# Part 3 for CS569 project

Kun Chen

chenk4@oregonstate.edu

## Algorithm Improvement:

In the Part 2 report for that milestone, an improved BFS algorithm is introduced. 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 for part 2, 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.

In the Part 3 for this milestone, more efforts are given to make modification of the proposed BFS algorithm. I find out when use pure BFS algorithm, it seems have lower efficiency than random test. Then I consider to take the advantage of random test and combine it to BFS. So assume the total running time of the algorithm is t, then I use t1 (t1<t) to run the random test to try to find some states which has bugs and record the depth and bug state as a tuple, and use a list to store all this tuple. Then we sort this list according to the ascending order of recorded depth in the tuples.

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. When given a fixed time t, after we run RT with t1, we still have (t-t1) time to run BFS. So we consider another situation, in the first running loop of BFS, if we run out (t-t1) 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.

## Future Improvement:

In the future, the idea of the coverage part and mutation in GA algorithm are consider to add to this algorithm. I want to have more coverage during the test, and in the class the

mutation and crossover method in GA have a high coverage. And I am quite interested and I am considering to increase the coverage and make it more efficient to find bugs in the next step.

## 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. And at this time, tester2.py written by authors are implemented. The similar bugs in last time have been found too, and the detail is similar to last report, so it is not reported here.

## 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