Project Final Report

Zheng Zhou
Student ID: 932463162

June 5, 2016

1 Introduction

This report is for reporting the Final version of my test generation algorithm. In my proposal, I was intending to implement Adaptive Random Tester using TSTL. Adaptive Random Testing algorithm is quite effective and efficient for finding failures based on random testing. Nevertheless, I felt that it is hard to implement and improve Adaptive Random Testing using TSTL API. Thus I plan to consider another simple but also effective algorithm for this project. In the first milestone, I considered to implement a variant of breath first search algorithm, which is quite effective and comprehensive algorithm for software testing. I was trying to improve BFS in the way that sets time limits on each level traversal. But after several attempts, I found that this algorithm cannot get decent result in a small amount of time budget. Besides, setting time limits on each level of test tree can really lose a great deal of coverage and state information. Thus in general, it is predictable that this algorithm will not work well on checking scripts. Due to this, I completely changed my thought on the algorithm. I was trying to explore another way that statistically makes sense on testing. I was inspired in class by the way to mutate populations and the paper that talks about *Genetic Algorithm*[1]. Thus in the second milestone I updated my test generation. My new test generation is based on random tester. The idea behind my new test generation is to apply mutation operator and crossover operator on the initial pool of tests that have been created. When the testing starts, tester uses one third of total time budget to perform pure random testing for collecting population. After getting the populations, tester will perform mutation and crossover on the populations, until they are fully "evolved" or timeout. Expectedly, This algorithm could get quite enough coverages and detect the bugs. Following sections will mainly discuss the implementation of my test generation and the results. In the implementation part, I will first introduce the formal final version of my tester, then present the more flexible version that can take more parameters.

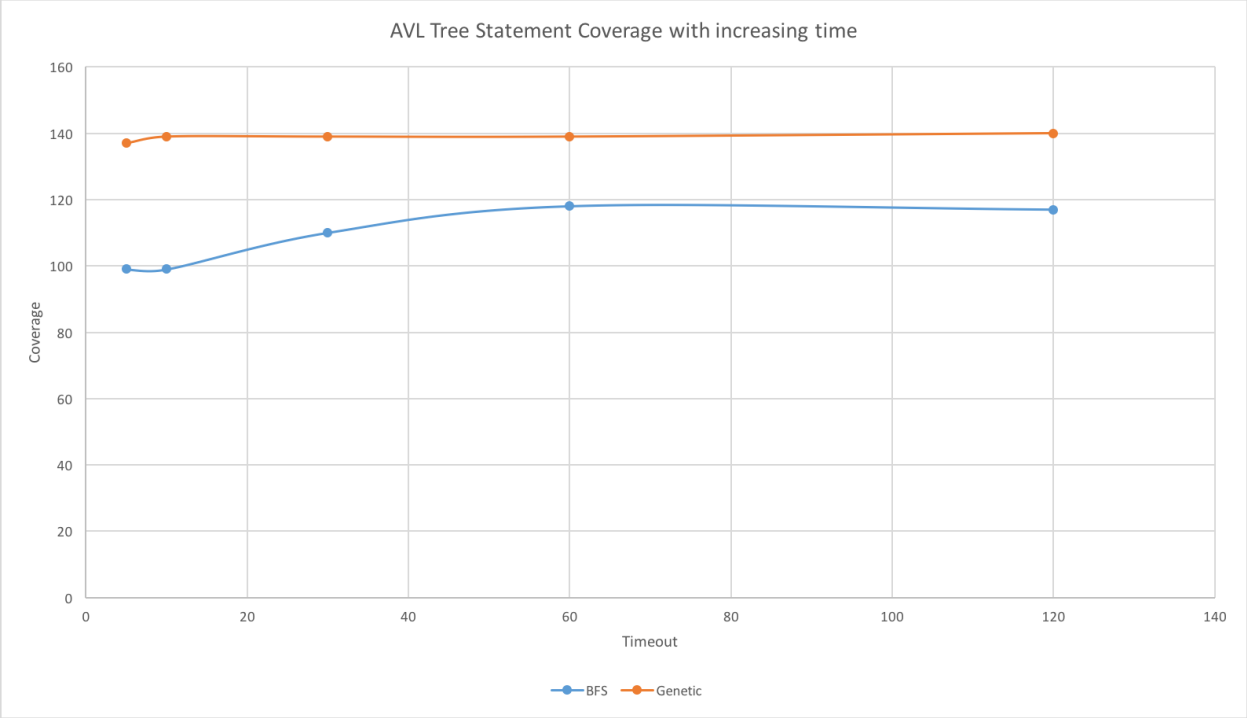
2 Implementation

The implementation of my algorithm is straightforward and similar to the method described in *Genetic Algorithm*[1]. However, I have to set extra parameters for both mutation and crossover operators, hence the complete version will be implemented in the flexible version "mytester.py". Next I am going to talk about the formal finaltester first
In finaltester, all the parameters are stored in **Config** variable, which is same as previous testers. There is a new introduced list called population, which is used to store the tuple of tests and

the corresponding statements. This list will be the main structure that is applied mutation and crossover operators. **population_time** is the time limit for collecting the populations. After these time elapsed, we actually start with mutation and crossover. The population is actually the representation of initial guess. By using random tester, we are randomly guessing the solution of testing problem. All tests and statements will be appended after perform **safely** function. One thing needs to note is that for some SUTs the initial guess really works good enough, which causes after mutation and crossover there is no much changes on coverages. This is reasonable becuause for simple SUT random testing is already a good solution. The crossover and mutation stage will consume two third of total time budget to run. In the while loop, I first sorted population list based on the length of statements with reverse order. Frankly the order does not matter since this version I only randomly choose the tests that need to do mutation or crossover. **crossover** function will take six arguments. The first two arguments are the specific tests that do crossover. The rest of arguments are just global configuration. The actual work that **crossover** function does is exchanges two tests and combine them to one tests and return. This is quite similar as genetic crossover. Inside **crossover** function we still need to check the faults. Similarly, **mutation** function merely takes one test and changes it to create a new one. Both mutation and crossover choose random number of actions to do the work. After both operations done, tester will append the results back to population list and perform next iteration.
In mytester, everything works similar as finaltester, except mytester have two more parameters which indicate the number of best or worst populations and the mode of selection. Actually selection operator that decribed in *Genetic Algorithm*[1] has been treated as a individual function. In my implementation I treat it as a program parameter which will be more easy to understand. The mode of selection has three choices: best, worst, random. Best and worst tells tester if you want to choose the population in descent or ascent order. With best mode, the population list will be sorted descent order in which you will choose the test that has more statements covered. Worst works oppositely. Random just works same as finaltester.
Above is the implemtation of my test generation. The next section will discuss the result I got.

3 Result

The SUT that I was doing experiment for my tester is mostly AVL tree. Since AVL tree itself is not a long program, the results I got from both testers are quite similar, which is understandable. But interesting thing is that in the end of collecting stage, I got the statements coverage with **134** and branches coverage with **178**. After mutation and crossover, I got the statements coverage with **185** and branches coverage with **139**. This shows that the mutation and crossover operators did work. They can actually improve from random tester in the initial stage. And this result was runing under 60 seconds. 20 seconds on population stage and 40 seconds on mutation and crossover stage with total executions 430. Following graph compares the statement coverage of my first version of BFS with the final version of Genetic Algorithm:



This graph clearly shows the new algorithm beats my initial thought to improve BFS. As the paper described, Genetic algorithms are often used for optimization problems in which the evolution of a population is a search for a satisfactory solution. In conclusion, this algorithm takes advantages of initial population to evolve for finding better solution of a problem, which is absolutely effective for testing purpose. My variant of genetic algorithm separate the whole algorithm into two stage: population stage and mutation&crossover stage. And I think this algorithm still has space to improve somehow.

References

[1] T.-h. K. Praveen Ranjan Srivastava. Application of genetic algorithm in software testing, 2009.