Improved Random Test Generation using TSTL APIs

Thang Hoang

June 6, 2016

1 Project Description

1.1 Motivation

Random testing has been intensively adopted to detect potential failures in software due to its simplicity, efficiency and ease of implementation [2] compared with other complicated testing techniques [1]. Despite its surprisingly effective production of error-revealing test cases, random testing sometimes might generate a numerous of test cases which are unreachable, illegal or duplicated with others, and therefore limit its effectiveness. Specifically, based on my observation, implementing purely random testing (e.g., Breadth first search random testing) using TSTL APIs to test Python software frequently generates irrelevant actions that cannot cover every single lines of the code in the program. Meanwhile, such unreachable lines may contains potential bugs. Random testing cannot cover these lines as test cases are generated randomly and independent to each other.

1.2 Proposed Method

I propose a novel test generation algorithm using TSTL APIs. My approach is an improvement of random testing, in which subsequent random actions are selected regarding to SUT state being observed to cover and explore new statements as many as possible.

Main idea. The main objective of my strategy is that given a budget of time t and a depth value d representing the length of random actions being performed, *the frequency distribution of all statements in the program being scanned should be uniform.* Main intuition of my test generation algorithm is as follows:

First, I spend $t/2$ time from the total time budget to perform purely random testing in phase 1 to scan preliminary statements being easily covered. d consecutive random accesses are performed in this case to explore new statements. For each time when new statements being found, I store its information as well as the state that can generate to it in two different arrays. After d consecutive random testings, I measure the scanning frequency of each statement being covered.

Next, I determine k (i.e., $k = 4$) statements which is least covered as $\vec{o}$ based on a threshold τ which is defined as:

$$\tau = f_l \tag{1}$$

where f_l is the k-least frequency. So, $\vec{o} = \{s_i | f_i \leq \tau\}$ is a collection of k statements whose frequency is less than or equal to τ.

After determining $\vec{o}$, I select states of SUT that explored such statements in the first phase ordered by least coverage statement. I perform ,with the probability of 0.7, random testing of these

Algorithm 1 Improved random testing using TSTL APIs

```
    Phase 1. Random exploring
1:  while TIME_BUDGET_FOR_PHASE_1 do
2:      for $depth = 1$ to $d$ do
3:          random_action()
4:          if new statements $\vec{o_i}$ are found then
5:              store sut.state() $s_i$ in $\vec{s}$
6:              store $\vec{o_i}$
7:          end if
8:      end for
9:      Calculate frequency $f_i$ of each statement $s_i$ being found
10: end while
11: Sort the frequency with ascending order, resulting in $\mathbf{f}'$
12: $\tau = f'[k]$
13: Select states $s_i$ that generate statements whose frequency less than $\tau$: $\vec{s'} = \{s_i\}$
    Phase 2: Exploit more important states and perform some genetic algorithms (i.e., mutation,
    crossover) with some probabilities to diversify the test cases
14: while TIME_BUDGET_FOR_PHASE_2 do
15:     for $i = (1, \mathsf{len}(\vec{s'}))$ do
16:         state1 $= s'[i]$
17:         if rand.random() $> 0.7$ then
18:             for $depth = 1$ to $d$ do
19:                 random_action()
20:                 if new statements $\vec{o_j}$ are found then
21:                     $s'[i] =$ sut.state()
22:                     state1 $= s'[i]$
23:                 end if
24:             end for
25:         else
26:             test1 $=$ $get\_test$(state1)
27:             if rand.random() $< 0.75$ then
28:                 mutation(test1)
29:                 $s'[i] =$ sut.state()
30:             else
31:                 for $j = (1, \mathsf{len}(\vec{s'}))$ and $j \neq i$ do
32:                     test2 $=$ get_test($s'[j]$)
33:                     crossover(test1,test2)
34:                     $s'[j] =$ sut.state()
35:                 end for
36:             end if
37:         end if
38:     end for
39: end while
```

states attempting to explore new statements. The remaining 30% of percentage is used to perform genetic algorithms (i.e., mutation and crossover) to yield new test cases. Algorithm 1 presents my idea in detail.

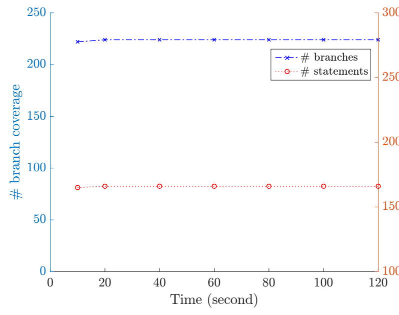
Running configuration. My program follows the default configuration being indicated. It takes 7 parameters including (time, seed, depth, width, save_failure, coverage_report, new_branch_report) as inputs to be executed, for example, as follows:

```
python2.7   finaltester.py   80   15   100   10   0   1   1   ,
```

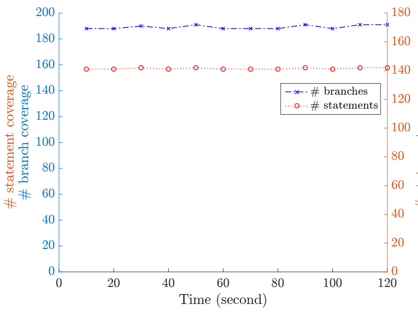
where 80 is the running time (in seconds), 15 is the random seed, 100 is the length of consecutive random test cases being generated, 10 is the width (not being used in this proposed method), etc.

1.3 Evaluation

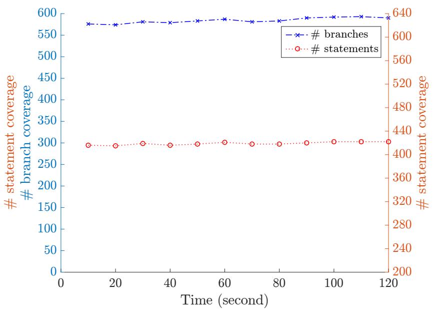
I tested my improved algorithm on various programs as shown in the "example" folder in tstl with various number of running time being allocated. Particularly, I tested on AVL with buggy and non-buggy versions, SQL parser, stack, numpy and biopython. Figures 1(a–f) illustrate the number of branches and statements being covered in such python programs by using this proposed testing algorithm. By allowing to execute genetic algorithms with some probabilities, I significantly increase the number of branches and statements being covered. Specifically, the latest program can



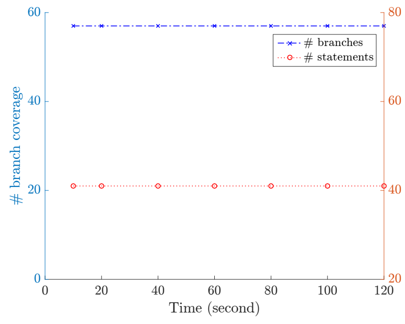
(a) non-buggy AVL



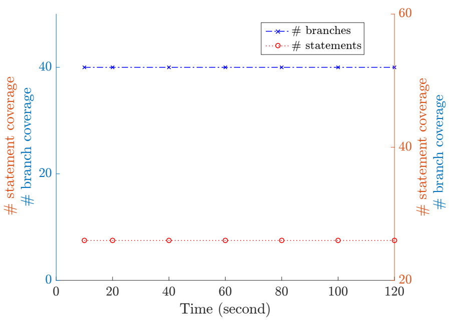
(b) buggy AVL (evil state)



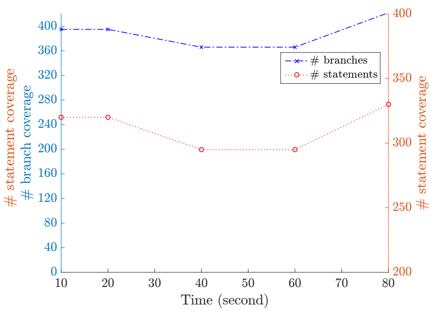
(c) MySQL Parser



(d) numpy



(e) stack



(f) biopython

Figure 1: Number of branches and statements being covered in some python programs by using this proposed method.

cover 191 branches and 142 statements in buggy AVL in 50 seconds, compared to 183 and 137 achieved with the last version.

The bug detecting capacity of the proposed testing program is also improved. It can find various bugs in buggy AVL-tree, stack, and biopython program as follows:

- buggy-AVL (evil state added):
  - running configuration:

    ```
    python2.7   finaltester.py   120   16   100   10   0   1   1
    ```

  - bugs found:

    ```
    FOUND A BUG! # 2
    (<type 'exceptions.ZeroDivisionError'>,
    ZeroDivisionError('integer division or modulo by zero',),
    <traceback object at 0x1038a7d40>)
    time:  60.2808039188
    FOUND A BUG! # 3
    (<type 'exceptions.ZeroDivisionError'>,
    ZeroDivisionError('integer division or modulo by zero',),
    <traceback object at 0x10386f3b0>)
    time:  60.2826659679
    ```

- biopython:
  - running configuration:

    ```
    python2.7   finaltester.py   80   16   100   10   0   1   1
    ```

  - bugs found:

    ```
    TRANSLATE: AGTAGAGAAGTAGAGAAAGTAGAGAAGTAGAGAAT =
    FOUND A BUG! # 4
    (<type 'exceptions.ValueError'>,
    ValueError('Proteins cannot be translated!',),
    <traceback object at 0x10a42ce60>)
    time:  52.5015630722
    FOUND A BUG! # 5
    (<type 'exceptions.ValueError'>,
    ValueError('Proteins do not have complements!',),
    <traceback object at 0x10a42b7e8>)
    time:  52.5033040047
    ```

- stack:
  - running configuration:

    ```
    python2.7   finaltester.py   60   16   100   10   0   1   1
    ```

– bugs found:

```
FOUND A BUG! # 11
(<type 'exceptions.ZeroDivisionError'>,
ZeroDivisionError('integer division or modulo by zero',),
<traceback object at 0x10b6fc560>)
time:  14.5674250126
FOUND A BUG! # 12
(<type 'exceptions.ZeroDivisionError'>,
ZeroDivisionError('integer division or modulo by zero',),
<traceback object at 0x10b6fc908>)
time:  14.5678789616
```

We can see that the number of branches and statements in biopython fluctuates a lot with different of running time and seed, compared with other programs. From our observation, the issue causing this problem happens is similar to SQLParser, in that some test cases takes longer times to be executed. Also, once a bug is found in biopython, it takes a quite long time to finish and save that test case. This issue prevents me from running the program with time being allocated larger than 80 seconds because there is a test case that makes my program hangs similar to SQLParser. Such factors caused different number of branches and statements being covered once I allocate different running time and change the random seed. For stack and numpy programs, the number of coverage is rather small compared with other programs, in that less than 60 branches and 30 statements have been found even though the running time is allocated from 10 to 120 seconds. That might be due to either the simplicity of such programs or there are such specific branches/statements that's extremely hard to be touched.

References

[1] Alex Groce, Gerard Holzmann, and Rajeev Joshi. Randomized differential testing as a prelude to formal verification. In *Software Engineering, 2007. ICSE 2007. 29th International Conference on*, pages 621–631. IEEE, 2007.

[2] Richard Hamlet. Random testing. *Encyclopedia of software Engineering*, 1994.