CS 569 — Final Report

Prof. Alex Groce

Tso-Liang Wu (932-227-899)

Introduction

According to the lecture material, we know that Random Testing has its advantages. Basically, it is a very effective and intuitive way to test a program or functions. By uniformly choosing test cases from operational profiles, the random testing mostly has good features of reliability and statistical estimates. In random testing, we only care about the rate of faille-causing inputs and take it as the most important factor of the effectiveness measurement. However, according to T.Y. Chen, H. Leung and I.K. Mak's paper "Adaptive Random Testing," we found that the geometric pattern of the failure-causing inputs is also another important factor when considering the testing performance.

According to the research, 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 which has up to 50% better than traditional random testings, so this is also the reason why I decided to pick adaptive random testing as my project topic.

By classifying the patterns of failure-causing inputs to threes different types (which are point, strip and block), we can modify the ordinary random testing to adapt random testing and get closer inspection. Because we are not solely 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, we should avoid picking neighbor test cases continuously, so that we can avoid meaningless random testing, and improved the testing performance efficiently. To implement this concept, we can easily separate test cases to multiple different groups, and then pick the best case from each group to test the program.

The research mentions a specific way to implement adaptive random testing, which I think is a good method that I can also implement in my project. We can separate two disjoint sets for random testing, which are executed set and candidate set. Firstly, we randomly pick a test case from input domain without replacement and put into a set, called candidate set. Then, we pick the farthest away test case from candidate set as our next testing case. If we didn't reveal any failure, then we put this test case into another set, called executed set.

In order to find the farthest away test case in candidate set, I should clearly define a measurement to calculate the distance in my project. In addition, I should also figure out how to construct the candidate set for adaptive random testing.

Implementation

My algorithm is basically based on the cover tester algorithm. In this implementation, we should also support seven different options: timeout, seed, depth, width, faults, coverage and running. For the timeout option, we should allow testers to setup testing time freely. For the seed option, owning to my algorithm is based on the random testing, we should use this option to set the seed of Random.random() in python. In the depth option, we use this parameter to set the limit of how deep our tester could test. Similarly, in the width option, we use this parameter to narrow the testing consumed memory. In the faults option, we can either assign it to 1 if we want to check for faults, or assign it to 0 in opposite. For the coverage option, we can also either use 1 to product the final coverage report or 0 for purely generate the test cases. Finally, the running option control whether the running info on branch coverage should be produced or not.

In the original cover tester, we only use one single phase to generate test cases, so I was trying to implement multiple phases like new cover algorithm does. By using repeated phases, we should ideally generate the testing cases which cover much wider. According to this idea, I tried to use different lists to monitor test cases I've tested, and generate different new test case in the next phases. Theoretically, we could generate wider testing cases each time under the same limit of budget. Beside using multiple phases to test wider, I also apply the count taken algorithm into my algorithm. I enable the random tester and create a random action list before finish each phase. By using this algorithm, we can try to equal the testing actions as hard as possible, so that to avoid the extreme testing possibility. Also, some features which also be implemented in the final tester as following:

1. Time out: the time limit in seconds for testing
2. Seed: seed for Python Random.random object used for random number generation
3. Depth: maximum length of a test generated by your algorithm
4. Width: maximum memory/BFS queue/other parameter that is basically a search width
5. Faults: either 0 or 1 depending on whether your tester should check for faults in the SUT
6. Coverage: either 0 or 1 depending on whether a final coverage report should be produced, using TSTL's internalReport() function.
7. Running: either 0 or 1 depending on whether running info on branch coverage should be produced.

In the currently version, the multiple phases for random testing is not completely implemented. I have tried to separate two phases having half of budget time respectively. However, owning to I didn't successfully figure out one effective way to avoid the testing cases which I've done in the previous phase, so the current outcome is pretty similar to the last version. In order to find a reasonable and effective what to separate multiple ways, I've tested multiple ways like make two phases totally equal to half of timeout, or the second phase would be half time of the first phase. However, there is still no strong evidence shows that it is an effective way to improve current algorithm.

References

1. Chen, T.Y., Leung H, Mak I.K.: *Adaptive Radom Testing*