CS569: Static Analysis and Model Checking for Dynamic Analysis
Part 4: Final Report

Hafed Alghamdi
Email: Alghamha@oregonstate.edu

Introduction

Generating test cases for a Software Under Test (SUT) is not an easy task as there are many approaches and techniques that can be adopted depending on the SUT and the main objective behind the testing. Moreover, it will be more challenging to generalize the test generator on deferent SUTs because of the nature of these SUTs and its level of complexity. After some research, we found that the most easies and effective approach is Random test generator because of its simplicity, fast execution of actions and great capabilities in finding faults. Therefore, in this project there are two algorithms have been implemented those depends on random test generator and adapt an idea of building-up test cases. The first algorithm's called "PROP" that combines both sequential and random tester techniques based on randomly selected probability to replay back good test cases those are saved whenever new branches get discovered. The aim of this algorithm is searching for faults as quickly as possible by taking advantage of simple random generator speed and efficacy. The second algorithm's called "Grouping" that uses the same idea of selecting random probabilities on the saved test cases to be replayed back but instead of using the sequential algorithm, we make the algorithm focus on randomly selected group of actions for some time. These algorithms can be used through an easy interface that will be described in "Mytester Usage Instruction" section.

Random/Sequential Algorithm Based on Replaying Saved Good Test Cases (PROP)

PROP algorithm combines both sequential and random tester techniques to search for faults as quickly as possible. Since, there are some actions in the SUTs could trigger a fault by just executing them with any random values, then it is a good idea to execute all the actions sequentially at least once before applying any other techniques. Thus, sequential search has been adapted in this algorithm as a first step to execute TSTL Enabled Actions only once. The second step is generating random test cases based on Random probability selection that satisfies 0.5 probability condition. It is a normal implementation of any random tester with the addition of saving good test cases in memory whenever new branches got discovered to be replayed back based on random selection of a probability that is less than 0.5. Testing this on the modified AVLTree SUT file reveals that detecting the combination lock problems is promising. *Figure 1* includes some parts of the code shortened to explain PROP algorithm:

```
startTime = time.time()
# Combined Sequntial and Random tester algorithm
for act in sut.enabled():
            seq = sut.safely(act)

sut.restart()
rgen = random.Random(seeds)
action = None
goodTests = []
# RandomTester based on randomly selcted propability
while (time.time() - startTime <= timeout):
            # Based on propability replayback saved good tests those are saved when new branches
            # got discovered. It is good for finding combanition luck faults
            if (len(goodTests) > 0) and (rgen.random() < Prop):
                        sut.backtrack(rgen.choice(goodTests)[1])
                        if (time.time() - startTime >= timeout):
                                    break
            else:
                        sut.restart()
            # Based on the depth randomly execute an action
            for s in xrange(0,depth):
                        action = sut.randomEnabled(rgen)
                        r = sut.safely(action)
                        # When getting new branches, save the test case into goodTest
                        # list to be re-executed based on random propability
                        if ((length != 0) and ((len(sut.newBranches()) > 0) or (len(sut.newStatements()) > 0))):
                                    goodTests.append((sut.currBranches(), sut.state()))
                                    goodTests = sorted(goodTests, reverse=True)[:length]
                        # Cleanup goodTest list based on the length of the goodTests
                        if (length != 0) and (len(sut.newBranches()) == 0) and (len(goodTests) >= length):
                                    RandomMemebersSelection = random.sample(goodTests,int(float((len(goodTests))*.20)))
                                    for x in RandomMemebersSelection:
                                                goodTests.remove(x)
```

Figure 1

Random Actions Grouping Algorithm Based on Replaying Saved Good Test Cases (Grouping)

This algorithm is almost similar to the previous one. However, instead of using sequential algorithm, we are grouping actions randomly and executing them based on 50% of the group length. Whenever, there is some tests cases saved in the list, it will be replayed back based on satisfying the random probability selection. This technique is also promising for the combination lock in AVLTree and other examples. *Figure 2* includes some parts of the code shortened to explain Grouping algorithm:

```
sut.restart()
rgen = random.Random(seeds)
action = None
TimeElapsed = time.time()
# RandomTester based on randomly selcted probability
while (time.time() - startTime <= timeout):
        # Based on propability replayback saved good tests those are saved when
        # new branches got discovered. It is good for finding combination luck faults
        if (len(goodTests) > 0) and (rgen.random() < Prop):
                sut.backtrack(rgen.choice(goodTests)[1])
                # Based on the depth execute an action randomly building on the previously executed savedTest
                for s in xrange(0,depth):
                        action = sut.randomEnabled(rgen)
                        r = sut.safely(action)
                        # When getting new branches, save the test case into goodTest
                        # list to be re-executed based on random probability
                        if ((length != 0) and ((len(sut.newBranches()) > 0) or (len(sut.newStatements()) > 0))):
                                goodTests.append((sut.currBranches(), sut.state()))
                                goodTests = sorted(goodTests, reverse=True)[:length]
                        # Cleanup goodTest list based on the length of the goodTests
                        if (length != 0) and (len(sut.newBranches()) == 0) and (len(goodTests) >= length):
                                RandomMemebersSelection = random.sample(goodTests,int(float((len(goodTests))*.20)))
                                for x in RandomMemebersSelection:
                                        goodTests.remove(x)
        # This part will randomly selects a group of actions and continue testing them based on a given probability
        else:
                sut.restart()
                # Select a group of action
                groupActions = random.sample(sut.enabled(),int(len(sut.enabled())* Prop))
                # Calculate the length of groupActions and take some percentage to be the depth
                for s in xrange(0,int(len(groupActions) * Prop)):
                        if (time.time() - startTime >= timeout):
                                break
                        # Use the actions them one by one
                        action = groupActions[s]
                        # Execute The actions one by one
                        r = sut.safely(action)
                        # When getting new branches, save the test case into
                        # goodTest list to be re-executed based on random probability
                        if ((length != 0) and ((len(sut.newBranches()) > 0) or (len(sut.newStatements()) > 0))):
                                goodTests.append((sut.currBranches(), sut.state()))
                                goodTests = sorted(goodTests, reverse=True)[:length]
                        # Cleanup goodTest list based on the length of the goodTests
                        if (length != 0) and (len(sut.newBranches()) == 0) and (len(goodTests) >= length):
                                RandomMemebersSelection = random.sample(goodTests,int(float((len(goodTests))*.20)))
                                for x in RandomMemebersSelection:
                                        goodTests.remove(x)
```

Figure 2

Algorithms Evaluations

Test generator algorithms evaluation is not an easy task as their inputs differ from each other and some of them are good in testing some SUTs while others are good at testing some other SUTs. However, in this section we will be trying to evaluate the algorithms by comparing them with some existing well-known algorithms in TSTL such as randomtester.py and SWARM using their default values. However, we are going to choose different time budgets to measure theses algorithms performance. There are four SUTs we have used in this evaluation to get an idea of how these algorithms behave in comparison to these well-known algorithms. The following are the SUTs used from the examples given in class and some other examples available in TSTL Examples folder.

1. ***tictactoe.tstl :***
   To test this SUT we have enabled the property check option on all algorithms. We used different time budgets to see how the algorithms would behave. The rest of default options are not changed and we are using a value of a hundred for both Depth and Length and the seed will be the default which is zero. In *Figure 3* and *Figure 4* we can see that after running all algorithms multiple times and taking the best results, we can see that all algorithms are behaving the same way and getting the same number of statements and branches while increasing the time budget (There was a 0.1 added to curve values to distinguish between the colors). However, when we check for the time it takes each algorithm to detect

the fault in *Figure 5*, we can see that grouping algorithm has the least time period to detect the fault after running each algorithm multiple times and SWARM has the longest period. We believe the reason that SWARM takes longer period than the others is that it spends some time for initialization and making some decisions before doing extensive work that causes the algorithm unable to detect the fault earlier.

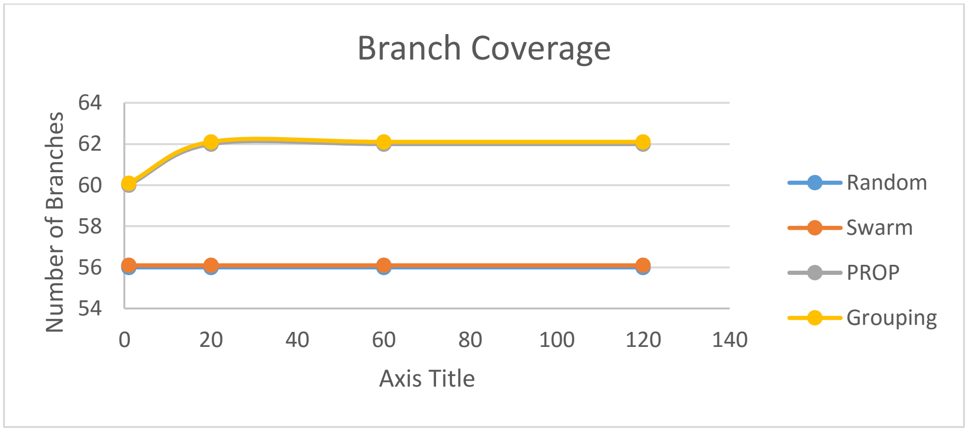

Figure 3

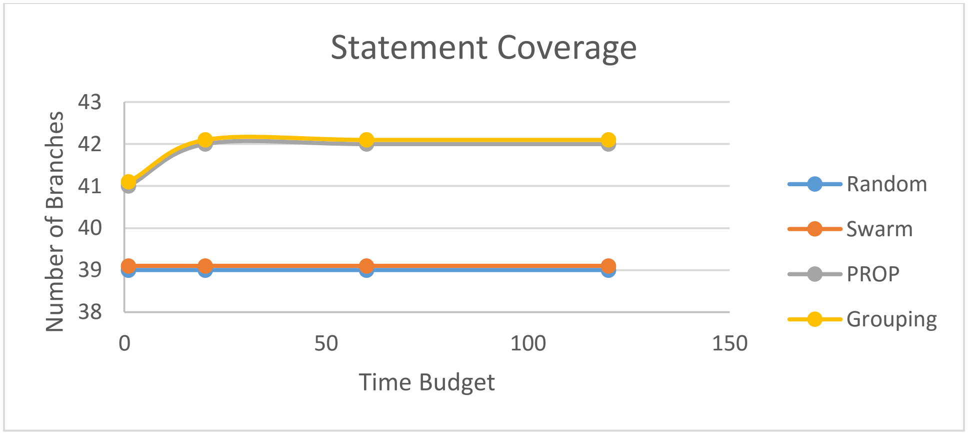

Figure 4

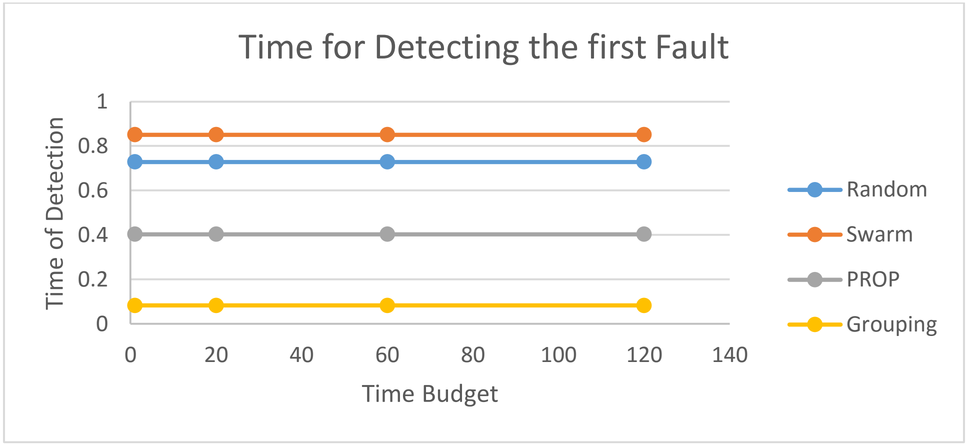

Figure 5

2. ***stack.tstl:***

To test this SUT we have enabled the property check option on all algorithms. We used different time budgets to see how the algorithms would behave. The rest of default options are not changed and we are using a value of 100 for both Depth and Length and the seed will be the default which is zero. In *Figure 6* and *Figure 7* we can see that after running all algorithms multiple times and taking the best results, we can see that all algorithms are behaving the same way and getting the same number of statements and branches while increasing the time budget. However, again when we check for the time it takes each algorithm to detect the fault in *Figure 8*, we can see that prop algorithm has the least time period to detect the fault after running each algorithm multiple times and SWARM has the longest period. We believe the reason that SWARM takes longer period than the others is that it spends some time for initialization and making some decisions before doing extensive work that causes the algorithm unable to detect the fault earlier.

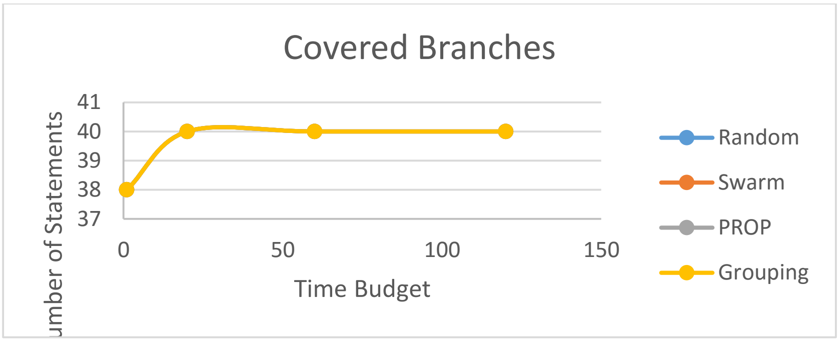

Figure 6

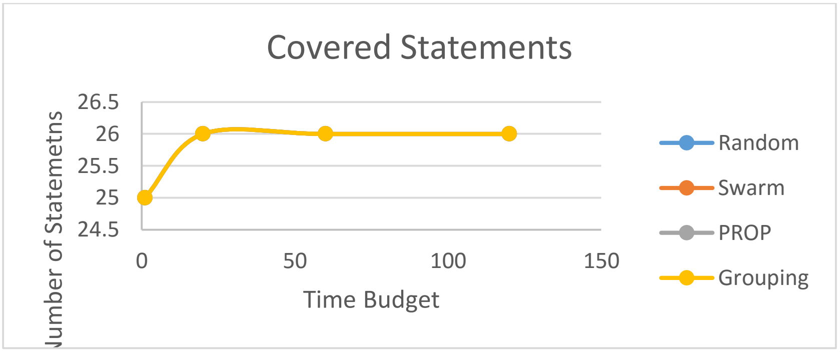

Figure 7

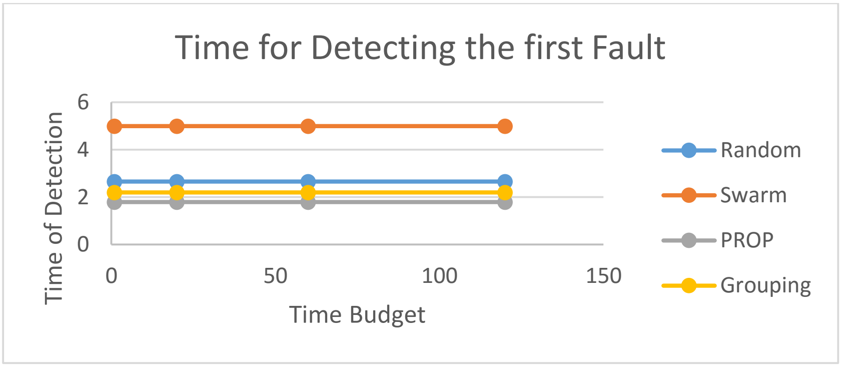

Figure 8

3. *avl.tstl :*

To test this SUT we have enabled the property check option on all algorithms. We used different time budgets to see how the algorithms would behave. The rest of default options are not changed and we are using a value of 100 for both Depth and Length and the seed will be the default which is zero. In *Figure 9* and *Figure 10* we can see that after running all algorithms multiple times and taking the best results, we can see that all algorithms are behaving the same way but covered branches number differ from one algorithm to another while increasing the time budget. Swarm starts with a small number of branches and statements then grows faster than the others and achieves more coverage. Moreover, when we check for the time it takes each algorithm to detect the fault in *Figure 11*, we can see that only swarm and grouping algorithms those were able to detect the combination lock in AVL tree. Swarm were the fastest after running each algorithm multiple times. We believe the reason that SWARM takes less time period than the others in this case because the faults cannot be detected in short periods. Therefore, SWARM were able to detect the faults faster than the others.

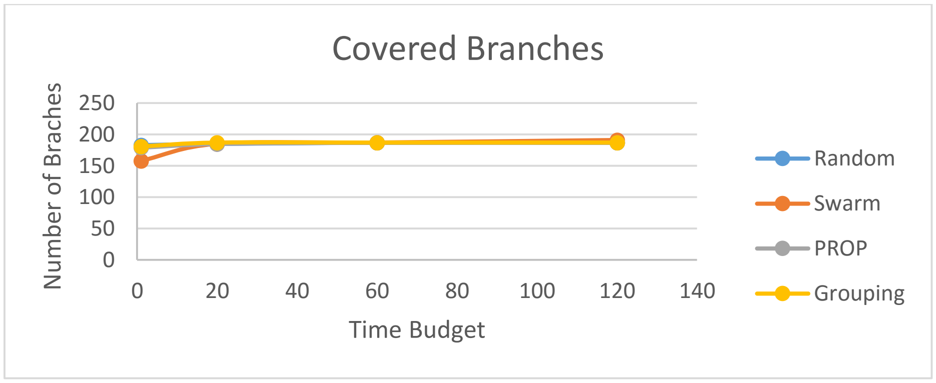

Figure 9

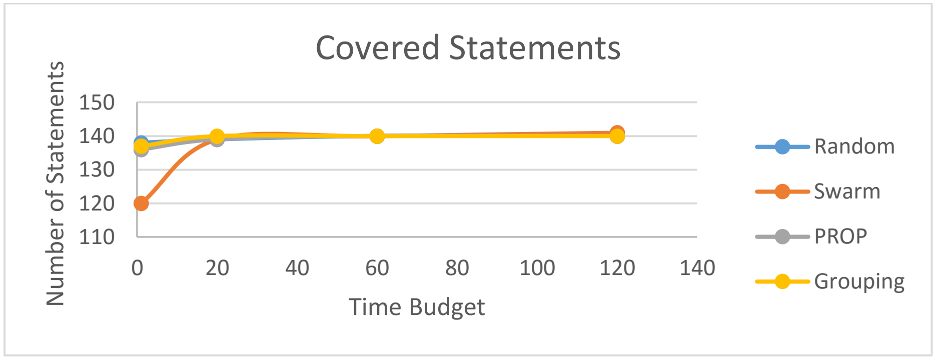

Figure 10

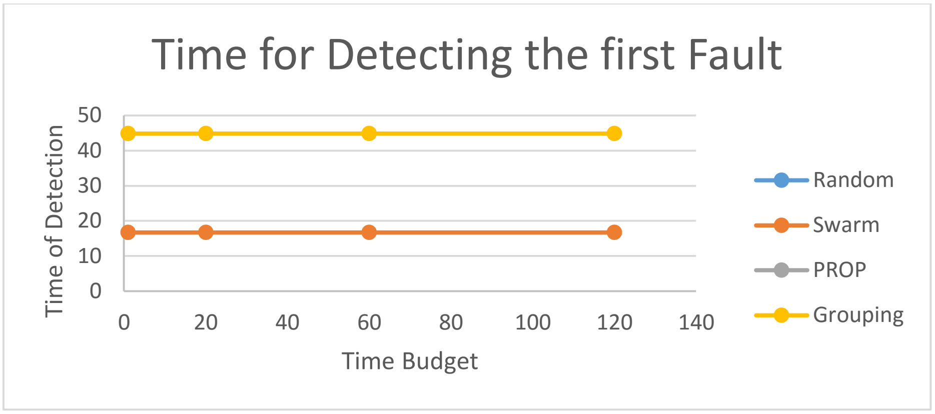

Figure 11

4. ***numpy.tstl***

To test this SUT we have enabled the property check option on all algorithms. We used different time budgets to see how the algorithms would behave. The rest of default options are kept unchanged and we are using a value of 100 for both Depth and Length and the seed will be the default value which is zero. Unfortunately, we were not able to get the coverage for this SUT, therefore coverage were not considered in this SUT testing. Also, Grouping algorithm were not able to generate test cases because of the probability budget is 0.5 by default and it needs at least 0.7 to be able to generate test cases for this SUT. That's why we don't see its curve in *Figure 13.* However, when we check for the time it takes each algorithm to detect the fault in *Figure 13,* we can see that prop algorithm has the least time period to detect the fault after running each algorithm multiple times and SWARM has the longest period. We believe the reason that SWARM takes longer period than the others is that it spends some time for initialization and making some decisions before doing extensive work that causes the algorithm unable to detect the fault earlier.

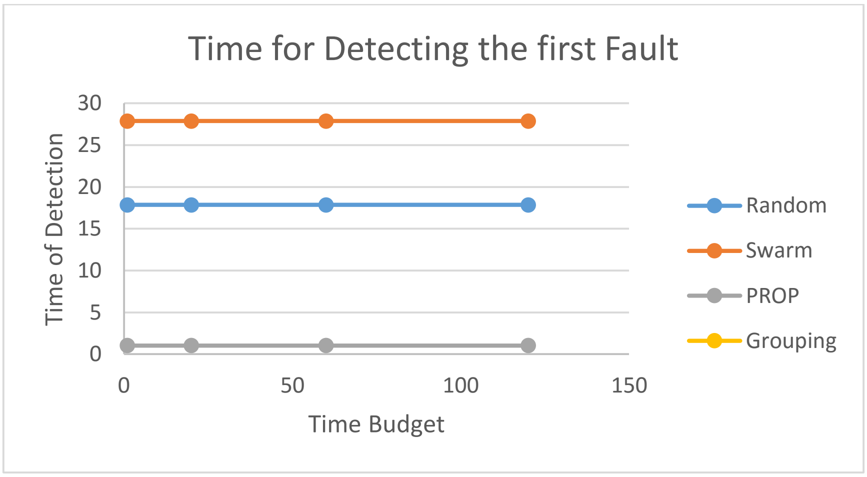

Figure 12

Comparison Conclusion

Previous figures gave us an idea on how these algorithms behave on some SUTs. Of course, four SUTs should not give us a full picture of the quality of these algorithms but the aim of this experiment is trying to get an idea on how could someone evaluate a testing generator algorithm. All of these algorithms are essentially based on Random test generator technique that uses a method to build test cases. Therefore, it is not possible to tell exactly what algorithm is faster than the others because most of the time random selection may detect faults at the early stages of initialization and sometimes it may take longer to detect the faults. Therefore, running the algorithms many times on a hundred of SUTs is required to get solid results. Moreover, the machine that runs these algorithms should have nothing else running in the back ground as that may causes an overhead and unexpected results. Moreover, unfortunately we are not aware of how to make Randomtester and Swarm algorithms keep generating test cases after detecting a fault because we are certain that Swarm algorithm will cover more branches and statements for the long time budgets. If we had that information, we will be able to give more accurate results on branches and statements coverage. Finally, property checking may detect some faults, however property checking has a huge overhead over the algorithms performance. Of course, it will cover some extra statements and branches but the overhead must be taken into account whenever we want to compare between algorithms.

Mytester Usage Instructions

The software can be used in TSTL under generator folder as follows:

```
python mytester.py -t 3 -s 2 -d 100 -l 100 -f True -c True -r True -a prop -p True -P 0.5
```

Figure 13 each parameter along with its default value:

```
  -h, --help            show this help message and exit
  -t [TIMEOUT], --timeout [TIMEOUT]
                        Timeout will be parsed in seconds - The default value
                        is 60 seconds
  -s [SEEDS], --seeds [SEEDS]
                        The number of seeds required. The default value is 0
  -d [DEPTH], --depth [DEPTH]
                        The depth of each test case. The default is 100
  -l [LENGTH], --length [LENGTH]
                        The length/Memory. The default value is 100
  -f [{True,False}], --FaultsEnabled [{True,False}]
                        Save Test Case when Failure is discovered. The default
                        value is True
  -c [{True,False}], --CoverageEnabled [{True,False}]
                        Report Code coverage. The default value is True
  -r [{True,False}], --RunningEnabled [{True,False}]
                        Check Coverage on the fly while running. The default
                        value is True
  -a [{prop,grouping}], --algorithm [{prop,grouping}]
                        There are 2 Algorithms implemented here. The first is
                        called [prop] that uses Random selections based on
                        sepcified propability. The second Algorithm is called
                        [grouping] that selects a group of actions and
                        concentrate on this group using automatically assigned
                        depths based on the length of enabled actions. The
                        default algorithm is [prop]
  -p [{True,False}], --propertyCheck [{True,False}]
                        Check All properties defined in the SUT. The default
                        Value is False
  -P [PROP], --Prop [PROP]
                        Assign the propability that can be used for both
                        algorithms. The default value is 0.5
```

Figure 13

Note: ReadMe.md file is updated under alghamha folder with all these information. You can check that using the URL

References:

[1] **“Adaptive Random Testing”,**
http://www.utdallas.edu/~ewong/SYSM-6310/03-Lecture/02-ART-paper-01.pdf

[2] **“Lightweight Automated Testing with Adaptation-Based Programming”,**
http://www.cs.cmu.edu/~agroce/issre12.pdf

[3] **“Adaptive Random Testing: the ART of Test Case Diversity”,**
https://pdfs.semanticscholar.org/11c1/87d3cd8394f4d13523b97d0a40cbdced1691.pdf

[4] **“New Strategies for Automated Random Testing”,**
http://etheses.whiterose.ac.uk/7981/1/ociamthesismain.pdf

[5] **“TSTL: The Template Scripting Testing Language”,**
https://github.com/agroce/cs569sp16/blob/master/tstl.pdf

[6] **“TSTL: The Template Scripting Testing Language”,**
https://github.com/agroce/tstl

[7] **“Python Website”,**
https://www.python.org/