Reducing the amount of computational overhead of Adaptive Random Testing

Daniel Lin

04/16/16

1 Introduction

Adaptive Random Testing (ART) improves the error detection abilities of Random Testing (RT) by enforcing an uniformly even spread of random test cases over the entire input domain. While ART is very efficient at locating faults, the majority of the implementations of ART requires significant computational costs to ensure that the random tests in ART are spread uniformly across the input domain. This can be problematic when software tests are expensive and contain large test sets. Even though ART is extremely efficient in locating faults and produce test cases that increases the likelihood of finding faults, ART is expensive because it considers every single possible test cases in the input domain. Can we reduce the computation cost of ART by leveraging the geometry of the input parameters?

In this proposal, we will try to integrate iterative partitioning into ART to reduce the amount of computation required for each test generation step in ART. Our proposal is inspired by [1], in which we will divide the input domain into grids with equal cell sizes. Then, the grid cells are categorized into three groups. Each group represents how close the test cases are to successful test cases. [1] claims that using this method, they were able to easily identify grid cells that contain test cases that are far away from successful test samples. The method proposed by [1] is effective because it can potentially reduce the number of test cases required to locate faults. Additionally, the grids that are the furthermost away from the successful test cases can be ruled out immediately using simple search or sorting algorithms.

The testing methods used in [1] can be applied to quickly spot faults in our AVL tree test case. During class, we used breadth and depth first search approach to find faults in our AVL tree. This was demonstrated to be expensive and clumsy. Using the method presented by [1], we are expected to greatly shorten the testing time and the number of test cases to find faults within the AVL tree. For example, using the test method presented in [1], we are expected to greatly shorten the computational time and complexity of the amount of tests required to find faults inside an AVL tree. Even though the storage requirement for the matrix might be big, the algorithm does discard the matrix if there are not failure test cases. To refine the test cases, the matrix's size will be increased to allow the algorithm to obtain more refined test cases.

2 Plan

The plan for this project is as follows:

- April 27-29: Survey papers on ART and related works on RT, ART, and other forms of ART
- May 1-6: Implement [1] with TSTL. This includes finding functions within the TSTL API that supports ART and the methods presented in [1].
- May 9-31: Make the implementation of [1] more efficient. Specifically, make [1] be able to locate faults quicker, optimize the test to include more code coverage, and the test execution time
- May 31 - June 06: Finalize the project, write up documentations, and final optimization.

3 Result

The result of this testing is a bit awful. I had to modify my testing algorithm in the end because the testing method mentioned in here is too complicated to implement. In the end, I implemented a mechanism that allows TSTL to re-test certain branches that fell below a user defined coverage count. This is the algorithm I decided to implement. The test coverage and count is not as good as I expected, as in the graph below, over a 20 second interval, the testing count and coverage is a constant number. I have not investigated too much on this, by my conjecture as in why my testing count and coverage is constant is because there is something wrong in my implementation. Specifically, TSTL did not visit the branches that are below coverage. Another thing worth noting is that the action count for my algorithm rises linearly with respect to time. This might be a good thing because we certainly don't want to have an exponential increase in action counts, or we might run out of computing resources. However, as evident with the class ranking, my algorithm ranks at the near bottem in terms of coverage count. This shows that my method is not as efficient as I thought it would be.

I felt that if I ever was to re-implement this algorithm, I would not have taken my approach. The reason is that my approach does not significantly increase test coverage nor test branch coverages. Even though my testing algorithm is good in identifying potential branches that are low in coverage count, there must be some kind of a harness to go with my testing algorithm in order to produce better test counts. Another thing that I might do is to combine my idea with the papers mentioned in the bibliography. I think that the approach might yield better testing coverages and better testing counts. Increase in testing counts can be done by implementing another testing algorithm that re-runs each under-covered branches found by my algorithm.

Even though the testing time interval was only 20 seconds, I felt that we can run this for a longer period with different running parameters. This might yield a better graph with better coverages. Or, I should modify the metric for determining whether a coverage branch is under-covered or not. If I had the time, I would have re-run my algorithm with more parameters and longer time to see if there is any substantial coverage counts or differences in my graph output.

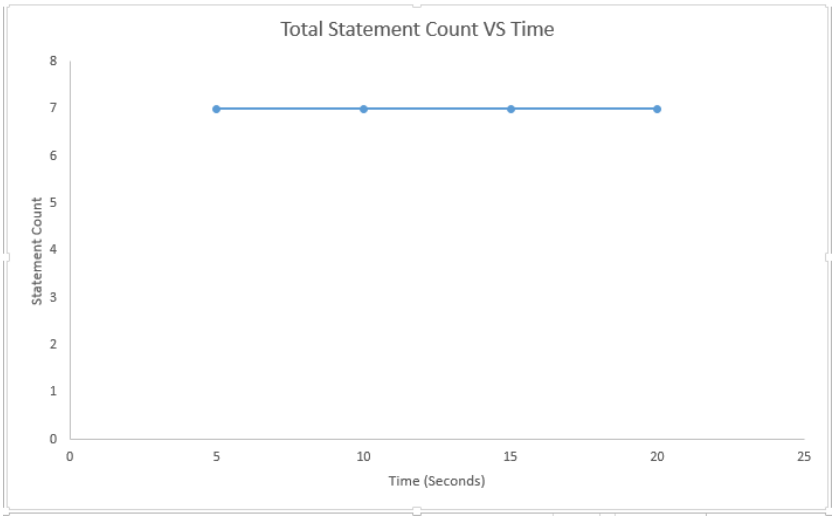

Figure 1: Result 1

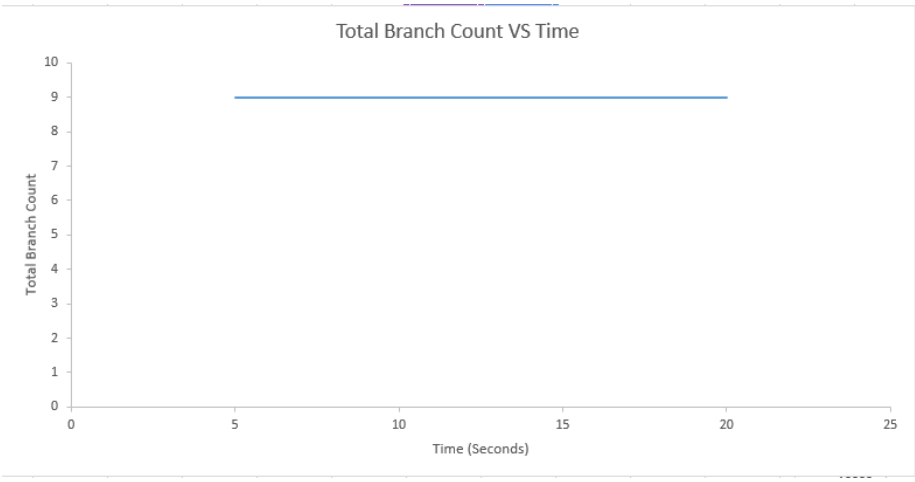

Figure 2: Result 2

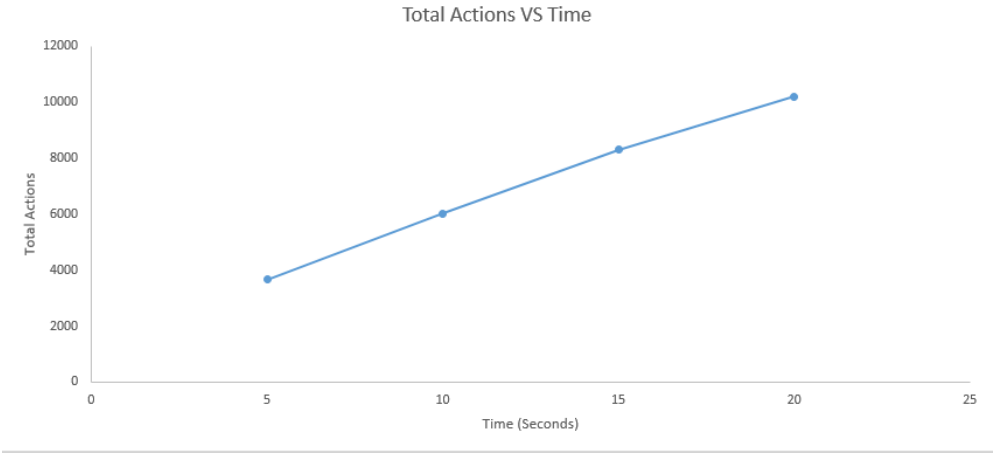

Figure 3: Result 2

4 Related works

Some of the related works are described in the references. Even though my algorithm is not novice (it was implemented in class before by Dr. Groce), it is novel in a sense because it was not implemented in literature. Even though my algorithm didn't work too well to be publishable, however the method is pretty new. In the future, if I have time, I should spend more time refining my method to make the results of my algorithm publishable.

References

[1] T. Y. Chen, De Hao Huang, and Zhi Quan Zhou. "Reliable Software Technologies – Ada-Europe 2006: 11th Ada-Europe International Conference on Reliable Software Technologies, Porto, Portugal, June 5-9, 2006. Proceedings". In: ed. by Luís Miguel Pinho and Michael González Harbour. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006. Chap. Adaptive Random Testing Through Iterative Partitioning, pp. 155–166. ISBN: 978-3-540-34664-7. DOI: 10.1007/11767077_13. URL: http://dx.doi.org/10.1007/11767077_13.

[2] Johannes Mayer. “Formal Methods and Software Engineering: 7th International Conference on Formal Engineering Methods, ICFEM 2005, Manchester, UK, November 1-4, 2005. Proceedings”. In: ed. by Kung-Kiu Lau and Richard Banach. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005. Chap. Adaptive Random Testing by Bisection with Restriction, pp. 251–263. ISBN: 978-3-540-32250-4. DOI: 10.1007/11576280_18. URL: http://dx.doi.org/10.1007/11576280_18.