Final Project Report

Name: Xuan He

Introduction

It is widely recognized that software testing is an essential component of any successful software development process. A software test consists of an input that executes the program and a definition of the expected outcome. There are many testing methods to find the bug of the programs.

Unit testing is a software testing method by which individual units of source code, sets of one or more computer program modules together with associated control data, usage procedures, and operating procedures, are tested to determine whether they are fit for use. Intuitively, one can view a unit as the smallest testable part of an application.

In procedural programming, a unit could be an entire module, but it is more commonly an individual function or procedure. In object-oriented programming, a unit is often an entire interface, such as a class, but could be an individual method.

Unit testing can find problems early in the development cycle. This includes both bugs in the programmer's implementation and flaws or missing parts of the specification for the unit. The process of writing a thorough set of tests forces the author to think through inputs, outputs, and error conditions, and thus more crisply define the unit's desired behavior. The cost of finding a bug before coding begins or when the code is first written is considerably lower than the cost of detecting, identifying, and correcting the bug later; bugs may also cause problems for the end-users of the software.

The Whole Test Suite Generation Algorithm

Automatic unit test generation aims to support developers by alle- viating the burden of test writing. Different techniques have been proposed over the years, each with distinct limitations. In order to overcome these limitations, the EVOSUITE unit test generator combines two of the most popular techniques for test case generation: Search-Based Software Testing and Dynamic Symbolic Execution (DSE). A novel integration of DSE as a step of local improvement in a genetic algorithm results in an adaptive approach, such that the best test generation technique for the problem at hand is favoured, resulting in overall higher code coverage.

Search-Based Software Testing describes the use of efficient search algorithms for the task of generating test cases. Genetic algorithms are one of the most commonly applied search algorithms. Genetic algorithms mimic the natural process of evolution: An initial population of usually randomly produced candidate solutions is evolved using search-operators that resemble natural processes. In the context of whole test suite generation, this population is made of different test suites. After the randomly produced candidates are ready, the fittest parents are selected for reproduction, mimicing the natural phenomena of survival of the fittest. Crossover combines different parents to generate offspring. The offspring is then mutated with certain probabilities. Once reproduction is finished, a new generation is ready to be used as new parents for selection. This process continues until either the target coverage criterion has been met (e.g., all branches are covered) or the search budget has been exhausted (e.g., timeout or maximum number of generations).

My Implement

In the finaltester.py, I improve my algorithm by simulating Genetic algorithms.

In first phase, the program uses the function “sut.randomEnabled(rgen)” to generate the random actions, and checks whether the action is correct by the function “sut.safely(action)”. Then it calls the function “collectCoverage()” to count and record the current statement in the array “coverageCount” from the set “sut.currStatements()”. If the set “sut.newStatements()” is not empty, it puts the current test to the fullpool which will be used in phase 2. When the bug is found, firstly it gets the reducing steps of the test by using the function “sut.reduce(sut.test(), sut.fails, True, True)” as R, then call the function “sut.saveTest(R,filename)” to record the faults case to the file. After that, it terminates the test. If Running is set, it will call the function “sut.newBranches()” and show the branches information. The running time of the first phase is 1/5 of the limted time “Timeout”.

In second phase, the program build an activePool that stores the statements whose coverage is less than the mean coverage among the fullPool. After that, it first define a array “fittest” to store the current action lists in activePool. When the action lists in activePool are less than that in fullPool, it will find two action lists randomly in activePool by calling the function “findParent()”. Then it will call the function “switchRandom()” to switch some actions randomly between two action lists. For each new action lists, it will execute the test case, and count and record the current statement

in the array “coverageCount”. If it generates newStatement, the test case will be added to the fullPool. At last, it will put the new action list to the array “fittest”. If the size of the array “fittest” is changed, It will rebuild the activePool. After that, the program uses the function “sut.randomEnabled(rgen)” to generate the random actions, and checks whether the action is correct by the function “sut.safely(action)”. After that it will execute the function “sut.replay(rgen.choice(activePool)[0])” randomly, which will cover one of the statement in the activePool. Then, it will continue do the test as same as the first phase.

At last, if Coverage is set, it will call the function “sut.internalReport()” to show the coverage information.

Relata Work

In order to improve my algorithm, I read a lot of papers about TSTL and The Whole Test Suite Generation, such as “A Little Language for Testing” and “On The Effectiveness of Whole Test Suite Generation”, these papers give me a lot of help to implement my algorithm. I also do a lot of experiments to test my algorithm.

Evaluate Algorithm

I use the AVLTree to evaluate my algorithm. The tester1.py just implements a basic idea of random testing. It just selects the actions randomly and then takes testing. Bellow is the testing result of the tester1.py :

```
0.0290558338165 130 New branch (u'/Users/hexuan/Documents/cs569/2/avl.py', (77, -70))
0.0290558338165 130 New branch (u'/Users/hexuan/Documents/cs569/2/avl.py', (76, 77))
ACTION: self.p_avl[1].insert(self.p_val[1])
0.0332789421082 131 New branch (u'/Users/hexuan/Documents/cs569/2/avl.py', (128, 132))
TSTL INTERNAL COVERAGE REPORT:
/Users/hexuan/Documents/cs569/2/avl.py ARCS: 131 [(-1, 6), (-1, 11), (-1, 16), (-1, 31), (-1, 35), (-1, 45), (-1, 56), (-1, 71), (-1, 86), (-1, 124), (-1, 149
), (-1, 161), (-1, 172), (-1, 184), (-1, 197), (-1, 267), (6, -5), (11, 12), (12, 13), (13, -10), (16, 17), (17, -15), (31, -30), (35, 36), (36, 37), (37, 38)
, (38, 40), (40, -34), (45, 46), (46, -44), (56, -55), (71, 73), (73, 74), (73, 76), (74, -70), (76, 77), (76, 79), (77, -70), (79, 80), (79, 81), (80, -70),
(81, 82), (82, -70), (86, 88), (88, 90), (90, 92), (92, 94), (94, 98), (98, 100), (100, 102), (102, 103), (102, 108), (103, 104), (104, 105), (105, 106), (106
, 117), (108, 109), (108, 111), (109, 117), (111, 112), (111, 115), (112, 117), (115, 117), (117, -85), (124, 125), (125, 126), (126, -119), (126, 127), (127,
 128), (128, 129), (128, 132), (129, 130), (130, 131), (131, 132), (132, 133), (133, 134), (134, 136), (136, 126), (149, 150), (150, 151), (151, 152), (152, 1
54), (154, 155), (155, 156), (156, -147), (161, 162), (162, 163), (163, 164), (164, 166), (166, 167), (167, 168), (168, -159), (172, 173), (172, 181), (173, 1
74), (173, 179), (174, 175), (175, 176), (176, 177), (177, 179), (179, -171), (181, -171), (184, 185), (184, 193), (185, 186), (185, 191), (186, 187), (187, 1
88), (188, 189), (189, 191), (191, -183), (193, -183), (197, 198), (197, 227), (198, 199), (198, 220), (199, 200), (200, 201), (201, 217), (217, 218), (218, 2
19), (219, -195), (220, 221), (220, 222), (221, 225), (222, 223), (223, 225), (225, -195), (227, -195), (267, 268), (268, -266)]
/Users/hexuan/Documents/cs569/2/avl.py LINES: 100 [6, 11, 12, 13, 16, 17, 31, 35, 36, 37, 38, 40, 45, 46, 56, 71, 73, 74, 76, 77, 79, 80, 81, 82, 86, 88, 90,
92, 94, 98, 100, 102, 103, 104, 105, 106, 108, 109, 111, 112, 115, 117, 124, 125, 126, 127, 128, 129, 130, 131, 132, 133, 134, 136, 149, 150, 151, 152, 154, 1
55, 156, 161, 162, 163, 164, 166, 167, 168, 172, 173, 174, 175, 176, 177, 179, 181, 184, 185, 186, 187, 188, 189, 191, 193, 197, 198, 199, 200, 201, 217, 218,
 219, 220, 221, 222, 223, 225, 227, 267, 268]
TSTL BRANCH COUNT: 131
TSTL STATEMENT COUNT: 100
0 FAILED
ACTIONS  :  100
RUN TIME :  0.0365569591522
Xuans-MacBook-Pro:2 hexuan$
```

The finaltester.py implement the random testing by using Genetic algorithms. Below is the testing result of the finaltester.py:

```
5.03234195709 183 New branch (u'/Users/hexuan/Documents/cs569/2/avl.py', (90, 91))
5.03234195709 183 New branch (u'/Users/hexuan/Documents/cs569/2/avl.py', (91, 100))
ACTION: self.p_avl[0].delete(self.p_val[2])
8.3482670784 185 New branch (u'/Users/hexuan/Documents/cs569/2/avl.py', (251, 254))
8.3482670784 185 New branch (u'/Users/hexuan/Documents/cs569/2/avl.py', (254, 249))
TSTL INTERNAL COVERAGE REPORT:
/Users/hexuan/Documents/cs569/2/avl.py ARCS: 185 [(-1, 6), (-1, 11), (-1, 16), (-1, 31), (-1, 35), (-1, 45), (-1, 56), (-1, 71), (-1, 86), (-1, 124), (-1, 149
), (-1, 161), (-1, 172), (-1, 184), (-1, 197), (-1, 246), (-1, 258), (-1, 267), (6, -5), (11, 12), (12, 13), (13, -10), (16, 17), (17, -15), (31, -30), (35, 3
6), (36, 37), (37, 38), (38, 40), (40, -34), (45, 46), (46, -44), (56, -55), (71, 73), (73, 74), (73, 76), (74, -70), (76, 77), (76, 79), (77, -70), (79, 80),
 (79, 81), (80, -70), (81, 82), (82, -70), (86, 88), (88, 89), (88, 90), (89, 100), (90, 91), (90, 92), (91, 100), (92, 94), (94, 98), (98, 100), (100, 102),
(102, 103), (102, 108), (103, 104), (104, 105), (105, 106), (106, 117), (108, 109), (108, 111), (109, 117), (111, 112), (111, 115), (112, 117), (115, 117), (1
17, -85), (124, 125), (125, 126), (126, -119), (126, 127), (127, 128), (127, 136), (128, 129), (128, 132), (129, 130), (130, 131), (131, 132), (132, 133), (13
3, 134), (134, 136), (136, 126), (136, 137), (137, 138), (137, 141), (138, 139), (139, 140), (140, 141), (141, 142), (142, 143), (143, 126), (149, 150), (150,
 151), (151, 152), (152, 154), (154, 155), (155, 156), (156, -147), (161, 162), (162, 163), (163, 164), (164, 166), (166, 167), (167, 168), (168, -159), (172,
 173), (172, 181), (173, 174), (173, 179), (174, 175), (175, 176), (176, 177), (177, 179), (179, -171), (181, -171), (184, 185), (184, 193), (185, 186), (185,
 191), (186, 187), (187, 188), (188, 189), (189, 191), (191, -183), (193, -183), (197, 198), (197, 227), (198, 199), (198, 220), (199, 200), (200, 201), (200,
 203), (201, 217), (203, 204), (203, 205), (204, 217), (205, 206), (205, 210), (206, 217), (210, 211), (211, 212), (212, 213), (213, 216), (216, 217), (217, 2
18), (218, 219), (219, -195), (220, 221), (220, 222), (221, 225), (222, 223), (223, 225), (225, -195), (227, -195), (246, 247), (247, 249), (249, 250), (250,
251), (251, 252), (251, 254), (252, -242), (254, 249), (258, 259), (258, 262), (259, -257), (262, 263), (263, 264), (264, -257), (267, 268), (267, 270), (268,
 -266), (270, 271), (271, 272), (272, 273), (272, 275), (273, 272), (275, 277), (277, 278), (278, 279), (278, 281), (279, 278), (281, -266)]
/Users/hexuan/Documents/cs569/2/avl.py LINES: 139 [6, 11, 12, 13, 16, 17, 31, 35, 36, 37, 38, 40, 45, 46, 56, 71, 73, 74, 76, 77, 79, 80, 81, 82, 86, 88, 89,
90, 91, 92, 94, 98, 100, 102, 103, 104, 105, 106, 108, 109, 111, 112, 115, 117, 124, 125, 126, 127, 128, 129, 130, 131, 132, 133, 134, 136, 137, 138, 139, 140
, 141, 142, 143, 149, 150, 151, 152, 154, 155, 156, 161, 162, 163, 164, 166, 167, 168, 172, 173, 174, 175, 176, 177, 179, 181, 184, 185, 186, 187, 188, 189, 1
91, 193, 197, 198, 199, 200, 201, 203, 204, 205, 206, 210, 211, 212, 213, 216, 217, 218, 219, 220, 221, 222, 223, 225, 227, 246, 247, 249, 250, 251, 252, 254,
 258, 259, 262, 263, 264, 267, 268, 270, 271, 272, 273, 275, 277, 278, 279, 281]
TSTL BRANCH COUNT: 185
TSTL STATEMENT COUNT: 139
FAILED   :  0
ACTIONS  :  535
RUN TIME :  10.0523519516
Xuans-MacBook-Pro:2 hexuan$
```

I also test the AVLTree using the randomtester.py in TSTL, but the result is not as well as my expectation. I am not sure why it is happen, maybe the argument I use is not very well. Below is the testing result of randomtester.py:

```
Xuans-MacBook-Pro:2 hexuan$ python randombeam.py -t40 -d1000
Random variation of beam search using config=Config(full=False, multiple=False, ignoreprops=False, failedLogging=None, maxtests=-1, coverfile='coverage.out',
keep=False, running=False, width=10, depth=1000, seed=None, logging=None, timeout=40, output=None, uncaught=False, html=None, nocover=False)
STOPPING TEST DUE TO TIMEOUT, TERMINATED AT LENGTH 1
STOPPING TESTING DUE TO TIMEOUT
4.77611940299 PERCENT COVERED
11 EXECUTED
17 BRANCHES COVERED
12 STATEMENTS COVERED
Xuans-MacBook-Pro:2 hexuan$
```

Therefore, we can see that finaltester.py work well for the random testing, and I think I do an improvement for the random testing algorithm.

Reference

1. Gordon Fraser and Andrea Arcuri. 2011. EvoSuite: automatic test suite generation for object-oriented software. In Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering (ESEC/FSE '11). ACM, New York, NY, USA, 416-419. 2. Gordon Fraser and Andrea Arcuri. 2013. Whole Test Suite Generation. IEEE Trans. Softw. Eng. 39, 2 (February 2013), 276-291.

3. J. P. Galeotti, G. Fraser, and A. Arcuri, “Extending a Search-Based Test Generator with Adaptive Dynamic Symbolic Execution (Tool paper),” in Proceedings of the 2014 International Symposium on Software Testing and Analysis, New York, NY, USA, 2014,

4. Alex Groce and Jervis Pinto “A Little Language for Testing” Volume 9058 of the series Lecture Notes in Computer Science pp 204-218 08 April 2015