Testing Monte Carlo Algorithmic Systems

ABSTRACT

Testing systems with non-deterministic outputs whose accuracy improves over repeated iterations of the same inputs presents a unique challenge in determining testing scope and expected results. A thorough understanding of the algorithms under test, and excellent communication between development and testing are essential in test scenario definition and predicting anticipated outcomes. Defining tests and expected behaviors prior to the start of testing is especially crucial in these types of conditions.

A Monte Carlo system uses non-deterministic algorithms to approximate a solution, and generally the more iterations of the algorithms that are run, the better the approximate solution will be. A classic simple example of this type of program is one estimating PI. If one pictures a unit circle inside a square, with radius of 1, then the area of the circle is PI*r^2 and the area of the square is (2r)^2, or 4. The area of the circle divided by the area of the square then comes to (PI*r^2) / (2r)^2, or PI / 4. If this ratio is computed, one can multiply it by 4 to get PI. A "Monte Carlo" approach to this would be to randomly select points on the square. If these points lie within the unit circle, it counts as a "hit", so the equation would be:

Pi ~ 4 * (Number of Hits) / (Number of Iterations).

As an example, say we ran the program with 10 iterations, and got 8 "hits", i.e. 8 of the points lay within the unit circle (i.e. point p satisfies condition: p = x^2 + y^2 <= 1). Then the equation would be:

Pi = (4 * 8) / 10 = 32 /10 = 3.2

The more iterations are input to the program, the closer one can get to a value for PI (defined as being 3.1415…).

Example source code for an implementation of such a program in c can be found at:
http://www.dartmouth.edu/~rc/classes/soft_dev/C_simple_ex.html

Now there are many types of solutions in enterprise software which use this basic strategy, the so-called "Monte Carlo" strategy, because, for example in the situation of computing PI, random numbers are being used to generate selected points within the square containing the unit circle, and Monte Carlo is the name of a famous casino, casinos being examples of the use of random numbers. A "Monte Carlo" type of solution will have two essential characteristics:

1) Each input / output pair is predicated on random or non-determinist processes
2) Increasing the number of input/output pair cycles for the same inputs during program execution (should) lead to better (more accurate) output.

How would one go about testing this sort of application? One would have to fit two factors into account: the non-determinism associated with the outputs, and the expected behavior that, despite this non-determinism, better or more accurate outputs should be obtained by increasing the number of iterations in execution. Here, for example, we could have 3 test scenarios, one with 10 iterations (Test Case 1), one with 100 iterations (Test Case 2), and one with 1000 iterations (Test Case 3). We could run these 3 simple scenarios each, say, 100 times. Now, we could put the outputs for the 100 runs of Test Case 1 into a spreadsheet, the outputs for the 100 runs of Test Case 2 into its own spreadsheet, and the outputs for Test Case 3 into a third spreadsheet. Looking at say, the spreadsheet for the results of Test Case 1, we can expect that (probably) each run will have a different output value, and the same with the other two test cases, due to the indeterminist nature of the algorithm under test. However, looking at the results of Test Case 1 in a spreadsheet, we can calculate the standard deviation from what we know (from talking to development or the requirements team) is a good result. So, for example, in preparing this article, I implemented a c version of the Monte Carlo example, and found that if I input 10 iterations then my output could vary widely from 3.14, which is known as a good approximation for PI. Say, for the sake of argument, that the standard deviation should be less than or equal to plus or minus 0.5 from 3.14, so anywhere from 2.64 to 3.64 would be the expected results for Test Case 1 (10 iterations). However, Test Case 2 (100 iterations) should have a smaller standard deviation, say, for the sake of argument, around 0.25, so the results could be anywhere from 2.89 to 3.39. Test Case 3 (1000 iterations) could have a still smaller standard deviation, say, 0.12, so the expected results could be anywhere from 3.02 to 3.26. (Incidentally these numbers are generally in the ball park of what I observed testing my own c implementation of the "Monte Carlo PI" problem.)

To test these types of applications, a test engineer needs to obtain what the expected "standard deviation" for a given number of iterations might be, and then, test for that expected standard deviation for that given number of iterations. The test engineer needs to further obtain what the expected reduction in the standard deviation of results should be when increasing the number of iterations by a certain reasonable amount (such as in this case, increasing by powers of ten). Then the tester's expected results are, for a given test case or number of iterations, that the standard deviation from multiple runs of that test case is not greater than the one expected.

Having well-defined numbers of iterations that one plans to test, and having at least an idea of what the error or standard deviation should be for each number of iterations, and having an agreed-upon number of times each test case will be run to get the "actual" or observed standard deviation, are crucial factors. Working with development or the requirements team, or with one's test manager, one can get agreed-upon criteria for testing prior to commencing test runs, which is important, because with non-deterministic outputs, it can be hard to determine what is an "expected result" or not, without very clear, specific, agreed-upon criteria defined from the outset.

Some test tool API's can be readily configured to allow for non-deterministic results, so, for example, if Test Case 1 (10 iterations) has a standard deviation of 0.5 around a certain quantity (in our example, meaning the expected results for multiple runs for Test Case 1 can be anywhere from 2.64 to 3.64), then the test tool can be configured to allow for this range. A unit test API I have used for XML testing with non-deterministic outputs is XMLUnit (http://xmlunit.sourceforge.net/), an extension of JUnit which has classes one can over-ride to define one's own range of expected outputs. For non-unit type of testing, tools on the market today such as SilkTest, Rational Robot, et al are programmable/configurable enough to customize automated test solutions to allow expected results to fall within a pre-defined range. With such a tool, either developed in-house or purchased through a vendor, one can automate expected results, even in a situation of non-deterministic outputs.

There may be cases where it is not known, or cannot be known due to the nature of the problem what an expected "standard deviation" is for a given test case, because the "optimal" result is not known, unlike in the case of finding PI. If this is the case, then perhaps the best thing that can be tested is what I might call a "relative standard deviation", meaning, the outputs should have "better" answers as one increases the number of iterations, so, 1000 iterations should lead to "better" outputs than 100 iterations, which themselves should be "better" than 10 iterations. Sometimes "better" could be a limit of some kind, i.e., outputs tending towards a certain limit. In other cases, a higher valued set of outputs could be "better", in others, a lower set of values. For example, if one is testing a computer game where one plays miniature golf in a virtual arcade against an automaton, and one is testing this automaton which is designed to get better at its game over multiple runs of the game, then one would expect the automaton to achieve a lower average score in a test scenario which included 1000 runs of the game, rather than 100.

Generally, irrespective of the sort of "Monte Carlo" application one is testing, one should have 1) Agreed-upon test scenarios (Test with 10, 100, 1000 iterations, for example) 2) Agreed-upon number of times each scenario needs to be run to get statistically significant results 3) Agreed-upon expected results for each scenario if possible, or at least an agreed upon expected result of the relationships between the scenarios (i.e. more iterations leading to "better" results). This needs to be done at the start of testing, so that one can know when tests "pass" or "fail" and the conditions of these eventualities, even if the application under test does not have deterministic outputs. Needless to say, this type of testing requires excellent communication between the development teams and the testing teams, so the testers can define the scope and expected results of their tests, and so that development can be apprised of unanticipated results in these tests, and possibly update their models of expected behavior accordingly.

BIO

Frank Erdman is a Software QA Engineer in Austin, Texas. His background includes testing mobile workforce management systems which use stochastic algorithms for mobile resource planning, forecasting, and scheduling. Unit test and automation tools he has worked with include JUnit, XMLUnit, CPPUnit, and Borland SilkTest. He is COMPTIA A+ and Network+ certified, and was a website consultant for Calliope, LLC, a talent agency in San Antonio.

Email: FrankErdman2000@yahoo.com Blog: http://blogkinnetic.blogspot.com