CS 569 -- Spring Quarter 2016

Static Analysis and Model Checking for Dynamic Analysis

Final Report

Professor: Alex Groce

Name: Kai Shi

ID: 932-504-512

June 2016

Introduction

When a program is finished, we need to test the functions of this program. Therefore, we need to use variables of test cases to test it. And it's important for tester to design test cases; maybe we can call it test suite. Test case is one execution of a program, and it can help tester to find a bug. I think random testing is a good way to test programs. Random testing is also called "fuzzing"[1]. But how to generate random test cases, that's a big problem. Because, random is not really random, if tester uses a huge amount of meaningless test cases, the efficiency will be low. So, testers need to come up with some algorithms to generate effective test cases to test programs. That's the test generation. The algorithm testers come up with should be able to cover more branches and use more effective random data but with minimization of test cases. In my opinion, to generate minimization of effective test cases is test generation, no matter using manual testing or automated testing.
In the class, professor introduces random test method, DFS, BFS checker algorithms to cover most branches. These two algorithms like to traversal tree or graph; so, they have high efficiency and high coverage of branches.
In this project, I choose a similar way to improve the efficiency of random tester. The detail of this algorithm will be discussed in the next section.

Algorithm and Implantation

In this project, I came up with a new algorithm according to NewCover.py, which is discussed in class.

First we set the arguments sequence when testing SUT.

The arguments from the command line for finaltester.py, in order, as follows:

<timeout>, <seed> , <depth>, <width>, <faults>, <coverage>, <running>

<timeout> is time in seconds for testing. <seed> is seed for Python Random. <depth> is maximum length of a test generated. <faults> is used to determine whether saving bugs. <coverage> is used to determine whether to report coverage.

<running> is used to determine whether to produce running information on branch coverage.

For a flexible version: mytester.py, I add another input to print result. This input is < RESULT>, which is used to determine whether to print the results detail, such as number of bugs, actions and running time.

General idea of this algorithm is dividing testing time into two equal parts. In the first part, we use random tester to test the SUT. In this period, it will generate the coverageCount. Then in the rest period, according to the coverageCount, we change the method of testing to improve the efficiency.

I divided first input <TIMEOUT> into two phases. Phase1 runs random testing normally. In phase2 first finding the median coverageCount and the branches tested less than this median will be tested in phase 2. These branches have been tested more than median times will not be executed. This algorithm will avoid only test some of branches and ignore some of others.

First, I spend t/2 time to run randomAction() as normal. After doing this, I just print the coverage. Then in the rest t/2 time, calling a function findMid(). Then, I used the idea from BFS algorithm to append the current state into Queue list if it's in the belowMid list, which returns from findMid() function. Then, adding queue list into curren statements. Last, still using randomAction() to test current statements. Also, printing the how many statements are blow median coverage out of total statements. It should be half size of the total statements.

```
Phase 1:
start = time.time()
print "Testing fo half ", TIMEOUT/2, "time..."
while start is less than Timeout/2:
        Doing randomAction();
        Computing CoverageCount;
printCoverage()


Phase 2:
```

```
start2 = time.time()
while current time - start2 < TIMEOUT/2:
findMid()      → This step will return a Mid coverage and queue list of
  branches below Mid
        Appending the current state into Queue[];
Adding queue list into curren statements;
        Doing randomAction();
        Computing new CoverageCount;
printCoverage()
Printing other results, bug reports, etc.
```

For the findMid() function. First sorting the results of phase1 by their coverageCount. Then, let ss be the half length of sorted list. Then, coverageCount[s] means current state's executed times, and coverageCount[sortedCov[ss]] means the tested times Median branches we found. If coverageCount[s]< coverageCount[sortedCov[ss]], add current state s into belowMid list.

```
def findMid():
  global belowMid
  sortedCov = sorted(coverageCount.keys(),key = lambda x : coverageCount[x])
  ss = len(sortedCov)/2
  for s in sortedCov:
    if coverageCount[s] < sortedCov[ss]:
      belowMid.add(s)
    else:
      break
```

Bugs Saving

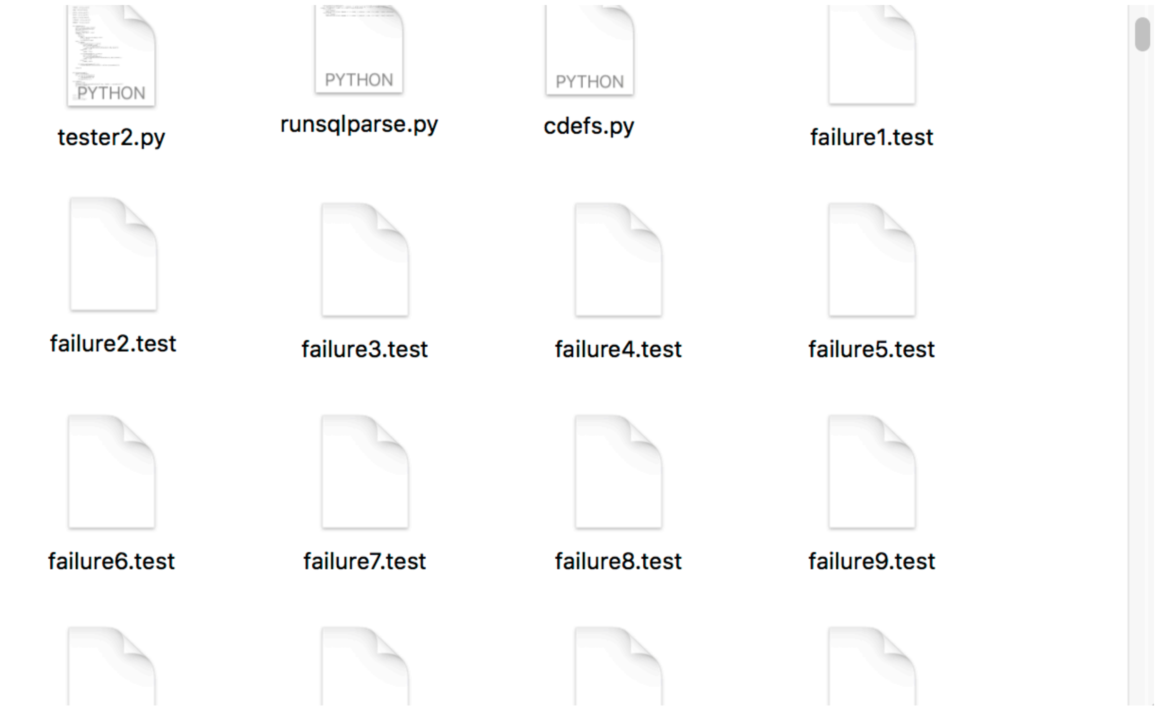
When bugs are found, I use the TSTL API to save bugs. TSTL provides sut.saveTest() API to save the bugs as independent files. This API is like open a file first, then write bugs into it and close it. It similar to:

```
with open("failure"+str(bugs)+".test",'w') as f:
        f.write('\n'+str(bugs)+' bugs found:\n')
        f.write(str(sut.failure())+'\n')
        f.close()
```

This API makes things simple. We just need to call it as a function.

When finding bugs and 5th input is 1, I create the name be "failure"+str(bugs)+".test". This name is the second parameter of sut.saveTest(). The first parameter of sut.saveTest() is sut.test().

Bugs saving situation is like this:

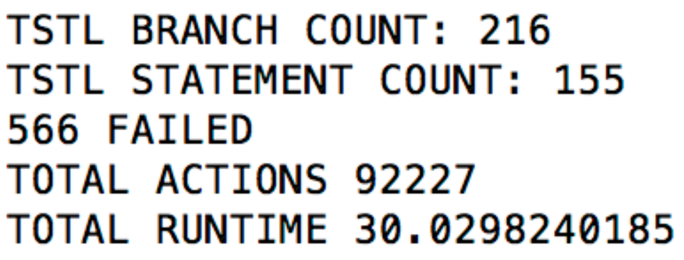


These failures are the results from testing avlblock.py file. In these failure files, steps of finding bugs are recorded.

Branch Count

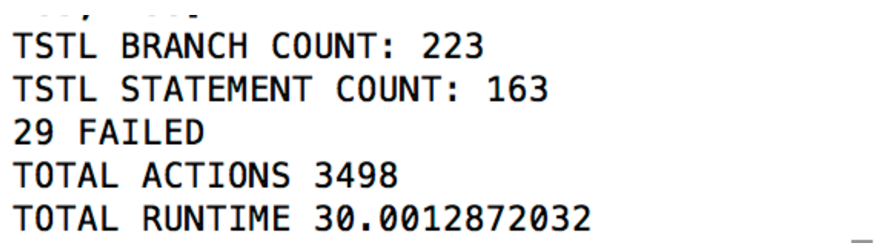
About coverage aspect, the most important evaluation is branch count. At the end of my tester file, I add sut internalreport to print the branch count. For random tester, when I choose to use 30 seconds to test avlblock.py file, the branches it covers is 216 and statement is covers is 155. After improving efficiency, branches count becomes 223 when testing time is 30 seconds too. From the branch count result, we can see more branches are covered at the same time when dividing 30s into two parts and only run below Mid branches in the second 15 seconds. That because in the second 15 seconds, only some uncovered of low covered branches can be tested, so, it’s highly possible to cover more branches. In 30 seconds, my algorithm can cover 7 more branches than random tester. It seems my algorithm only has slight improvement, but if testing time is long, the improvement will be obvious. Here are two results:

Result for Random tester:



```
TSTL BRANCH COUNT: 216
TSTL STATEMENT COUNT: 155
566 FAILED
TOTAL ACTIONS 92227
TOTAL RUNTIME 30.0298240185
```

Result for my algorithm:



```
TSTL BRANCH COUNT: 223
TSTL STATEMENT COUNT: 163
29 FAILED
TOTAL ACTIONS 3498
TOTAL RUNTIME 30.0012872032
```

Conclusion

In this project, I come up with a new algorithm, which divides testing time into two equal parts. In the first half time, testing SUT with random tester method normally. Then according to the branch count result, test low covered branches in the next half time. In conclusion, my algorithm of test generation can cover more branches than random tester generation at the same time. From the result, this algorithm can actually test more branches than random tester.

References

[1] W.Howden,“SystemsTestingandStatisticalTestDataCov- erage”, COMPSAC ‘97, Washington, DC, August 1997, p. 500-504.