Hao Liu
Applying ART by using APIs in TSTL
Final report
6/7/2016

Introduction:

The mainly differences between adaptive random testing and ordinary random testing are uniform distribution and without replacement. These features not only give adaptive random testing easier mathematical model for analyzing, but also provide testing result which is closer to the reality. Therefore, adaptive random testing is believed that providing a better random testing method. Because I don't take the testing input as alone points in adaptive random testing, now we can judge the following test case is humble or not by observing the range or distance of testing inputs. In other word, in adaptive random testing, a pure random test is necessary, in order to get credible result and good performance.

In all random testing studies, only the rate of failure-causing inputs (hereafter referred to as the failure rates) is used in the measurement of effectiveness. For example, the expected number of failures detected and the probability of detecting at least one failure are all defined as functions of the failure rates. A recent study has shown that failure patterns may be classified as point, strip or block failure patterns. Intuitively speaking, when the failure pattern is not a point pattern, more evenly spread test cases have a better chance of hitting the failure patterns. Based on this intuition, we propose a modified version of random testing called adaptive random testing. An empirical analysis of 12 published programs has shown that adaptive random testing outperforms random testing significantly for most of the cases.

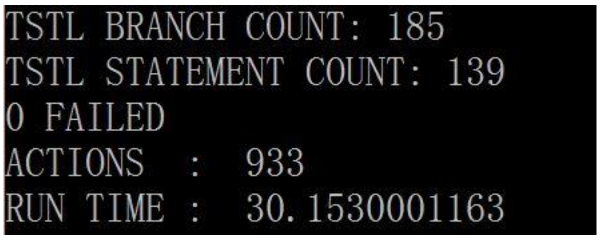
Algorithm:

The input is almost the same as tester1. I have used codes in coverTester.py. And the seven inputs are showed at the beginning of the code. TIME_BUDGET is the total time my tester will run on the sut.py. And faults are just a switch for the tester. Coverage gives the coverage report and running is just the switch for new branch discovered report. Last time I failed to put internal report in my tester and I add it this time. With internal report we can easily find the new action and which action has been tested the most time.
Within a given time we will keep doing random test until we hit a bug. The input will judge whether we should print out the new branch or not. And any new statement will be stored in the storedTest. While we are exploiting we will count the statement number as coverage.
The program will randomly test action until hitting a bug. And any new actions will be stored as new statement in order to count the depth of the test. By considering the

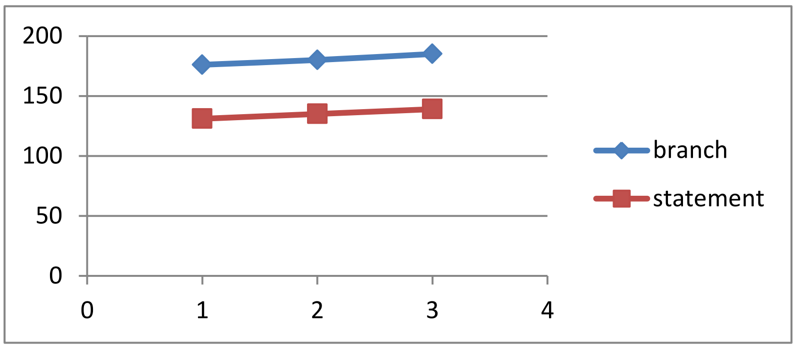
coverage and depth we can figure out whether my tester is efficient or not.
Another limitation for the whole program is width, which means how many states have been visited during the testing. And for the same level of state the tester will decide if a new action appears or it is an enabled action.
One method or multiple methods with arguments are randomly picked to generate a sequence. The randomly generated sequence is append to the previous sequence to create new sequence. This new sequence is checked if it is already present or not. Contracts are used to find errors in the system and filters are used to avoid bad tests.
In milestones, I have added code for random testing of creation of new branches. This code would test sut.py for the time passed as an argument. In this code if the argument running is set to 1, then I am checking for new branches in sut. If it is not equal to null set, then I iterate over the branches and print the time taken along with total branch count and the new branch in every iteration. If it is null then I do nothing

Experiment result

The test generation is able to be run in the command line which allows us to specify different parameters (timeout, seed, depth, width, faults, coverage and running) with it. To run the test generation on the command line we need to type for example: python finaltester.py 30 1 100 1 0 1 1. The first parameter is the timeout which the test generation will stop when the runtime reaches that number.
The second parameter is the seed which generates a random number of seeds. The third parameter is the depth and the fourth parameter is width. The fifth parameter is to check for fault when it is 1. 1 means the test generation will search for bugs and 0 means the test generation does not look to find bugs. The sixth parameter is printing TSTL internal coverage report to know the statement and branches were covered during the test to guide it. This will work when we specify 1 and 0 for do not print it.
The last parameter is for running which will print the elapsed time, total branch count and new branch. It works when we put 1, 0 will not do anything.
Here is the result of final version:


```
TSTL BRANCH COUNT: 185
TSTL STATEMENT COUNT: 139
0 FAILED
ACTIONS  :  933
RUN TIME :  30.1530001163
```

And a comparison between the 3 tester:



The optimization of code works and I can achieve high performance in the final version

Conclusion

Random testing is simple in concept and is easy to be implemented. Besides, it can infer reliability and statistical estimates. Hence, it has been used by many testers. Since random testing does not make use of any information to generate test cases, it may not be a powerful testing method and its performance is solely dependent on the magnitude of failure rates. As a result this tester is not pure ART algorithm, but something mixed with the content studying in the class. What I get is better than the pure random testing and worse than swarm testing. This is not a bad result and if I have chance I may spend time on the optimization again.

Reference

[1] Chen T Y, Leung H, Mak I K. Adaptive random testing[M]//Advances in Computer Science-ASIAN 2004. Higher-Level Decision Making. Springer Berlin Heidelberg, 2004: 320-329.
2009, pp.87 – 96
[2] A. Groce; J. Holmes; J. Pinto; J. O’Brien; K. Kellar; P. Mottal, “TSTL: The Template Scripting Language,” in International Journal on Software Tools for Technology Transfer.