CS 569 — Competitive Milestone 1

Prof. Alex Groce

Tso-Liang Wu (932-227-899)

Introduction

In the previous proposal, I was trying to improve the current random testing based on the Adaptive Random Testing algorithm. According to current testing research, adaptive random testing is an effective and reliable way to improve random testing. However, even though there are some implementations which apply adaptive random testing, I realized that it is a little bit difficult to implement a new improved adaptive random testing by using TSTL API. Therefore, instead of improving adaptive random testing, I was trying to combine features from different SUTs algorithms and improve current cover tester algorithm.

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.

Plan

Owing to I didn't completely implement the multiple testing phases algorithm yet, I will mainly continuously working on implement this algorithm before next milestone due. In addition, I am guessing that by applying different sorted algorithm, it is possible to improve the current count taken algorithm in second part. So I will also focus on improving this part if possible. Finally, I hope I could figure out some methods to effectively document the coverage of the testing cases, so that I could read wether my testing generator is much wider than the original algorithm.