0.2. Note that the generation number has been increased from 50 in the final presentation to 200 for this report.

Parameter	**Setting**	**Comment**
Population size	50	
Generation number	200	
Selection method	tournament	size = 4
Mutation rate	0.2	Rate of mutation for argument lists and method call sequence
Crossover rate	1.0	Rate of single-point crossover for argument lists and cut-and-splice crossover for method call sequence
Initial function sequence length	5	
Max string length	10	
Max int value	100	
Max float value	100	

Evaluation Class Dataset.

[Written by Sungmin] In a manner similar to RuTeG, we gather various classes of different complexity from the internet. A total of 9 classes and 11 functions were used. Below, a table of the classes and methods tested is provided.

Class name	**Class Description**	**Methods used**
Chessnut	A chess simulator.	apply_move; set_position
SunriseSunset Calculator	A calculator that calculates sunrise and sunset time given day and global position.	__init__
LinkedList	A linked-list implementation in Python.	remove, index

Line	Originally a line implementation.	intersect
triangle	A triangle implementation	testTriangle
Red-Black-Tree	A red-black tree implementation in Python	add
unionfind	An implementation of disjoint sets	union
stdnum.isbn	ISBN checker	validate
sut	A simple class made for testing only	f

RQ1. Can the PyTeG algorithm achieve higher branch coverage than random testing?

T-test comparison of random testing and PyTeG

[Written by Sungmin] For the PyTeG algorithm to be meaningful, it must perform better than random testing in terms of branch coverage. To this end, we compare the branch coverage results for each algorithm. Because both random testing and PyTeG are stochastic algorithms, we run each algorithm 30 times on the evaluation classes and perform a t-test to check whether the difference in performance between the algorithms is statistically significant. Below is a table of the results we obtained, comparing PyTeG to random testing.

CUT	MUT	Rand. Testing	PyTeG	p-value
Chessnut	apply_move	0	0	0.5
Chessnut	set_postiion	0.833	0.833	~ 0.5
SunriseSunset Calculator	__init__	0.656	0.634	> 0.1
LinkedList	remove	0.411	0.496	> 0.1
LinkedList	index	0.750	0.766	> 0.1
Line	intersect	0.415	0.534	**< 0.001**
triangle	testTriangle	0.587	0.657	**< 0.01**

Red-Black-Tree	add	0.730	0.740	> 0.1
unionfind	union	0.497	0.682	**< 0.001**
stdnum.isbn	validate	0.312	0.314	> 0.1
sut	f	0.857	0.851	> 0.1

Table 1. A table of results comparing PyTeG and random testing. Average best branch coverage presented. On the right, green means PyTeG performed better; red means random testing performed better.

In the two cases in which random testing performs better than PyTeG, the difference is negligible; indeed, even when the average random testing performance is better than PyTeG performance, there is no case in which the difference between the two values diverges enough to create a p-value less than 0.1. On the other hand, there are 7 cases in which PyTeG performs better than random testing, 3 of them being statistically significant. In two cases, there was virtually no difference between the algorithm results.

Distributional comparison of random testing and PyTeG

While comparison of methods using average performance is a widely used method, it obscures an important factor: what is the variance of a method, and thus can one method achieve greater performance than the other when tried multiple times even if the aggregate is similar? For example, the best performance over every 50 individuals is plotted in Figure 1 for the `Red-Black-Tree add` method, which did not show a statistically meaningful difference between random testing and PyTeG.

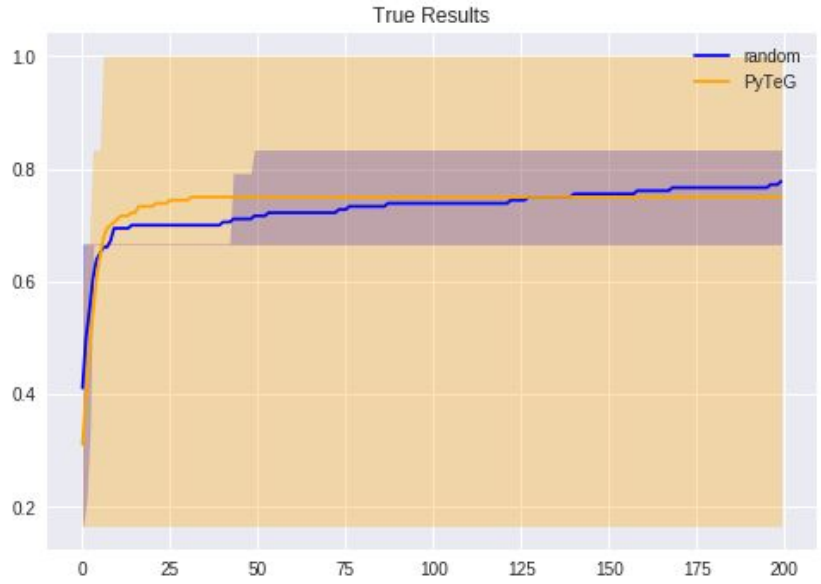

Figure 1. A plot showing the average performance of each algorithm over time. The range between the 1st quartile and 3rd quartile is shown shaded. The average performance is shown in bold lines.

Note that while the mean value is not far away between the two algorithms, the quartile ranges are significantly different. This indicates a difference in performance that is not captured by the t-test: PyTeG can either perform far better than random testing or far worse. This generally seems to be the case for all methods we examined. When aggregate performance is similar between the two algorithms, this indicates that PyTeG may yield better results over a few runs than random testing, while experiencing some worse results that keep the mean similar. Then provided enough resources, one may prefer using PyTeG to random testing in such cases.

From these experiments we may conclude that PyTeG indeed can achieve higher branch coverage than random testing.

RQ2. How important is the ‘type tracker’ to PyTeG’s performance?

[Written by Sungmin] Having obtained bad results when not using the ‘type tracker’ (the true name of this mechanism is never given in the RuTeG paper) which keeps types that caused `TypeErrors`, we compared the performance of PyTeG with and without this type tracker. The results are presented in the table below for each method that we used.

CUT	MUT	No Type Track	Type Track	p-value
Chessnut	apply_move	0	0	> 0.1
Chessnut	set_postiion	0.833	0.833	> 0.1
SunriseSunset Calculator	__init__	0.667	0.634	**< 0.05**
LinkedList	remove	---	0.496	---
LinkedList	index	0.817	0.766	> 0.1
Line	intersect	0.576	0.534	< 0.1
triangle	testTriangle	0.686	0.657	> 0.1
Red-Black-Tree	add	0.861	0.740	> 0.1
unionfind	union	0.625	0.682	> 0.1
stdnum.isbn	validate	0.289	0.314	> 0.1
sut	f	0.85	0.851	> 0.1

Table 2. A table of results comparing models with and without type tracking. Average best branch coverage presented. On the right, red means model not using type tracking worked better.

As one can note from the table above, actually there is little difference between the two models; while there is only one case statistically significant, in a majority of cases the version that doesn't track the type errors actually performs better. Then we can conclude that actually type checking doesn't contribute much to performance, and the reason we obtained mixed results for the final presentation was because the generation number was too low. Thus we can answer the research question that type tracking doesn't really matter, which is honestly surprising. (This experiment was conducted about 9 hours before submission, but the 'remove' process just won't end… that is why it does not have an entry in Table 2.)

Discussion

Comparing results with RuTeG

[Written by Sungmin] When we compare our results to RuTeG, their results seem impossible. For example, RuTeG reports that a random test algorithm could achieve 100% coverage on an ISBN validation function. How can a "random tester" generate valid ISBN numbers when generating generic random strings? For example, we used an alphanumeric character set when generating random strings which yields 66 characters overall, 10 of which are digits. Even if one made strings with only 10 characters, the odds that a string is completely consisted of digits is less than one in 100 million. The odds of a random 10-digit string being a valid ISBN number (considering the checksum condition) is about one tenth, so the odds that a random alphanumeric string is a valid ISBN number is less than one in a billion. A billion tests in six minutes does not seem plausible. Then the only way RuTeG could have such high coverage is by using a more specific random input generator, as noted in Section 4.2, which states that "a major design decision was to have different generators for specific problems." Despite this, they do not specify what generators were eventually used, making reproduction extremely difficult. (We reached out to the corresponding author, but could not get a response.) Another problem this poses is that while the authors argue that their method can achieve high coverage, this is in fact not particularly due to RuTeG, but rather thanks to the effective random input generators that the authors seemed to have made for each specific class. Perhaps it is possible to discern the problems of papers prior to experimentation and avoid putting much effort into reproducing them, but that seems to require more experience than we currently have.

Difficulties of applying SBST to Python with RuTeG framework

[Written by Jaehwang] Among the challenges that Bihel (2017) compiled, we tried to solve the problems related to the dynamic type system of Python. In short, one cannot decide what type of value to pass to the test function just by looking at the code. At first, we thought that we can

utilize the advance of the type checkers for Python such as mypy and pytype. Since most of the core libraries have type stubs which are used as a basis for type checking, our expectation was that a type checker could reconstruct type information (type inference) of the methods we wanted to test. This is how type checkers usually work for programming languages with strong type systems, like the ML-family of programming languages. However it turned out that this is not the case for Python. The type system of Python is just a thin layer which has nothing to do with the language semantics. Therefore, the type checkers function only as a sanity checker for code that is annotated manually by the programmer.

Because of the challenge described above, we chose to do post-execution analysis. Given an individual that is generated totally randomly, the executor checks if there is a type related error and marks the specific line in the test code that invoked the error. For example, given a UnionFind class for disjoint sets of non-negative integers, a line in test case that calls union(1.1, "asdf") will raise a `TypeError` and thus caught by our executor. Then the executor can remember that the `(union, (float, str))` combination is problematic and avoid it when generating a new individual. This scheme worked well for simple cases like UnionFInd. After a few executions, the generator started to generate method calls with only integer arguments.

However, this method is neither sound nor complete. Consider the following code:

```
1 obj = LinkedList()
2 obj.addToStart("bP0")
3 obj.addToStart(-41.717121457169256)
4 obj.Max()
```

When executing line 4, the `Max()` method will raise a `TypeError` while comparing the `string` and `float` value. In turn, the executor will spuriously conclude that `(Max, ())` is a problematic type combination. In fact, the root cause of the error is line 2 and 3, where values of different types are inserted into the list. In order to remedy this situation, we changed the error analyzer to remember the sequence of type combination that lead to error, namely `[(__init__,()), (addToStart, (str)), (addToStart, (float)), (Max, ())]`. While this approach is sound, it reveals more problems and challenges. First, there are too many combinations to remember. Second, mutations should consider the sequence of type combinations. Third, code that has obviously wrong argument types may not be sorted out fast enough. These challenges seem to pose an interesting venue for research.

Another approach

[Written by Jaehwang] There is an interesting approach that we could not implement during the project period. The assumption here is that we don't need to generate the test set from scratch. In this scheme, we require a small test set that scores low coverage but runs the all the functions defined in the module. Writing such tests is not a difficult task; people write simple tests frequently. A tool named `MonkeyType` made by Instagram can be used to fetch type information while running this seed test and generate a type stub for the module. We can use

the type stub to reduce the search space. Perhaps we could even evolve the seed test set on its own, negating the need for manual initial test cases.

Conclusion

We present an implementation of the RuTeG algorithm on the Python language, which we dub PyTeG. We introduce the principles of RuTeG into PyTeG, and run it on 11 methods from 9 classes. We compare the coverage results with random testing results to establish how well PyTeG performs against this baseline method. T-tests are performed that demonstrate the superiority of PyTeG over random testing. A study in which type tracking is turned off shows that actually type tracking isn't necessary and seems detrimental to performance. A discussion is provided about RuTeG paper results, the difficulties we experienced in applying SBST to dynamic languages, and a promising future direction of research.