NTimeUnit: Testing Software with Respect to Execution Time

Robbie Burda, Lam Dinh, David Malec, Alek Ortynski, Joe Hummel
Dept of Mathematics and Computer Science
Lake Forest College
Lake Forest, IL 60045
hummel@lakeforest.edu

Abstract

Testing is an important part of the software development process. Many forms of testing are used, including unit testing, integration testing, and stress testing. Unit testing is particularly important in that it allows errors to be detected sooner, saving time and money.

An often overlooked aspect of unit testing is *expected performance*. For example, binary search is an O(lgn) algorithm — what if a binary search method returns the correct result, but performs in O(n) time. Is it still correct? Most would argue it is not.

We are developing an extension to NUnit called *NTimeUnit* that allows the tester to assert the expected performance of a method. The testing framework makes repeated calls to the method for a range of n, collects timing data, and attempts to identify the big-Oh category the method falls into. We have a preliminary implementation of NTimeUnit, and present our initial findings.

1. Introduction

Testing is an important part of the software development process. Many forms of testing are used, including unit testing, integration testing, and stress testing. Unit testing is particularly important in that it allows errors to be detected sooner, saving time and money. Unit testing is a critical component of numerous software engineering processes (such as the Agile methodology [1, 2] and Test Driven Development [3, 4]), and supported by software engineering tools (such as JUnit and NUnit [5, 6, 7]).

Unit testing is driven by assertions that confirm expected behavior. The tester asserts that method M, when passed X, should return Y, and confirms this through execution. The tester repeats this process for as many values of X as possible, and for all methods M. While this does not prove software correctness, it helps identify programming errors, and increases confidence in the software's correctness. It also provides a mechanism for regression testing (re-testing after a change has been made).

For example, suppose the ArrayList AL contains common English words in alphabetical order. Unit testing of Java's built-in Collections.binarySearch method might look as follows:

```
/** @attribute Test() */
public void test1(ArrayList AL)
{
  assert( Collections.binarySearch(AL, "apple")    >= 0 );
  assert( Collections.binarySearch(AL, "familiar") >= 0 );
  assert( Collections.binarySearch(AL, "familar")  < 0  );
  .
  .
  .
}
```

The comment-based attribute *Test()* identifies the method to the testing framework as a test method. If the asserted expression evaluates to false, assert throws an exception to be caught by the testing framework. The testing framework then reports this test as a failure. If the test runs to completion and returns normally, the test is reported as a success.

Note that the testing in this case can be made stronger by comparing the index returned by binary search to that of linear search:

```
assert( Collections.binarySearch(AL, "apple") ==
        AL.indexOf("apple") );
assert( Collections.binarySearch(AL, "familiar") ==
        AL.indexOf("familiar") );
```

An often overlooked aspect of unit testing is *expected performance*. For example, binary search is an O(lgn) algorithm — what if a binary search method returns the correct result, but performs in O(n) time. Is it still correct? Most would argue it is not, since the method

is not performing as expected. Discovering this requires both a testing and a timing framework.

To this end, we are developing an extension to NUnit [7] called *NTimeUnit* that allows the tester to assert the expected performance of a method. The testing framework makes repeated calls to the method for a range of n, collects timing data, and attempts to identify the big-Oh category the method falls into. If the observed performance does not match the tester's expected performance, an exception is thrown and the test case fails.

2. Related Work

While a great deal of work has been done in the area of unit testing for correctness (e.g. [5, 6, 7] represent just the tip of the iceberg), most work on automatic complexity analysis is focused on estimation (e.g. [8, 9]). This type of approach has two obvious advantages: (1) the tester does not need to know the expected performance, and (2) actual execution is not required. However, the power of this approach is limited in the context of imperative programming languages. NTimeUnit has the advantage that it works for any programming language, and does not require a sophisticated analysis tool. The ideas behind NTimeUnit can be integrated easily into any unit testing framework.

3. Our Approach

Our approach is to extend an existing unit testing framework to include support for the specification and testing of performance. Since we are working in .NET, we based our work on the popular NUnit tool [7], for which complete source code is available. NUnit is the .NET equivalent of the Java-based JUnit tool [6].

Our software package is called NTimeUnit, and consists of NUnit + additional classes and internal modifications.

3.1 NTimeUnit Attributes

NTimeUnit offers an additional attribute for specifying expected performance — i.e. a *performance test*. The format is as follows:

```
/** @attribute Analyze(bigOh, initialN, finalN, runsPerN) */
public void test1(int n)
{
   .
   .
   .
}
```

The first parameter *bigOh* is a string denoting the expected performance for this test across a range of n. This can be any one of the following strings: "O(1)", "O(lgn)", "O

(n)", "O(nlgn)", "O(n^2)", "O(n^3)", and "O(2^n)". The second and third parameters (*initialN* and *finalN*) specify the range of n to be used for testing, such 1000 .. 128000. NTimeUnit starts the testing process by calling the test method and passing initialN. The value is then doubled on each subsequent call to the test method until it exceeds finalN. Finally, the fourth parameter *runsPerN* controls how the time for each n is computed. For a given value of n, the reported execution time is the average elapsed time over the course of R calls to the test method (where R = *runsPerN*). Using the average time across a set of runs attempts to minimize the impact of the occasional anomaly when timing programs on multiprocessing systems.

3.2 NTimeUnit Test Methods

When using NTimeUnit, test methods are written to operate on n data items, where n is a parameter to the test method:

```
/** @attribute Analyze(bigOh, initialN, finalN, runsPerN) */
public void test1(int n)
{
   .
   .
   .
}
```

For example, if we are performance testing the implementation of a sorting algorithm, the test method would be written to apply the sorting algorithm to n items. Since the entire call to the test method is timed, code to setup or teardown the test should be avoided since it could impact the time complexity of the result. For this reason, NTimeUnit (like NUnit) supports the notion of setup methods:

```
/** @attribute AnalyzeSetup() */
public void test1setup(int n)
{
   .
   .
   .
}
```

The setup method is called before each test run, and is not timed. For one-time test initialization (such as reading test data from a file), the test class's constructor is typically used.

3.3 NTimeUnit Analysis

The most important aspect of NTimeUnit is the analysis component. Given a set of m times, the analyzer attempts to fit the data to one of the following curves: O(1), O(lgn), O (n), O(nlgn), $O(n^2)$, $O(n^3)$, or $O(2^n)$. As mentioned earlier, the times are average elapsed times across a specific range of n: initialN, initialN * 2, initialN * 4, …, finalN. Doubling the value of n for each subsequent test simplifies the analysis. For example, if the time for

each subsequent test doubles, then the observed performance is O(n). If the time for each subsequent test quadruples, then the observed performance is $O(n^2)$. And if the time for each subsequent test increases by a constant c, then the observed performance is O(lgn). Why? Suppose test(n) takes T time. If the algorithm is O(lgn), then test(2n) will take O (lg2n) = O(lgn + 1) = T + O(1) time.

Given a set of times $\{T_1, T_2, \ldots, T_m\}$, the analyzer proceeds as follows. For each complexity function F, the analyzer computes the set of coefficient as if F is the function describing the times. This is the set $\{c_1, c_2, \ldots, c_m\}$, where $c_i = T_i / F(\text{initialN} * 2^i)$. In other words, for a given test value of n, we divide the observed time T by the theoretical time predicted by the function F. If the function F "fits" the timing data, the coefficients will be the same.

Given the imprecise nature of timing program execution, it is unrealistic to expect a function to fit perfectly. However, the fit must be very, very good to avoid false positives. The analyzer improves accuracy by normalizing the coefficients based on c_m, the last coefficient. This coefficient is the most accurate since it is derived from the test with the largest n, and thus the longest running time. Given a set of coefficients $\{c_1, c_2, \ldots, c_m\}$, the analyzer divides each coefficient by c_m, yielding a set of ratios $\{r_1, r_2, \ldots, r_m\}$. In this case, if the function is a perfect fit, the set of ratios will be {1.0, 1.0, …, 1.0}.

3.4 An Example

Let's look at a complete example. The algorithm we are going to performance test is selection sort, an algorithm with $O(n^2)$ behavior:

```
public static void selectionSort(ISortable ds, int N)
{
      for (int i = 0; i < N-1; i++)
      {
            for (int j = i+1; j < N; j++)
            {
                  Comparable  obj1;
                  Object      obj2;

                  obj1 = (Comparable) ds.get(i);
                  obj2 = ds.get(j);

                  if (obj1.compareTo(obj2) > 0)  // swap
                  {
                        ds.set(i, obj2);
                        ds.set(j, obj1);
                  }
            }//for
      }//for
}
```

The test class imports the NUnit framework, and then reads a collection of words from a file to provide data for sorting:

```
import NUnit.Framework.*;

/** @attribute TestFixture() */
public class TestHarness
{
      private ArrayList  words;

      public TestHarness() throws java.io.IOException
      {
            // read 256,000 words from a randomly-ordered text file:
            words = StringsIO.GetStrings("words256000.txt", 256000);

         // duplicate to get up to 1Meg of words:
         for (int i = 0; i < 3; i++)
                  for (int j = 0; j < 256000; j++)
                        words.add(words.get(j));
      }
```

Next comes the setup method, which ensures this is a list of n words to sort when the time comes:

```
      private ArrayList  list;

   /** @attribute AnalyzeSetUp() */
      public void Setup(int n)
      {
            list = new ArrayList(n);

            for (int i = 0; i < n; i++)
                  list.add(words.get(i));
      }
```

Finally, the performance test is simply a call to selectionSort to sort the requested n items. The associated attribute conveys our expected analysis of $O(n^2)$, specifies a small range of n given the slowness of the algorithm, and asks for the average of 3 runs per n:

```
      /** @attribute Analyze("O(n^2)", 1000, 16000, 3) */
      public void TimeSelectionSort(int n)
      {
            Algorithms.selectionSort(list, n);
      }

}//TestHarness
```

Here are timing results from a sample run (in clock ticks):

1000	2000	4000	8000	16000
781250	2760417	11250000	43958333	175989583

For each of the complexity functions, the ratios are:

O(1)	0.0044	0.0157	0.0639	0.2498	1.0
O(lgn)	0.0062	0.0199	0.0746	0.2690	1.0
O(n)	0.0710	0.1255	0.2557	0.4995	1.0

O(nlgn)	0.0995	0.1598	0.2984	0.5381	1.0
$O(n^2)$	1.136	1.0038	1.0228	0.9991	1.0
$O(n^3)$	18.183	8.0308	4.0912	1.9982	1.0
$O(2^n)$	Inf	NaN	NaN	NaN	NaN

As you can see from the data, the only function that comes close to fitting is $O(n^2)$. And other than the first data point, $O(n^2)$ is a perfect fit.

4. Preliminary Results

The implementation of NTimeUnit is based on C#, NUnit version 2.2, .NET version 1.1, and Visual Studio .NET 2003. We analyzed three algorithms hand-written in J# (the .NET version of Java): selection sort, merge sort, and quicksort. The test methods mirror the example shown in section 3.4:

```
/** @attribute Analyze("O(n^2)", 1000, 16000, 3) */
public void TimeSelectionSort(int n)
{
      Algorithms.selectionSort(list, n);
}

/** @attribute Analyze("O(nlgn)", 4000, 1024000, 3) */
public void TimeMergeSort(int n)
{
      Algorithms.mergeSort(list, n);
}

/** @attribute Analyze("O(nlgn)", 4000, 1024000, 3) */
public void TimeQuickSort(int n)
{
      Algorithms.quickSort(list, n);
}
```

After analyzing the timing data for numerous execution runs, the analysis engine was augmented in three ways. Firstly, the analyzer is programmed to ignore the first timing data point T_1; this value was always an outlier, even though an initial run was performed, and ignored, to "prime" the system. Secondly, there needed to be a margin of error — i.e. an allowed variance from 1.0 — to account for timing imprecision. This was set at 0.2. Finally, the analyzer was allowed to drop at most one additional outlier. Once these modifications were in place, NTimeUnit correctly matched the observed performance to the expected performance in each test case.

5. Conclusions and Future Work

Unit testing is an important part of software development, and NTimeUnit presents an interesting twist on the traditional notion of testing correctness. NTimeUnit has the potential to become a very useful tool. However, before it can be put into practice, much

more work must be done collecting results across a far greater variety of algorithms and implementations.

Other ideas for future work include an automatic determination of the values of n to test with, based on spiking an initial value and then adjusting up or down based on the elapsed time. A finer-grain timing mechanism may help improve precision, and a focus on user time instead of elapsed time (user + system) might help as well.

References

[1] Augustine, S., Payne, B., Sencindiver, F., and Woodcock, S. 2005. Agile project management: steering from the edges. *Commun. ACM* 48, 12 (Dec. 2005), 85-89.

[2] http://agilemanifesto.org/.

[3] George, B. and Williams, L. 2003. An initial investigation of test driven development in industry. In *Proceedings of the 2003 ACM Symposium on Applied Computing* (Melbourne, Florida, March 09 - 12, 2003). SAC '03. ACM Press, New York, NY, 1135-1139.

[4] Miller, K. W. 2004. Test driven development on the cheap: text files and explicit scaffolding. *J. Comput. Small Coll.* 20, 2 (Dec. 2004), 181-189.

[5] Noonan, R. E. and Prosl, R. H. 2002. Unit testing frameworks. In *Proceedings of the 33rd SIGCSE Technical Symposium on Computer Science Education* (Cincinnati, Kentucky, February 27 - March 03, 2002). SIGCSE '02. ACM Press, New York, NY, 232-236.

[6] http://junit.org/.

[7] http://nunit.org/.

[8] Reistad, B. and Gifford, D. K. 1994. Static dependent costs for estimating execution time. In *Proceedings of the 1994 ACM Conference on LISP and Functional Programming* (Orlando, Florida, United States, June 27 - 29, 1994). LFP '94. ACM Press, New York, NY, 65-78.

[9] Gómez, G. and Liu, Y. A. 2002. Automatic time-bound analysis for a higher-order language. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation* (Portland, Oregon, January 14 - 15, 2002). PEPM '02. ACM Press, New York, NY, 75-86.