

# Final Submission

**Name:** Weiyu Lin

**Date:** 5 June, 2016

**Student ID:** 932-479-491

**Mail:** linweiy@oregonstate.edu

---

## Introduction

I picked MART(Mirror Adaptive Random Test) as my testing alorithm at the beginning of this semaster because of following reasons: Compare to RT(Random Testing), becuase MART inherit the core character of ART[3], MART is more effective than RT cause MART distribute test cases more evenly within the input spaces[1]. Compare to ART, due to the ART need vast quantity of computation(distance calculations, comparisons), MART is much simper and do not need much computation. I changed my algorithm from MART to ART(Adaptive Random Test) at milestone1 since MART is not easy to implement by TSTL. Random testing is known as a black-box software testing method. The inputs will be tested randomly and individually, the test ouput will show if the software get pass or fail. Random testing usually do not have any bias when testing and bugs can be captured quick by random testing. However, it random testing is imprecise (precise as specifications and specifications are not precise enough). Compare with other testing strategies, static program analysis can be served as an example, random testing has poor peformance. The appearance of adaptive random testing, which proposed by T.Y.Chen et al , enhance the ability of random testing by distribute test cases more evenly with the input space[1].The algorithm step shows following:

Step1: sperate input domian as m size subdomains. Determining one source subdomain.

Step2: Implment test case generated by source subdomian.

Step 3: Test cases will be executed in sequential order by fail detection.

Step 4: repeat Step2&3 until bugs are defected (stop if bugs defected).

ART(Adaptive Random Testing) focus on improve the failure-detection effectiveness of random testing[2]. In the milestone2, I improved my algorithm depend on the core code from T.Y. Chen's paper:

```

Algorithm 1:
/*
selected set := { test data already selected };
candidate set := {};
total number of candidates := 10;
*/
function Select The Best Test Data(selected set, candidate set,
total number of candidates);
best distance := -1.0;
for i := 1 to total number of candidates do
candidate := randomly generate one test data from the program
input domain, the test data cannot be in
candidate set nor in selected set;
candidate set := candidate set + { candidate };
min candidate distance := Max Integer;
foreach j in selected set do
min candidate distance := Minimum(min candidate distance,
Euclidean Distance(j, candidate));
end foreach
if (best distance < min candidate distance) then
best data := candidate;
best distance := min candidate distance;
end if
end for
return best data;
end

```

It uses some special predefined values which can be simple boundary values or values that have high tendency of finding faults in the SUT[1]. For this assignment, I did a basic implementation of Adaptive Random Testing Algorithm using the TSTL APIs.

## Implementation

In my final code `finaltester.py`. There's 7 parameters following:

My implementation file `tester2.py` and `tester1.py` can be found here. For the program, there are five parsing parameters in my `tester1.py`, `tester2.py`, which is:

**BUDGET** : BUDGET is a parameter that can budge time in the program.

**SEED** : for random object and using for random number generation in the tester.py

**DEPTH** : DEPTH defines largest length of test in the algorithms.

**WIDTH** : WIDTH is for the deep search config of the algorithms.

**FAULT** : FAULT is applied for check the status (either 0 or 1) fault cheking in the SUTs. The working principle for FAULT is saving the failure found in the current directory.

**COVERAGE** : Coverage of testing of final coverage are reported by this parameter for using `internalReport()` function.

**RUNNING** : checking the brach of coverage(either 0 or 1) when after running by `randomtester.py`. For implementation, you can tpying python BUDGET SEED DEPTH WIDTH FAULT , for example:

```
python tester1.py 30 1 100 1 0 1 1
```

Algorithms with the help of the algorithm in the T.Y. Chen's paper following:

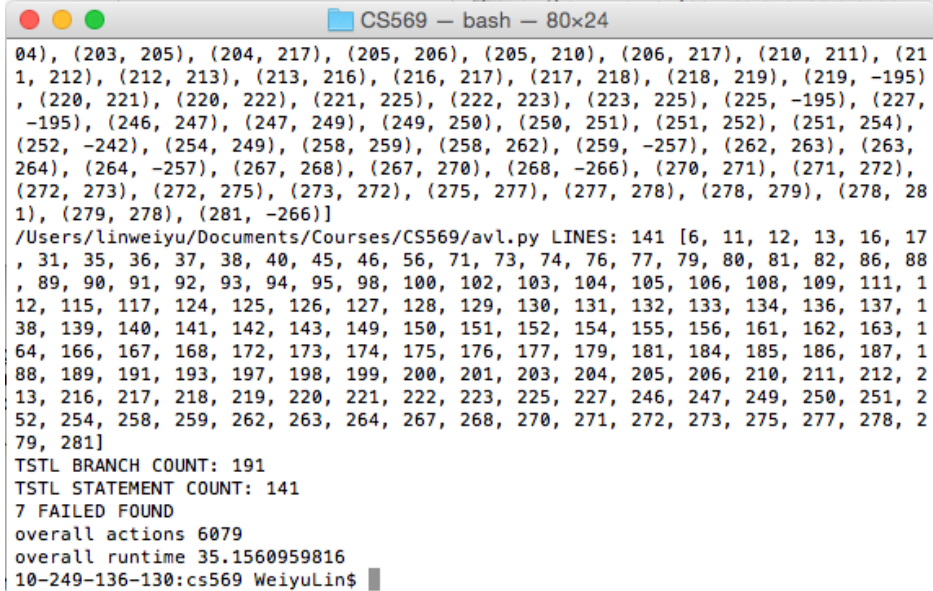
1. At first, parsing parameter (DE)will be deleared.
2. `randomEnable()` will be refered with for loop to collect statement.
3. check the coverage of statement, find the coverage which lower than the tolerance.

4. Store the collected statement, if bugs were found, refer the failure in the TSTL API.
5. print out BUGs found, overall actions and total runtime.

This will test sut.py for 30 seconds, with tests of maximum length 100, a very narrow width (1), and not report or check for failures. It will collect and output an internal coverage report, and also report each new branch as it is discovered. Once the bug get found, it will generate a file called `failureN.test` and stored in the local path.

## Result

For most time, my code is tested by `ALVTree` for the correctness. The result for `python tester2.py 30 1 100 1 1 1 1`



```

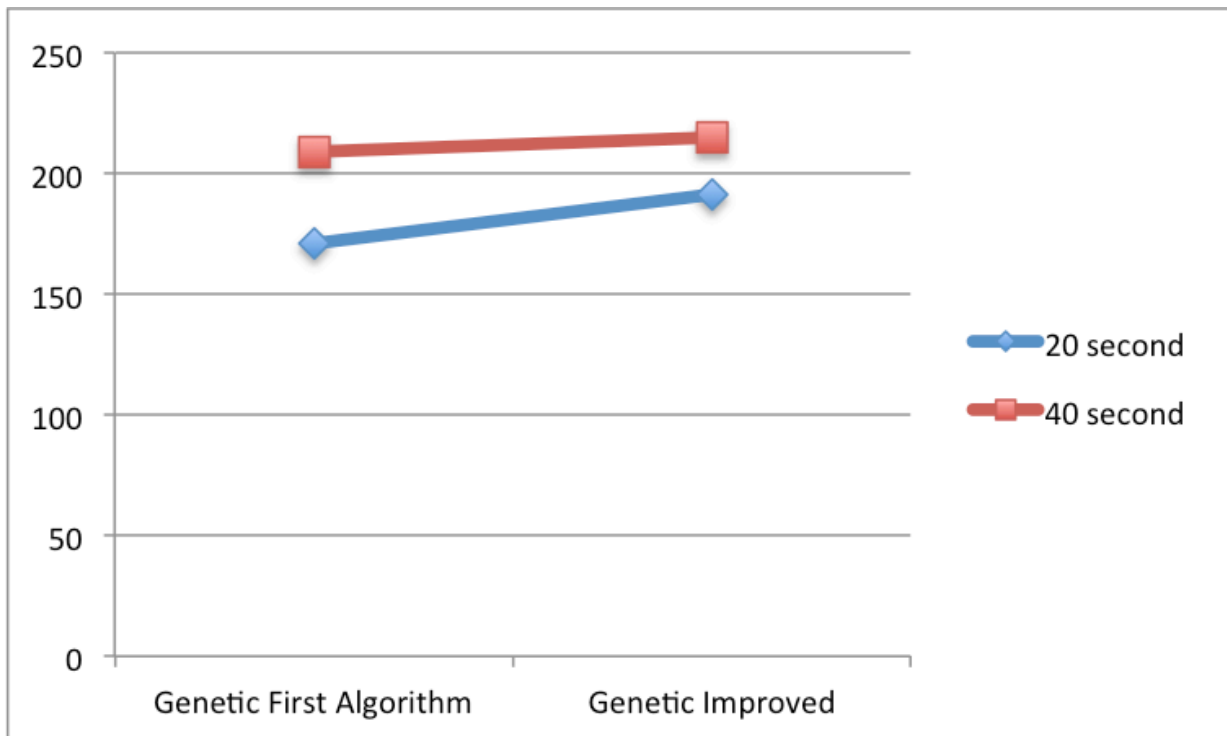
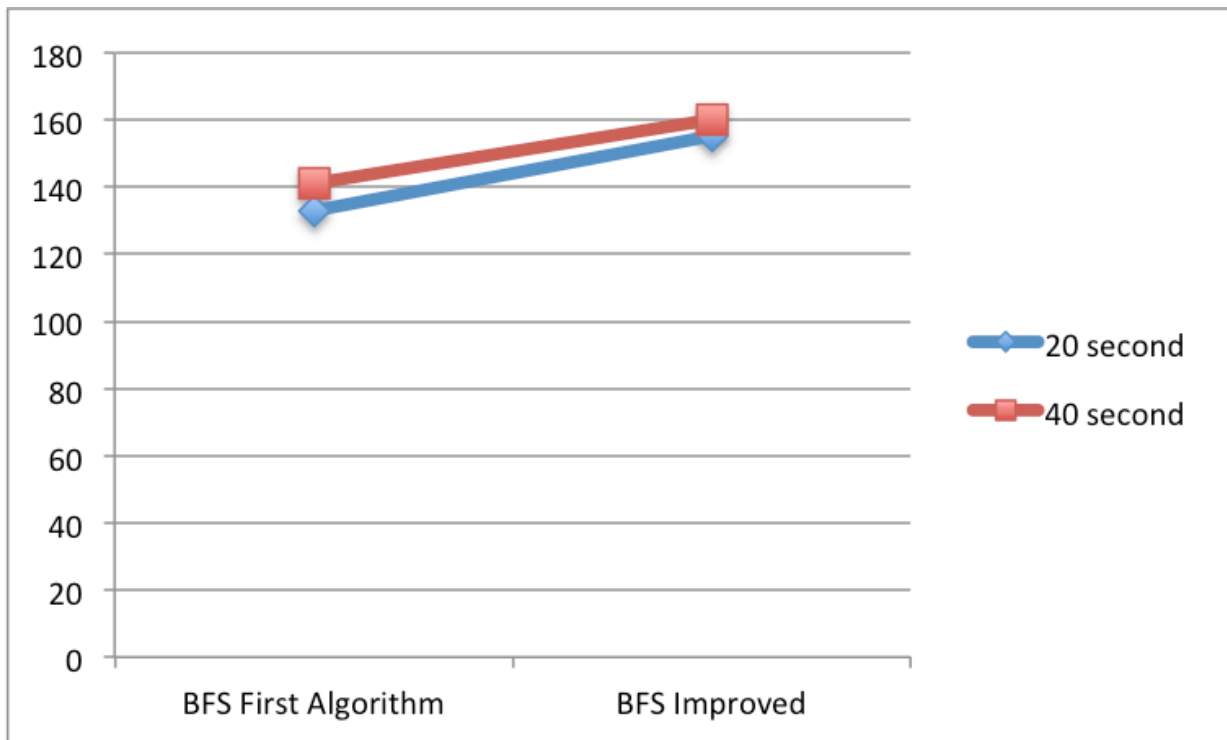
04), (203, 205), (204, 217), (205, 206), (205, 210), (206, 217), (210, 211), (21
1, 212), (212, 213), (213, 216), (216, 217), (217, 218), (218, 219), (219, -195)
, (220, 221), (220, 222), (221, 225), (222, 223), (223, 225), (225, -195), (227,
-195), (246, 247), (247, 249), (249, 250), (250, 251), (251, 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, 28
1), (279, 278), (281, -266)]
/Users/linweiyu/Documents/Courses/CS569/avl.py LINES: 141 [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, 93, 94, 95, 98, 100, 102, 103, 104, 105, 106, 108, 109, 111, 1
12, 115, 117, 124, 125, 126, 127, 128, 129, 130, 131, 132, 133, 134, 136, 137, 1
38, 139, 140, 141, 142, 143, 149, 150, 151, 152, 154, 155, 156, 161, 162, 163, 1
64, 166, 167, 168, 172, 173, 174, 175, 176, 177, 179, 181, 184, 185, 186, 187, 1
88, 189, 191, 193, 197, 198, 199, 200, 201, 203, 204, 205, 206, 210, 211, 212, 2
13, 216, 217, 218, 219, 220, 221, 222, 223, 225, 227, 246, 247, 249, 250, 251, 2
52, 254, 258, 259, 262, 263, 264, 267, 268, 270, 271, 272, 273, 275, 277, 278, 2
79, 281]
TSTL BRANCH COUNT: 191
TSTL STATEMENT COUNT: 141
7 FAILED FOUND
overall actions 6079
overall runtime 35.1560959816
10-249-136-130:cs569 WeiyuLin$

```

shows following:

For the 20 second test, I improved my algorithm from RUNNING: 179 BRANCHES: 171 STATEMENTS: 133 to RUNNING: 179 BRANCHES: 191 STATEMENTS: 141.

For the 40 second test, I improved my algorithm from RUNNING: 179 BRANCHES: 209 STATEMENTS: 155 to RUNNING: 179 BRANCHES: 215 STATEMENTS: 160.



This image clearly indicates that the improved algorithm is better than first one in both BFS and genetics. In conclusion, the ART algorithm has advantage on testing for distributing test cases more evenly with the input space. Because there's still room for improvement at tolerance setting, I think this algorithm still has space to take advantage.

## Reference

- [1] T.Y Chen, H. Leung, and I.K Mak. Adaptive Random Testing, 2004.
- [2] Tsong Yueh Chen, Fei-Ching Kuo, Robert G. Merkel, T.H. Tse. Adaptive Radom Testing: The ART of test case diversity, 2009
- [3] T.Y.Chen, F.C.Kuo, R.g.Merkel, S.P.Ng. Mirror Adaptive Random Testing, 2004. word count: 511

[Word Count: 1078]