Avoid known bugs generated by TSTL Random Tester

Swathi Sri Vishnu Priya Rayala (932-696-872)

Background

Random Testing (RT) is a black-box software testing technique where programs are tested by generating random, independent inputs. Results of the output are compared against software specifications to verify that the test output is pass or fail. Thus, RT is used to assess software reliability as well as to detect software failures by simply selecting test cases in a random manner from the whole input domain. However, there is a lot of scope in improving the Random Testing to detect the software failures depending on failure-causing inputs.

Related work

The existing Random Tester of TSTL, tries to identify all the possible bugs for a given timeout period. But, all the identified bugs are not necessarily the unknown bugs. Some bugs can be repetitive. So, my idea is to modify the existing random tester of TSTL in such a way that bugs identified should be distinct. Also, I'm curious to see what the branch coverage and code coverage would be. There are few algorithms to implement this as mentioned in [1] and [2]. One of the approach is to generalize each bug as it is found, to a 'bug pattern', and then adapt test case generation so that test cases matching an existing bug pattern are never generated again randomly [2]. The main contributions of [2] are:

• A fully automatic method to avoid provoking already known bugs at test case generation time, and to avoid bug slippage when failed tests are minimized.

• Experimental results showing that this method can, in some cases, reduce the number of tests needed to find a set of bugs quite dramatically, and even find new bugs in well-studied software.

Algorithm and Implementation

So, I want to implement the same kind of algorithm in TSTL. I noticed that **TSTL doesn't have equivalence check for the variable names**. For example (as shown below), I have 2 test cases T1 and T2, which are identical but differ in variable names only.

T1

val0 = 18 *# STEP 0*

avl1 = avl.AVLTree() *# STEP 1*

avl1.delete(val0) *# STEP 2*

T2

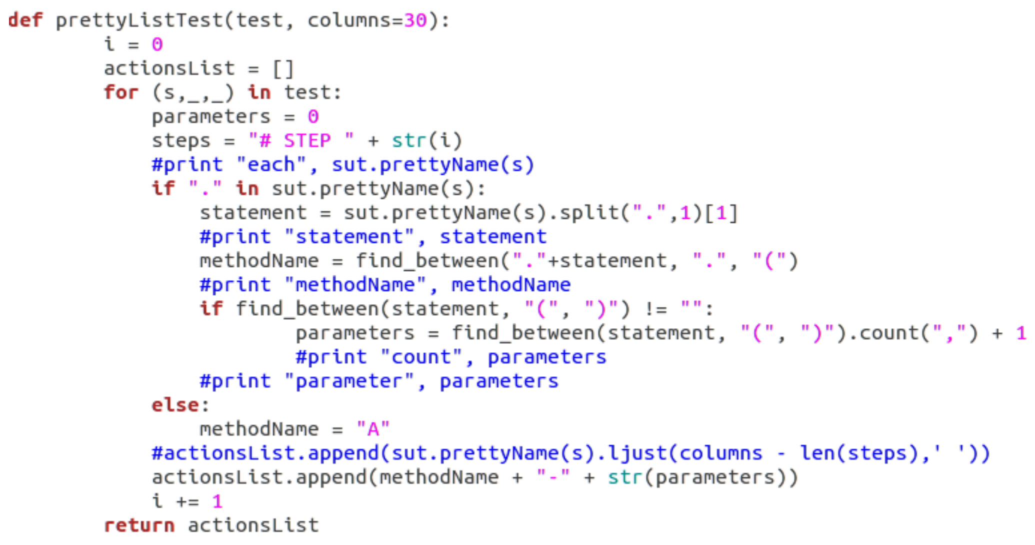
val0 = 18 *# STEP 0*

avl0 = avl.AVLTree() *# STEP 1*

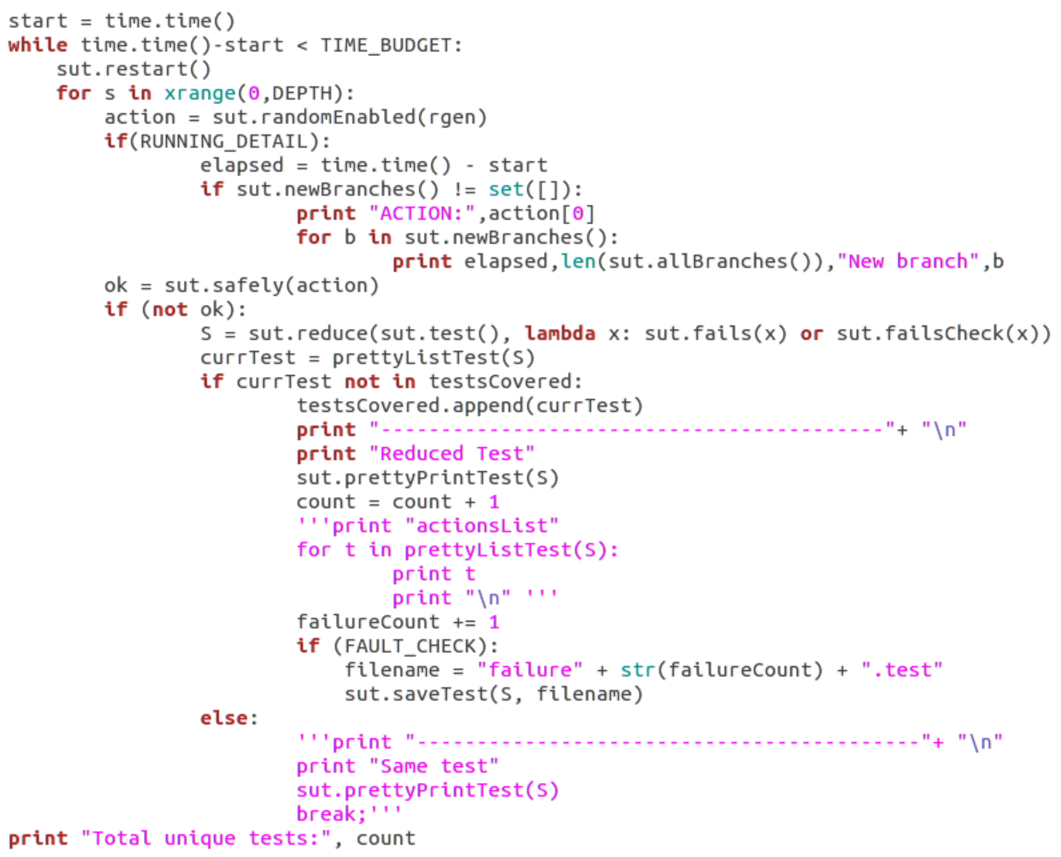
avl0.delete(val0) *# STEP 2*

According to the algorithm defined [2] to be implemented, both these tests are same. But TSTL shows these as 2 different tests as they differ in the variable names. Also for now, there is no way to capture the sequence of operations in a particular test case in TSTL. So, for our algorithm to be successfully implemented, it is important to save the sequence of operations that caused the failure. Thus, further test cases with same values can be ignored. This would help us in giving more time to test for new sequence of operations which causes a failure.

A screenshot of the implementation where the sequences are captured in the list actionsList is shown below –



```
def prettyListTest(test, columns=30):
        i = 0
        actionsList = []
        for (s,_,_) in test:
            parameters = 0
            steps = "# STEP " + str(i)
            #print "each", sut.prettyName(s)
            if "." in sut.prettyName(s):
                statement = sut.prettyName(s).split(".",1)[1]
                #print "statement", statement
                methodName = find_between("."+statement, ".", "(")
                #print "methodName", methodName
                if find_between(statement, "(", ")") != "":
                        parameters = find_between(statement, "(", ")").count(",") + 1
                        #print "count", parameters
                #print "parameter", parameters
            else:
                methodName = "A"
            #actionsList.append(sut.prettyName(s).ljust(columns - len(steps),' '))
            actionsList.append(methodName + "-" + str(parameters))
            i += 1
        return actionsList
```



```
start = time.time()
while time.time()-start < TIME_BUDGET:
    sut.restart()
    for s in xrange(0,DEPTH):
        action = sut.randomEnabled(rgen)
        if(RUNNING_DETAIL):
                elapsed = time.time() - start
                if sut.newBranches() != set([]):
                        print "ACTION:",action[0]
                        for b in sut.newBranches():
                                print elapsed,len(sut.allBranches()),"New branch",b
        ok = sut.safely(action)
        if (not ok):
                S = sut.reduce(sut.test(), lambda x: sut.fails(x) or sut.failsCheck(x))
                currTest = prettyListTest(S)
                if currTest not in testsCovered:
                        testsCovered.append(currTest)
                        print "------------------------------------------"+ "\n"
                        print "Reduced Test"
                        sut.prettyPrintTest(S)
                        count = count + 1
                        '''print "actionsList"
                        for t in prettyListTest(S):
                                print t
                                print "\n" '''
                        failureCount += 1
                        if (FAULT_CHECK):
                            filename = "failure" + str(failureCount) + ".test"
                            sut.saveTest(S, filename)
                else:
                        '''print "------------------------------------------"+ "\n"
                        print "Same test"
                        sut.prettyPrintTest(S)
                        break;'''
print "Total unique tests:", count
```

Whenever a new bug is encountered, the sequence of its operations is saved in a list. So for the next time when a new test is generated, its sequence of operations are verified with the already existing list of test sequences captured. In this way, only unique tests are generated. I'm successful in the execution of this. For example (as shown below), I have 2 test cases T1 and T2, which are identical but differ in variable names only. Now, my implementation identifies them as same test and ignores generating such tests if one test of that kind is already saved.

T1

val0 = 18 # STEP 0

avl1 = avl.AVLTree() # STEP 1

avl1.delete(val0) # STEP 2

T2

val0 = 18 # STEP 0

avl0 = avl.AVLTree() # STEP 1

avl0.delete(val0) # STEP 2

After implementing the defined algorithm, I further added few parameters as arguments to better analyze the test cases and those arguments include –

Timeout – time taken by the tester for testing

Seed – Seed for python Random.random object used for random number generation in code

Depth – maximum length of a test generated by the algorithm

Width – maximum memory/BFS queue/other parameter that is basically a search width

Faults – either 0 or 1 depending on whether your tester should check for faults in the SUT.

Coverage – either 0 or 1 depending on whether a final coverage report should be produced, using TSTL's internalReport() function.

Running – either 0 or 1 depending on whether running information on branch coverage should be produced.

Algorithm – either variable or sequence depending on whether unique variables or unique sequence of operations respectively are to be considered while generating the test cases.

When I ran mytester.py with argument as variable and sequence for 300 seconds, below are the counts for bugs found and coverage improvement –

Unique variable test case results

27 bugs found (not necessarily unique)

Branch count is 225

Statement count is 164

Unique sequence test case results

53 unique bugs found

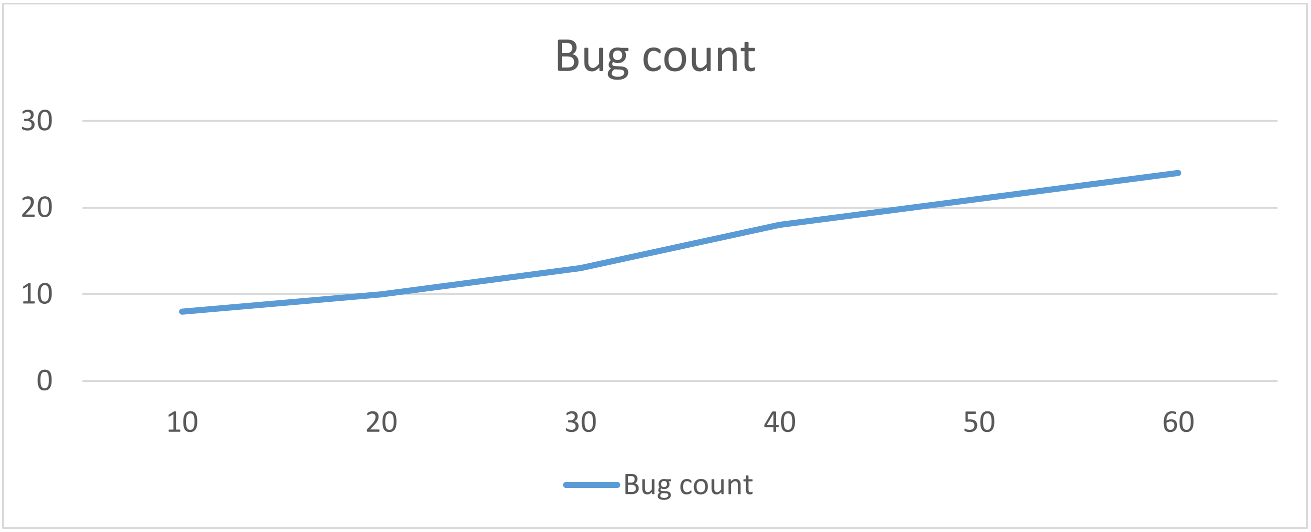
Branch count is 231

Statement count is 166

Thus there is a significant improvement with the implement of new algorithm in TSTL.

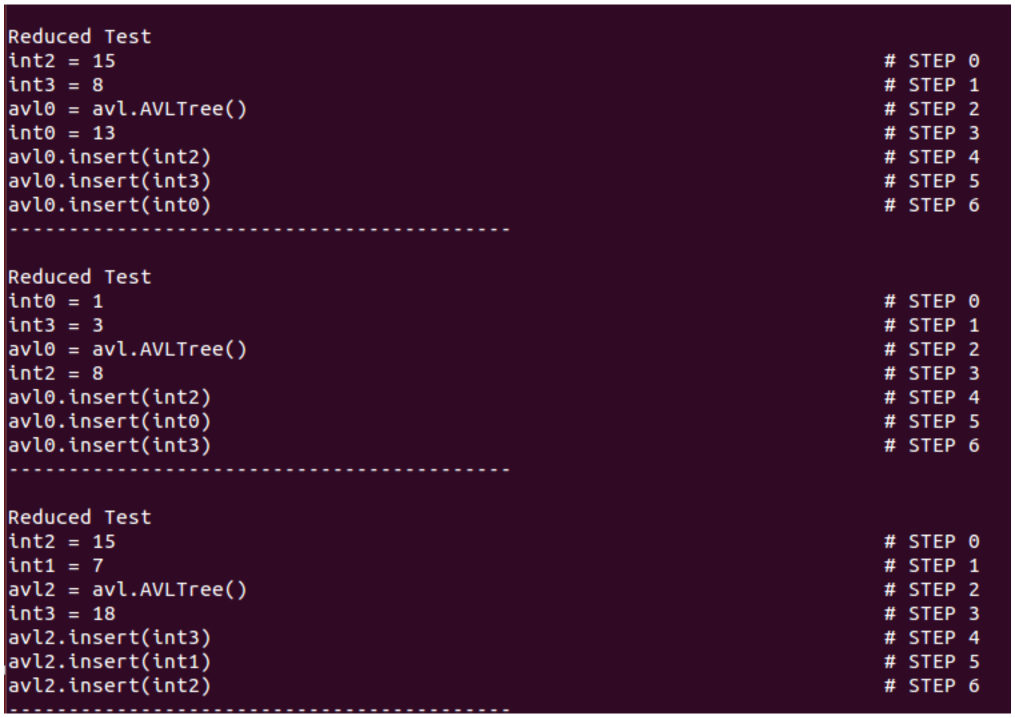
Analysis

I tested my algorithm for different time periods and identified the number of unique bugs that it can find. I can see that the bug count always increases when the time of running the tester increases. Below is the screenshot of the graph –



Samples of my execution

Screenshot of failed test cases when the tester is run with argument *algorithm* = *variable*. The sequence of operations are same but the variable names are different.

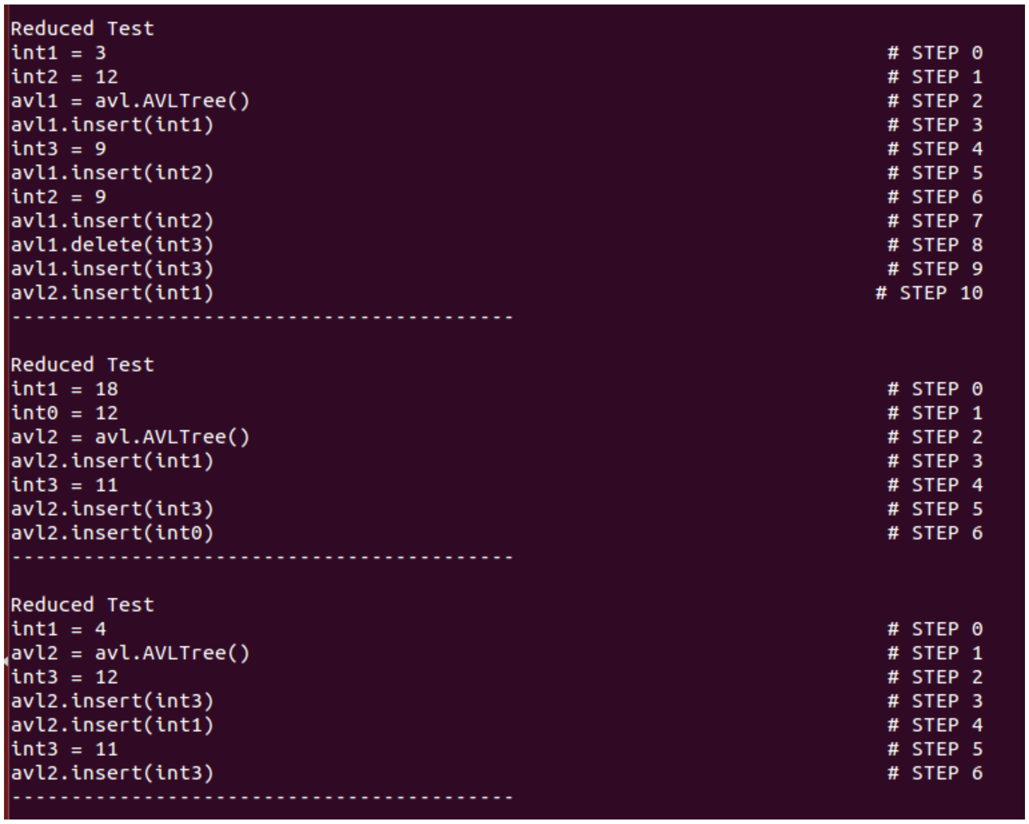


```
Reduced Test
int2 = 15                                                     # STEP 0
int3 = 8                                                      # STEP 1
avl0 = avl.AVLTree()                                          # STEP 2
int0 = 13                                                     # STEP 3
avl0.insert(int2)                                             # STEP 4
avl0.insert(int3)                                             # STEP 5
avl0.insert(int0)                                             # STEP 6
-----------------------------------------

Reduced Test
int0 = 1                                                      # STEP 0
int3 = 3                                                      # STEP 1
avl0 = avl.AVLTree()                                          # STEP 2
int2 = 8                                                      # STEP 3
avl0.insert(int2)                                             # STEP 4
avl0.insert(int0)                                             # STEP 5
avl0.insert(int3)                                             # STEP 6
-----------------------------------------

Reduced Test
int2 = 15                                                     # STEP 0
int1 = 7                                                      # STEP 1
avl2 = avl.AVLTree()                                          # STEP 2
int3 = 18                                                     # STEP 3
avl2.insert(int3)                                             # STEP 4
avl2.insert(int1)                                             # STEP 5
avl2.insert(int2)                                             # STEP 6
-----------------------------------------
```

Screenshot of failed test cases when the tester is run with argument *algorithm* = *sequence*. The sequence of operations are different and hence the test cases are also different.

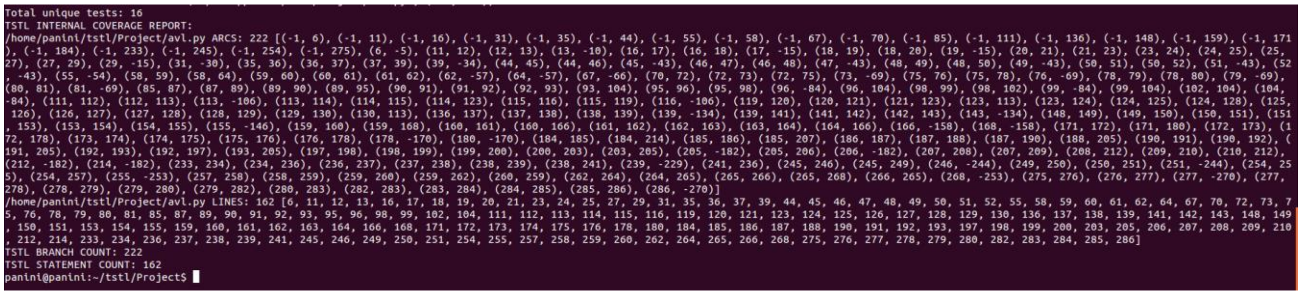


```
Reduced Test
int1 = 3                                                                    # STEP 0
int2 = 12                                                                   # STEP 1
avl1 = avl.AVLTree()                                                        # STEP 2
avl1.insert(int1)                                                           # STEP 3
int3 = 9                                                                    # STEP 4
avl1.insert(int2)                                                           # STEP 5
int2 = 9                                                                    # STEP 6
avl1.insert(int2)                                                           # STEP 7
avl1.delete(int3)                                                           # STEP 8
avl1.insert(int3)                                                           # STEP 9
avl2.insert(int1)                                                          # STEP 10
------------------------------------------

Reduced Test
int1 = 18                                                                   # STEP 0
int0 = 12                                                                   # STEP 1
avl2 = avl.AVLTree()                                                        # STEP 2
avl2.insert(int1)                                                           # STEP 3
int3 = 11                                                                   # STEP 4
avl2.insert(int3)                                                           # STEP 5
avl2.insert(int0)                                                           # STEP 6
------------------------------------------

Reduced Test
int1 = 4                                                                    # STEP 0
avl2 = avl.AVLTree()                                                        # STEP 1
int3 = 12                                                                   # STEP 2
avl2.insert(int3)                                                           # STEP 3
avl2.insert(int1)                                                           # STEP 4
int3 = 11                                                                   # STEP 5
avl2.insert(int3)                                                           # STEP 6
------------------------------------------
```

Screenshot of internal report that shows the branch count, statement count and number of unique tests generated.



```
Total unique tests: 16
TSTL INTERNAL COVERAGE REPORT:
/home/panini/tstl/Project/avl.py ARCS: 222 [(-1, 6), (-1, 11), (-1, 16), (-1, 31), (-1, 35), (-1, 44), (-1, 55), (-1, 58), (-1, 67), (-1, 70), (-1, 85), (-1, 111), (-1, 136), (-1, 148), (-1, 159), (-1, 171
), (-1, 184), (-1, 233), (-1, 245), (-1, 254), (-1, 275), (6, -5), (11, 12), (12, 13), (13, -10), (16, 17), (16, 18), (17, -15), (18, 19), (18, 20), (19, -15), (20, 21), (21, 23), (23, 24), (24, 25), (25,
27), (27, 29), (29, -15), (31, -30), (35, 36), (36, 37), (37, 39), (39, -34), (44, 45), (44, 46), (45, -43), (46, 47), (46, 48), (47, -43), (48, 49), (48, 50), (49, -43), (50, 51), (50, 52), (51, -43), (52
, -43), (55, -54), (58, 59), (58, 64), (59, 60), (60, 61), (61, 62), (62, -57), (64, -57), (67, -66), (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), (96, 104), (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), (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, 113), (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, -158), (171, 172), (171, 180), (172, 173), (1
72, 178), (173, 174), (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, -182), (205, 206), (206, -182), (207, 208), (207, 209), (208, 212), (209, 210), (210, 212),
(212, -182), (214, -182), (233, 234), (234, 236), (236, 237), (237, 238), (238, 239), (238, 241), (239, -229), (241, 236), (245, 246), (245, 249), (246, -244), (249, 250), (250, 251), (251, -244), (254, 25
5), (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), (275, 276), (276, 277), (277, -270), (277,
278), (278, 279), (279, 280), (279, 282), (280, 283), (282, 283), (283, 284), (284, 285), (285, 286), (286, -270)]
/home/panini/tstl/Project/avl.py LINES: 162 [6, 11, 12, 13, 16, 17, 18, 19, 20, 21, 23, 24, 25, 27, 29, 31, 35, 36, 37, 39, 44, 45, 46, 47, 48, 49, 50, 51, 52, 55, 58, 59, 60, 61, 62, 64, 67, 70, 72, 73, 7
5, 76, 78, 79, 80, 81, 85, 87, 89, 90, 91, 92, 93, 95, 96, 98, 99, 102, 104, 111, 112, 113, 114, 115, 116, 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, 245, 246, 249, 250, 251, 254, 255, 257, 258, 259, 260, 262, 264, 265, 266, 268, 275, 276, 277, 278, 279, 280, 282, 283, 284, 285, 286]
TSTL BRANCH COUNT: 222
TSTL STATEMENT COUNT: 162
panini@panini:~/tstl/Project$ 
```

References

[1] Algorithms to identify failure pattern, Bhuwan Krishna Som, Poudel

Url: http://www.diva-portal.org/smash/get/diva2:655645/FULLTEXT01.pdf

[2] Find more bugs with QuickCheck, John Hughes, Ulf Norell, Nicholas Smallbone

Url: http://publications.lib.chalmers.se/records/fulltext/232554/local_232554.pdf

[3] TSTL: A Language and Tool for Testing, Alex Groce, Jervis Pinto, Pooria Azimi, Pranjal Mittal

Url: http://www.cs.cmu.edu/~agroce/issta15.pdf