CS 569 - Part 4 – Prof. Alex Groce

By: Eman Almadhoun – 932909951

Introduction:

Nowadays, computers are very important part in our life which needs many programs to run on them. But for ensuring the reliability and quality of these programs, we need to test them to ensure that there are no faults. Software testing plays a very important role in the software development life cycle. We need Software testing to ensure the quality and reliability of the software and caught any bugs in the system and analyze them. Doing that by hand is much harder, detecting all the bugs are impossible and takes so long time. So, we need an automated tool for generating the unit test to find bugs in very short time with the lowest cost. Test generation is a test that generates set of test cases, each of them has a sequence of statements which depend on criterion specified [4][5].

It seems that creating test generation for testing the Software Under Test is a bit hard with misleading a perfect technique. Not every test generation can cover all the statements and branches of the SUTs which is very important to cover them all for detecting the bugs. Especially when you want to get to the narrow paths for finding any missed bugs on them. With using a good algorithm or some knowledge of machine learning, the test generation might be better to find most failures and cover most the code. It is very important to create a test generation with a minimum cost. As Prof. Alex explain that in the machine learning there is a technique that we can explore thing which looks at an interesting data to collect and exploit which means we get to this data using some statistics to find more bugs.

My first idea was using seeding strategies for solving the testing problems and finding bugs from previous knowledge in very short time but as the time limitation and the heavy load of other courses I searched to a simpler idea which based on Professor Alex ideas from the class. So, I worked on exploring and exploit method for finding the failures in the Software Under Test and put a lot of affording to maximize code coverage. In this project, I introduced a method that using code coverage to measure the least covered branches [1].

The Algorithm:

This project aims to maximize the coverage on the Software Under Test which will make the test generation is more effective for finding bugs. Branch coverage is my goal which I focus on my test generation. The algorithm which I used is focusing on a part of guiding the test cases using coverage data. In my algorithm, firstly I will save the state of the current test and do 20 % backtrack on the save test to get any useful action which might help in my experiments and then I collect the branches coverage and find the least coverage in the current branches. Then, I run pure random tests which generate random tests with random actions. After that, I look up for any failure while the tests cases are generated. Failures mean the actions in TSTL are not ok and have errors. TSTL is Template Scripting Language which will check on the actions for the SUT and check on properties but for my tester just check for actions. Before I was checking for properties but I removed it as Professor Alex said it will effect on the coverage because some student did not check for properties. If there is any bug, the tester will save the test cases in a file named failure”i”.test in the current directory. After that, I will take the name of all actions and sort them by their action count to analyze the data. If the running time reaches the specified time by the users, the tester should stop and give the user test results.

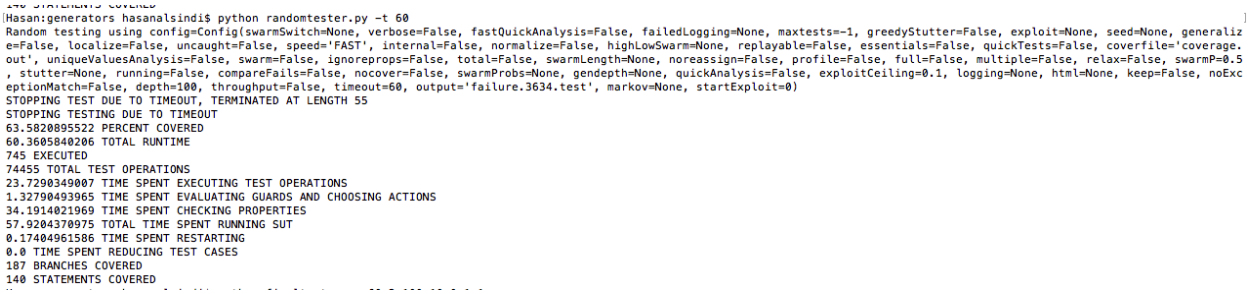
Run Test Generation:

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 do 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.

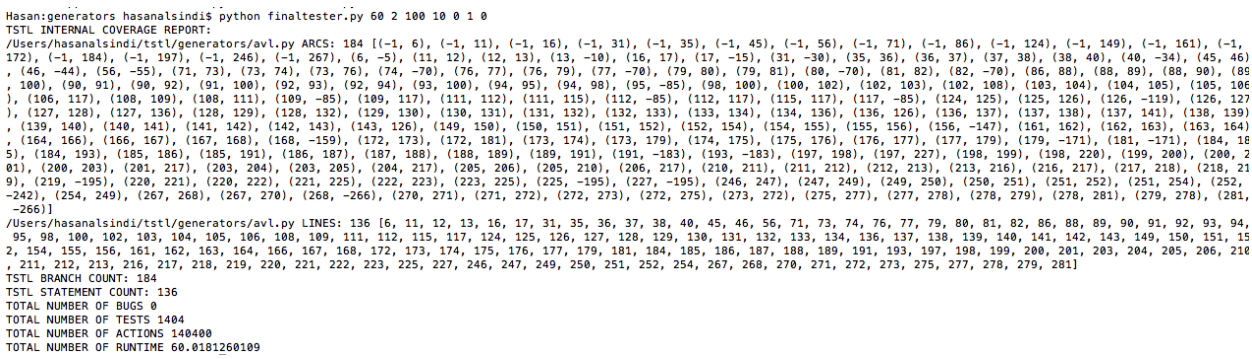
Now, I have another configuration for my tester in another file called mytester.py. It accepts parameters like the random tester and the idea is from randomtester.py. There are default settings, so you might do not need to write any parameters. The default values are 60s for timeout(-t), 100 for deep(-d), 100 for width(w), seed (s) None and False for running(r), faults(f), check for property(p) and coverage(c). You can change the setting, for example, python mytester.py –t 120 –s 10 –d 120 –w 10 –f -p –c –r. This mean the running time is 120 seconds, the random seed is 10, the depth is 120, the width is 10, the tester can check if the actions and the property are not ok, and print running and coverage reports. I was thinking to implement another parameter in mytester but due to time limitation, I could not add any more to it.

The Experiments Results:

In my tester, I tried to maximize the coverage but I could not beat randomtester.py. In depth 100, 60s running time on avl.py and 40% backtrack the saved tests. Randomtester covered 187 branches and 140 statements as the screenshot shows below:



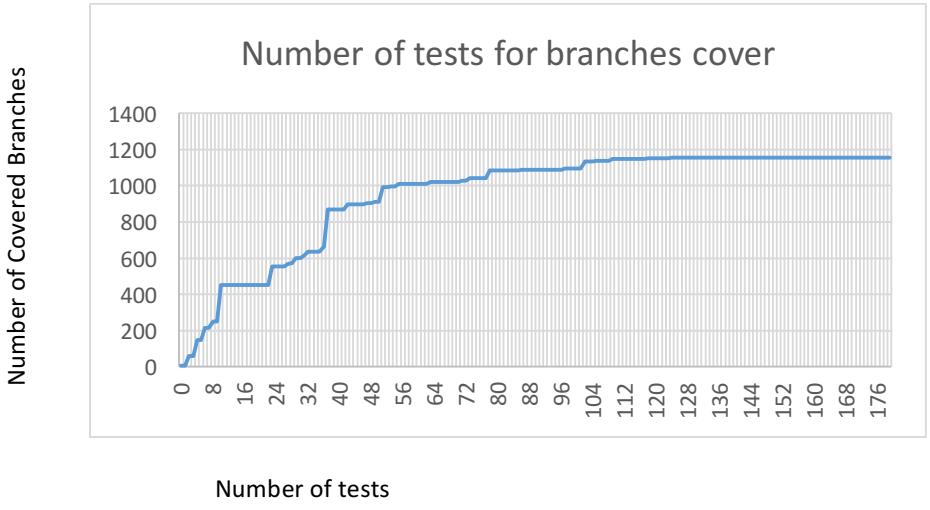
While my tester covered184 branches and 136 statements with the same depth and running time as shown below.



During my running the tester, I observed that the test coverage increase when we increase the depth, for example coverage with deep 100 is better than depth 10. I tried to improve the coverage using mutate function as Professor Alex implemented in the class. The coverage statements and branches improved as randomtester.py but my tester crash (integer division or modulo by zero) when I find the bug, so I inactivate it because the crash. The tester can find bugs with different SUTs. The figure below shows how the tester can find bugs with 60 seconds runs.

```
Hasan:generators hasanalsindi$ python finaltester.py 60 10 100 10 1 0 0
FAILURE FOUND.....FAILURES ARE STORING IN FILES
Number bugs found is 1
FAILURE FOUND.....FAILURES ARE STORING IN FILES
Number bugs found is 2
FAILURE FOUND.....FAILURES ARE STORING IN FILES
Number bugs found is 3
FAILURE FOUND.....FAILURES ARE STORING IN FILES
Number bugs found is 4
FAILURE FOUND.....FAILURES ARE STORING IN FILES
Number bugs found is 5
FAILURE FOUND.....FAILURES ARE STORING IN FILES
Number bugs found is 6
FAILURE FOUND.....FAILURES ARE STORING IN FILES
Number bugs found is 7
TOTAL NUMBER OF BUGS 7
TOTAL NUMBER OF TESTS 1473
TOTAL NUMBER OF ACTIONS 147300
TOTAL NUMBER OF RUNTIME 60.0194971561
Hasan:generators hasanalsindi$
```

To show how many times does the test cover from the least to the most, I stored the data in a file and I analyzed it using Microsoft Excel as below:



This graph shows that how many times did the branches covered from the least to the most during the tests. I observed that the at the start, the tests are so hard to cover new branches while at the end the graph shows that branches are easy to covered as the line graph get stabled and have a similar result.

Related Work:

There are different methods to measure test cases distance, Euclidean as proposed by Balcer and Ostrand [5]. Also, there is Jaccard measurement introduced by Jiang et al. [6] Jaccard method is based on this formula $D(a,b) = 1 - |A \cap B|/|A \cup B|$, where a and b are two test cases A and B are sets of branches or statements. I tried to use this method but it did not work. It leads to crash my tester because zero division problem so, I changed the tester a little bit. In [1], Zhou measured the distance between test cases using coverage to guide test cases in Adaptive Random Testing to improve detecting bugs. The author proposes a new method for measuring the distance between two arbitrary test cases which called Coverage Manhattan Distance (CMD). CMD is given by $CMD(x,y) = \sum_{i=1}^{n} |x_{i-} y_i|$ where x and y are test cases. He sets a flag to 1 if the branch has covered or 0 if not. My tester is quite similar to this idea which is using branch coverage to measure the least covered branches and sets a flag to True if the branch covered. Also, this algorithm did explore method to backtrack the save tests. This idea comes from Professor Alex during the class. I tried to use mutate to increase the coverage for enhancing failed detection. It causes coverage improvement but it does not work properly when finding bugs.

References:

1. Z. Zhou, "Using Coverage Information to Guide Test Case Selection in Adaptive Random Testing," in 34th Annual IEEE Computer Software and Applications Conference Workshops, 2010.
2. B. Jiang, Z. Zhang, W. K. Chan, and T. H. Tse, "Adaptive random test case prioritization," in *Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society Press, November 2009, pp. 233–244.
3. 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.
4. Sao N., Patel N., "A Survey on Automated Test Data Generations by Using Hybrid Approach", International Journal of Innovative Science Engineering & Technology (IJISET), 2015, Vol. 2 Issue 4.
5. S Rojas1 J., Fraser G., Arcuri A. Seeding strategies in search-based unit test generation. Software Testing, Verification and Reliability (STVR) 2016. T. J. Ostrand and M. J. Balcer, "The category-partition method for specifying and generating functional tests," Communications of the ACM, vol. 31, no. 6, pp. 676–686, 1988.
6. B. Jiang, Z. Zhang, W. K. Chan, and T. H. Tse, "Adaptive random test case prioritization," in Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering. IEEE Computer Society Press, November 2009, pp. 233–244.