

Final Report for CS569 project

Kun Chen chen4@oregonstate.edu

1 TSTL Part:

TSTL has proposed by Prof. Alex, the template scripting testing language, to help find bugs in libraries which is required to be written in Python [1]. By making use of TSTL, the Software Under Test (SUT) is needed to support the TSTL, and SUT is created by the functions in libraries for testing. Application examples has been given such as the one by using Random Test (RT) algorithm to test the AVL Tree library, the source code are available in the github[4].

Before starting a test by using TSTL, 3 files are needed. One is the libraries for Testing, its name suffix should be .py. The second file is a .tstl file, which will be compiled into python codes, and this python codes will define a class for testing tools to interface [1]. How to write a good TSTL is not so easy which requires the writer to familiar with the parameters and writing form in TSTL such as Property, Action, Pool, etc. CS562 is an excellent class for studying this part. After compiling .tstl file, a SUT file will be created. The third file is an algorithm to with the name suffix .py to find the bugs as well as report the branch coverage and statement coverage. I will introduce my algorithm in Section 2 for this project.

2 Algorithm Part:

2.1 Algorithm Description

Initially I am going to implement a novel algorithm from a paper in the test [5]. But in order to implement an algorithm in TSTL, one should be familiar with not only the API in TSTL, but also the ways how TSTL works. Prof. Alex mentioned that some of algorithm from papers may confront three kind of problems when they were be implemented: 1) Difficult to adapt to TSTL. 2) Hard to implement. 3) Unlikely to work well.

Then I am considering to start from the bfsrandom.py algorithm and make improvement of this algorithm [4]. The pure BFS algorithm in TSTL is very similar with the BFS algorithm. As it tracks every width in every layer (concrete depth), so it is easier to run out of time before going to deeper layer. So in the first step, I would like to make the change in the searching efficiency in each width. This idea is very similar to the one in Beam search algorithm [2], where k actions are executed rather than all enabled actions are required to be implemented. And in the last BFS algorithm, if size of layers is very large, then traverse each layer will cost a lot of time. Especially, If bugs exist quite deeper, then it is better to consider to go deeper in the search. So in the improved BFS algorithm, in every layer, the

algorithm knows which group of nodes has the largest number which belong to the same parent, so in this layer we this group of nodes. As a result, in all visited layers, greedily the largest sequence of states which come from the same ancestors have been visited.

Then more efforts are given to make modification of the proposed BFS algorithm. Also the improved algorithm will do the search more quickly comparing to pure BFS algorithm, but I found bugs are more likely to be found after searching for a few of depth, and comparing the random Test, it seems have lower efficiency than random test too. Then I consider taking the advantage of random test and combine it to the improved BFS.

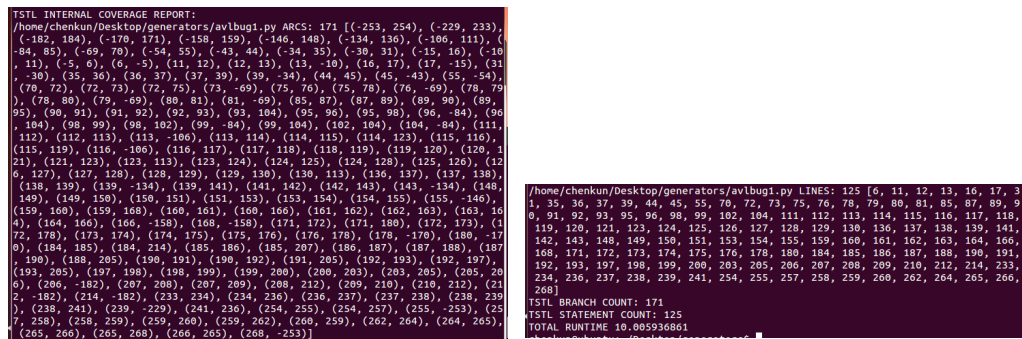
I make three modifications. Step 1: Assuming the total running time of the algorithm is t , then I use t_1 ($t_1 < t$) to run the random test to try to find some states which has bugs. Step 2. recording the depth and bug states as a tuple, and use a list to store all this tuple. Then the list is sorted according to the ascending order of recorded depth in the tuples. Step 3. When given a fixed time t , after we run RT with t_1 , we still have $(t-t_1)$ time to run BFS. So we consider another situation, in the first running loop of BFS, if we run out $(t-t_1)$ time before visited all depth, the algorithm will terminated; otherwise, if we still have time left after all depth have been visited, this time we fetch the second tuple in the given list and now set this tuple as the first start point for second loop, until we run out all of the time.

This consideration has two advantages: firstly, given a state with bugs as the root of BFS algorithm will makes it easier to find bugs, as it is assumed that bugs may have centrality [3]. Secondly, start BFS with a lower depth could have a long path than others, and it increases the possibility to find bugs. So the first tuple in the list gained in RT gives an initial start state of BFS and its depths.

2.2 Results and Bugs Report:

I used this algorithm to test the same avlTree library, called avlbug1.py, as we do in the last milestone. The tstl file called avlefficient.tstl is used here. We assume the phase 1 of RT takes $\frac{1}{4}$ times, and improved BFS takes $\frac{3}{4}$ timeout. And following command is used.

python2.7 finaltester.py 10 1 100 10 1 1 0



```

TSTL INTERNAL COVERAGE REPORT:
/home/chenkun/Desktop/generators/avlbug1.py ARCS: 171 [(-253, 254), (-229, 233),
(-182, 184), (-170, 171), (-158, 159), (-146, 148), (-134, 136), (-106, 111), (-84, 85),
(-69, 70), (-54, 55), (-43, 44), (-34, 35), (-30, 31), (-15, 16), (-10, 11), (-5, 6),
(6, -5), (11, 12), (12, 13), (13, -10), (16, 17), (17, -15), (31, -30), (35, 36),
(36, 37), (37, 39), (39, -34), (44, 45), (45, -43), (55, -54), (70, 72), (72, 73),
(72, 75), (73, -69), (75, 76), (75, 78), (76, -69), (78, 79), (78, 80), (79, -69),
(80, 81), (81, -69), (85, 87), (87, 89), (89, 90), (89, 95), (90, 91), (91, 92),
(92, 93), (93, 104), (95, 96), (95, 98), (96, -84), (98, 100), (98, 99), (98, 102),
(99, -84), (99, 104), (102, 104), (104, -84), (111, 112), (112, 113), (113, -106),
(113, 114), (114, 115), (114, 123), (115, 116), (115, 119), (116, -106), (116, 117),
(117, 118), (118, 119), (119, 120), (120, 121), (121, 123), (123, 113), (123, 124),
(124, 125), (124, 128), (125, 126), (126, 127), (127, 128), (128, 129), (129, 130),
(130, 136), (136, 137), (137, 138), (138, 139), (139, -134), (139, 141), (141, 142),
(142, 143), (143, -134), (148, 149), (149, 150), (150, 151), (151, 153), (153, 154), (154, 155),
(155, -146), (159, 160), (159, 168), (160, 161), (160, 166), (161, 162), (162, 163), (163, 164),
(164, 166), (166, -158), (168, -150), (171, 172), (171, 180), (172, 173), (173, 178),
(178, 179), (179, 184), (174, 175), (175, 176), (176, 178), (178, -170), (180, -170),
(184, 185), (184, 214), (185, 186), (185, 207), (186, 187), (187, 188), (187, 190),
(188, 205), (190, 191), (190, 192), (191, 205), (192, 193), (192, 197), (193, 205),
(197, 198), (198, 199), (199, 200), (200, 203), (203, 205), (205, 208), (208, -182),
(208, 200), (207, 209), (208, 212), (209, 210), (210, 212), (212, 214), (214, -182),
(233, 234), (234, 236), (236, 237), (237, 238), (238, 239), (238, 241), (239, -229),
(241, 236), (254, 255), (254, 257), (255, -253), (257, 258), (258, 259), (259, 260),
(259, 262), (260, 259), (262, 264), (264, 265), (265, 266), (265, 268), (266, 265),
(268, -253)]

/home/chenkun/Desktop/generators/avlbug1.py LINES: 125 [6, 11, 12, 13, 16, 17, 9,
1, 35, 36, 37, 39, 44, 45, 55, 70, 72, 73, 75, 76, 78, 79, 80, 81, 85, 87, 89, 9,
0, 91, 92, 93, 95, 96, 98, 99, 102, 104, 111, 112, 113, 114, 115, 116, 117, 118,
119, 120, 121, 123, 124, 125, 126, 127, 128, 129, 130, 136, 137, 138, 139, 141,
142, 143, 148, 149, 150, 151, 153, 154, 155, 159, 160, 161, 162, 163, 164, 166,
168, 171, 172, 173, 174, 175, 176, 178, 180, 184, 185, 186, 187, 188, 190, 191,
192, 193, 197, 198, 199, 200, 203, 205, 206, 207, 208, 209, 210, 212, 214, 233,
234, 236, 237, 238, 239, 241, 254, 255, 257, 258, 259, 260, 262, 264, 265, 266,
268]
TSTL BRANCH COUNT: 171
TSTL STATEMENT COUNT: 125
TOTAL RUNTIME 10.005936861

```

Fig.1 Results of Internal Coverage Report:

The results is shown in Fig.1. And we find 218 failure#.test is saved in the corresponding directory. And following 6 lines are the contests in one the failure#.test files.

```
self.p_avl[0] = avlbug1.AVLTree()
self.p_val[2] = 18
self.p_avl[0].insert(self.p_val[2])
self.p_val[3] = 12
self.p_avl[0].insert(self.p_val[3])
self.p_avl[0].insert(self.p_val[2])
```

Then we want to find out the efficiency of RT and improved BFS, so we use give the same parameter as in running finaltester.py, and the only parameter we changed is the percentage of RT accounting for the total running time. The results of the evaluation of counts of statement and branch are shown in the Fig.2. From Fig.2, we can observe, when percentage of RT is about 10%, there is a mutation of counts both in branch and statement.

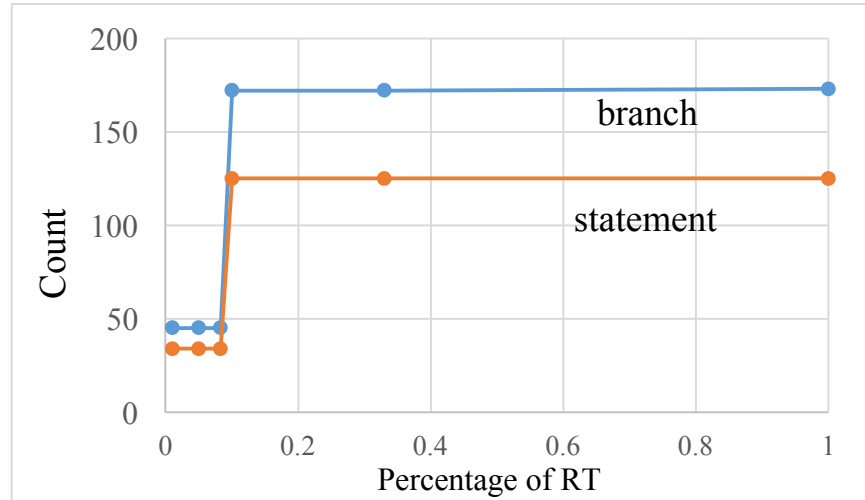


Fig.2 The evaluation of count with percentage

Then we fixed the percentage of RT as 10%, then we keep other parameters in the input as the same, and we make the depth as variable to see the evaluation of statement and branch. The results are shown in Fig,3

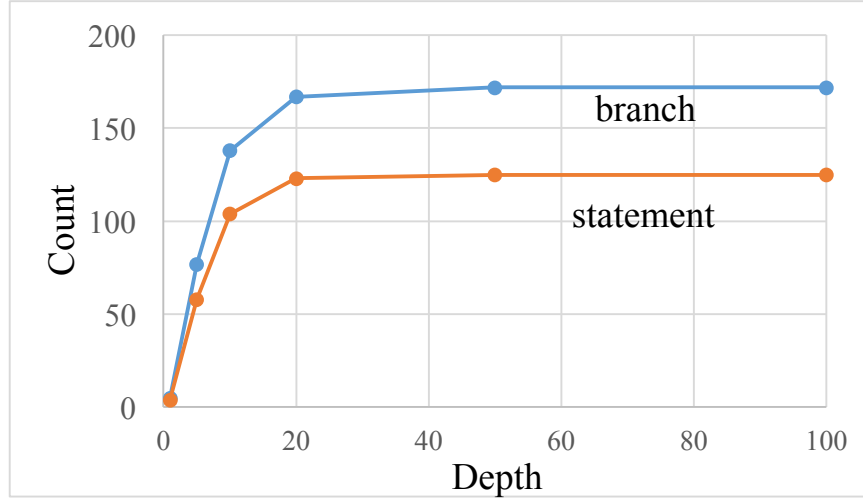


Fig.3 The evaluation of count with depth

From Fig.3, we can observe in this case, when depth is about 20, the counts of branch and statement almost arrive at their maximum values.

2.3 Comparison with other Algorithm

In order to compare the BFS (bfsrandom.py in reference [4]) and new algorithm, we give the seed=1, width=10, depth=100, and percentage of RT =10%. The results are in Table.1.

Table1. Comparison of two algorithm (timeout=10)

	BFS	New algorithm
Timeout	10	10
Arrived Depth when timeout	10	29
# of Statement	45	172
# of Branch	34	125
# of Bugs in failure.test	0	127

From table.1, we can observe the new algorithm can go deeper when given the same timeout and it seems to be more efficient to find the bugs.

2.4 The mytester.py file

By using the same algorithm, this time a more-flexible version named mytester.py which takes 9 parameters and has a good-looking configuration interface. These parameters are shown in Fig.4.

```
def parse_args():
    parser = argparse.ArgumentParser()
    parser.add_argument('-d', '--depth', type=int, default=100, help='Maximum search depth (100 default).')
    parser.add_argument('-w', '--width', type=int, default=10, help='Maximum memory/BFS queue/other parameter for search width (10 default).')
    parser.add_argument('-t', '--timeout', type=int, default=30, help='Timeout in seconds (30 default).')
    parser.add_argument('-s', '--seed', type=int, default=None, help='Random seed (default = None).')
    parser.add_argument('-b', '--reistribute', type=int, default=4, help='Redistribute the time for running Radom Test (default = 4).')
    parser.add_argument('-f', '--faults', action='store_true', help='Save the failure cases.')
    parser.add_argument('-g', '--reducing', action='store_true', help='reduce — Do not report full failing test.')
    parser.add_argument('-r', '--running', action='store_true', help='running info on branch coverage')
    parser.add_argument('-c', '--coverage', action='store_true', help='Give a final report.')
    parsed_args = parser.parse_args(sys.argv[1:])
    return (parsed_args, parser)
```

Fig.4 Parameters in parse_args function

There are total 4 functions in mytester.py, they are parse_args(), make_config (), branchFun(), main (). And in main () functions, there two-phase calculations, one is RT and another is improved BFS, which is as the same as the ones in finaltester.py.

To run the mytester.py, we can use following command:

```
python2.7 mytester.py -t 10 -s 1 -d 100 -w 10 -c
```

3 Future Improvement:

In the future, as the mutation and crossover method in GA have a high coverage, GA method can be added to the existent algorithm to increase the coverage and make it more efficient to find bugs.

4. Conclusions

In this project, a new algorithm by the combination of advantage of BFS and RT are proposed. After testing of this algorithm, following conclusions can be obtained:

- 1) when the percentage of RT is about 10%, the branch and statement coverage can arrive the maximum.
- 2) when the depth of BFS is about 20, the branch and statement coverage can arrive the maximum.
- 3) In the same timeout, comparing to the BFS algorithm in the reference [4], the proposed algorithm can go deeper depth and have more efficiency to find bugs.

Reference:

- 1) Alex Groce, Jervis Pinto, Pooria Azimi, Pranjal Mittal. TSTL. A Language and Tool for Testing (Demo). ISSTA'15, 2015
- 2) Alex Groce, Jervis Pinto. A Little Language for Testing. NFM'15, 2015
- 3) CHEN Tsong-Yueh, KUO Fei-Ching, SUN Chang-Ai. Impact of the Compactness of Failure Regions on the Performance of Adaptive Random Testing. Journal of Software, Vol.17, No.12, 2006, pp.2438–2449.
- 4) <https://github.com/agroce/tstl>
- 5) CHEN Tsong-Yueh, KUO Fei-Ching, SUN Chang-Ai. Impact of the Compactness of Failure Regions on the Performance of Adaptive Random Testing. Journal of Software, Vol.17, No.12, 2006, pp.2438–2449.