Clever Algorithms: Unit Testing Algorithms*

Jason Brownlee
jasonb@CleverAlgorithms.com
The Clever Algorithms Project
http://www.CleverAlgorithms.com

December 06, 2010
Technical Report: CA-TR-20101206a-1

Abstract

The Clever Algorithms project aims to describe a large number of Artificial Intelligence algorithms in a complete, consistent, and centralized manner, to improve their general accessibility. The project makes use of a standardized algorithm description template that uses well-defined topics that motivate the collection of specific and useful information about each algorithm described. This report provides an introduction to software testing and the testing of Artificial Intelligence algorithms.

Keywords: `Clever, Algorithms, Test, Unit, Test, Testability`

1 Introduction

The Clever Algorithms project aims to describe a large number of algorithms from the fields of Computational Intelligence, Biologically Inspired Computation, and Metaheuristics in a complete, consistent and centralized manner [2]. The project requires all algorithms to be described using a standardized template that includes a fixed number of sections, each of which is motivated by the presentation of specific information about the technique [4].

This report provides an introduction to software testing and the testing of Artificial Intelligence algorithms. Section 2 introduces software testing and focuses on a type of testing relevant to testing algorithms called unit testing. Section 3 provides a specific example of an algorithm and a prepared suite of unit tests, and Section 4 provides some rules-of-thumb for testing algorithms in general.

2 Software Testing

Software testing in the field of Software Engineering is a process in the life-cycle of a software project that verifies that the product or service meets quality expectations and validates that software meets the requirements specification. Software testing is intended to locate defects in a program, although a given testing method cannot grantee to locate all defects. As such, it is common for an application to be subjected to a range of testing methodologies throughout the software life-cycle, such as unit testing during development, integration system testing once

modules and systems are completed, and user acceptance testing to allow the stakeholders to determine if their needs have been met.

Unit testing is a type of software testing that involves the preparation of well-defined procedural tests of discrete functionality of a program that provide confidence that a module or function behaves as intended. Unit tests are referred to as 'white-box' tests (contrasted to 'black-box' tests) given that they are written with full knowledge of the internal structure of the functions and modules under tests. Unit tests are typically prepared by the developer that wrote the code under test and are commonly automated, themselves written as small programmers that are executed by a unit testing framework (such as JUnit for Java). The objective is not to test each path of execution within a unit (called complete-test or complete-code coverage), but instead to focus tests on areas of risk, uncertainty, or criticality. Each test should focus on one aspect of the code (test one thing) and are organized into test suites of commonality.

Some of the benefits of unit testing include:

- *Documentation*: The preparation of a suite of tests for a given system provide a type of programming documentation highlighting the expected behavior of functions and modules and providing examples of how to interact with key components.
- *Readability*: Unit testing encourages a programming style of small modules, clear input and output and possibly fewer inter-component dependencies. Code written for easy of testing (testability) may be easier to read and follow by third parties.
- *Regression*: Together, the suite of tests can be executed as a regression-test of the system. The automation of the tests means that any defects caused by changes to the code can easily be identified, when a defect is found that slipped through, a new test can be written to catch it to ensure it will be identified in the future.

Unit tests were traditionally written after the program was completed. A popular alternative is to prepare the tests before the functionality of the application is prepared, called Test-First or Test-Driven Development (TDD). In this method, the tests are written and executed, failing until the application functionality is written to make the test pass. The early preparation of tests allow the programmer to consider the behavior required from the program and the interfaces or functions that the program needs to expose, before they are written.

The concerns of software testing are very relevant to the development, investigation, and application of Metaheuristic and Computational Intelligence algorithms. In particular the strong culture of empirical investigation and prototype-based development demand a baseline level of trust in the systems that are presented in articles and papers. Trust can be instilled in an algorithm by assessing the quality of the algorithm implementation itself. Unit testing is lightweight (requiring only the writing of automated test code) and meets the needs of promoting quality and trust in the code while prototyping and developing algorithms. It is strongly suggested as a step in the process of empirical algorithm research in the fields of Metaheuristics, Computational Intelligence, and Biologically Inspired Computation.

3 Unit Testing Example

This section provides an example of an algorithm and its associated unit tests as an illustration of the presented concepts. Section 3.1 presents the Genetic Algorithm and Section 3.2 presents the unit tests for the Genetic Algorithm.

3.1 Algorithm

Listing 1 presents the source code for the Genetic Algorithm in the Ruby Programming language. An important consideration in using the Ruby test framework, is ensuring that functions of the

algorithm are exposed for testing and that the algorithm demonstration itself does not execute. This is achieved through the use of the(`if __FILE__ == $0`) condition that ensures the example only executes when the file called directly, allowing the functions to be imported and executed independently by a unit test script.

The Genetic Algorithm is used as the basis for example as it is well understood ad provides an implementation that is a good case for demonstrating unit testing methodology [3]. The algorithm is very modular with its behavior partitioned into well-contained functions, most of which are independently testable.

The `reproduce` function has some dependencies although is still testable. The `search` function is the only monolithic function, which both depends on all other functions in the implementation (directly or indirectly) and is difficult to unit test. At best the `search` function may be a case for system testing addressing functional requirements, such as "*does the algorithm deliver optimized solutions*".

```
def onemax(bitstring)
  sum = 0
  bitstring.each_char {|x| sum+=1 if x=='1'}
  return sum
end

def random_bitstring(num_bits)
  return (0...num_bits).inject(""){|s,i| s<<((rand<0.5) ? "1" : "0")}
end

def binary_tournament(population)
  s1, s2 = population[rand(population.size)], population[rand(population.size)]
  return (s1[:fitness] > s2[:fitness]) ? s1 : s2
end

def point_mutation(bitstring, prob_mutation)
  child = ""
   bitstring.each_char do |bit|
    child << ((rand()<prob_mutation) ? ((bit=='1') ? "0" : "1") : bit)
  end
  return child
end

def uniform_crossover(parent1, parent2, p_crossover)
  return ""+parent1 if rand()>=p_crossover
  child = ""
  parent1.length.times do |i|
    child << ((rand()<0.5) ? parent1[i].chr : parent2[i].chr)
  end
  return child
end

def reproduce(selected, population_size, p_crossover, p_mutation)
  children = []
  selected.each_with_index do |p1, i|
    p2 = (i.even?) ? selected[i+1] : selected[i-1]
    child = {}
    child[:bitstring] = uniform_crossover(p1[:bitstring], p2[:bitstring], p_crossover)
    child[:bitstring] = point_mutation(child[:bitstring], p_mutation)
    children << child
    break if children.size >= population_size
  end
  return children
end

def search(max_generations, num_bits, population_size, p_crossover, p_mutation)
  population = Array.new(population_size) do |i|
```

```
    {:bitstring=>random_bitstring(num_bits)}
  end
  population.each{|c| c[:fitness] = onemax(c[:bitstring])}
  best = population.sort{|x,y| y[:fitness] <=> x[:fitness]}.first
  max_generations.times do |gen|
    selected = Array.new(population_size){|i| binary_tournament(population)}
    children = reproduce(selected, population_size, p_crossover, p_mutation)
    children.each{|c| c[:fitness] = onemax(c[:bitstring])}
    children.sort!{|x,y| y[:fitness] <=> x[:fitness]}
    best = children.first if children.first[:fitness] >= best[:fitness]
    population = children
    puts " > gen #{gen}, best: #{best[:fitness]}, #{best[:bitstring]}"
    break if best[:fitness] == num_bits
  end
  return best
end

if __FILE__ == $0
  # problem configuration
  num_bits = 64
  # algorithm configuration
  max_generations = 100
  population_size = 100
  p_crossover = 0.98
  p_mutation = 1.0/num_bits
  # execute the algorithm
  best = search(max_generations, num_bits, population_size, p_crossover, p_mutation)
  puts "done! Solution: f=#{best[:fitness]}, s=#{best[:bitstring]}"
end
```

Listing 1: Genetic Algorithm in the Ruby Programming Language

3.2 Unit Tests

Listing 2 provides the `TC_GeneticAlgorithm` class that makes use of the built-in Ruby unit testing framework by extending the `Test::Unit::TestCase` class. The listing provides an example of 10 unit tests for 6 of the 7 functions in the Genetic Algorithm implementation. Two types of unit tests are provided:

- *Deterministic*: Tests that directly test the function in question, addressing questions such as: does `onemax` add correctly? and does `point_mutation` behave correctly?
- *Probabilistic*: Tests the probabilistic properties of the function in question, addressing questions such as: does `random_bitstring` provide an expected 50/50 mixture of 1's and 0's over a large number of cases? and does `point_mutation` make an expected number of changes over a large number of cases?

This second type that tests for probabilistic expectations is a weaker form of unit testing that can be used to either provide additional confidence to deterministically tested functions, or to be used as a last resort when direct methods cannot be used.

Given that a unit test should 'test one thing' it is common for a given function to have more than one unit tests. The `reproduce` function is a good example of this with three tests in the suite. This is because it is a larger function with behavior called in dependent functions which is varied based on parameters.

```
class TC_GeneticAlgorithm < Test::Unit::TestCase

    # test that the objective function behaves as expected
    def test_onemax
```

```
5         assert_equal(0, onemax("0000"))
6         assert_equal(4, onemax("1111"))
7       end
8
9       # test the creation of random strings
10      def test_random_bitstring
11        assert_equal(10, random_bitstring(10).length)
12        assert_equal(0, random_bitstring(10).delete('0').delete('1').length)
13      end
14
15      # test the approximate proportion of 1's and 0's
16      def test_random_bitstring_ratio
17        s = random_bitstring(1000)
18        assert_in_delta(0.5, (s.delete('1').length/1000.0), 0.05)
19        assert_in_delta(0.5, (s.delete('0').length/1000.0), 0.05)
20      end
21
22      # test that members of the population are selected
23      def test_binary_tournament
24        pop = Array.new(10) {|i| {:fitness=>i} }
25        10.times {assert(pop.include?(binary_tournament(pop)))}
26      end
27
28      # test point mutations at the limits
29      def test_point_mutation
30        assert_equal("0000000000", point_mutation("0000000000", 0))
31        assert_equal("1111111111", point_mutation("1111111111", 0))
32        assert_equal("1111111111", point_mutation("0000000000", 1))
33        assert_equal("0000000000", point_mutation("1111111111", 1))
34      end
35
36      # test that the observed changes approximate the intended probability
37      def test_point_mutation_ratio
38        changes = 0
39        100.times do
40          s = point_mutation("0000000000", 0.5)
41          changes += (10 - s.delete('1').length)
42        end
43        assert_in_delta(0.5, changes.to_f/(100*10), 0.05)
44      end
45
46      # test recombination
47      def test_uniform_crossover
48        p1 = "0000000000"
49        p2 = "1111111111"
50        assert_equal(p1, uniform_crossover(p1,p2,0))
51        assert_not_same(p1, uniform_crossover(p1,p2,0))
52        s = uniform_crossover(p1,p2,1)
53        s.length.times {|i| assert( (p1[i]==s[i]) || (p2[i]==s[i]) ) }
54      end
55
56      # test reproduce cloning case
57      def test_reproduce_clone
58        pop = Array.new(10) {|i| {:fitness=>i,:bitstring=>"0000000000"} }
59        children = reproduce(pop, pop.length, 0, 0)
60        children.each_with_index do |c,i|
61          assert_equal(pop[i][:bitstring], c[:bitstring])
62          assert_not_same(pop[i][:bitstring], c[:bitstring])
63        end
64      end
65
66      # test reproduce mutate case
67      def test_reproduce_clone
```

```
68        pop = Array.new(10) {|i| {:fitness=>i,:bitstring=>"0000000000"} }
69        children = reproduce(pop, pop.length, 0, 1)
70        children.each_with_index do |c,i|
71          assert_not_equal(pop[i][:bitstring], c[:bitstring])
72          assert_equal("1111111111", c[:bitstring])
73          assert_not_same(pop[i][:bitstring], c[:bitstring])
74        end
75      end
76
77      # test reproduce size mismatch
78      def test_reproduce_mismatch
79        pop = Array.new(10) {|i| {:fitness=>i,:bitstring=>"0000000000"} }
80        children = reproduce(pop, 9, 0, 0)
81        assert_equal(9, children.length)
82      end
83  end
```

Listing 2: Unit Tests for the Genetic Algorithm the Ruby Programming Language

4 Rules-of-Thumb

Unit testing is easy, although writing good unit tests is difficult given the complex relationship the tests have with the code under test. Testing Metaheuristics and Computational Intelligence algorithms is harder again given their probabilist nature and their ability to 'work in-spite of you', that is, provide some kind of result even when implemented with defects.

The following provide 10 guidelines that may help when it comes time to unit testing an algorithm:

- *Start Small*: Some unit tests are better than no unit test and each additional test can improve the trust and potentially the quality of the code. For an existing algorithm implementation, start by writing a test for a small and simple behavior and slowly build up a test suite.
- *Test one thing*: Each test should focus on verifying the behavior of one aspect of one unit of code. Writing concise and behavior-focused unit tests are the objective of the methodology.
- *Test once*: A behavior or expectation only needs to be tested once, do not repeat a test each time a given unit is tested.
- *Don't forget the I//O*: Remember to test the inputs and outputs of a unit of code, specifically the pre-conditions and post-conditions. It can be easy to focus on the decision points within a unit and forget its primary purpose.
- *Write code for testability*: The tests should help to shape the code they test. Write small functions or modules, think about testing while writing code (or write tests first), and refactor code (update code after the fact) to make it easier to test.
- *Function independence*: Attempt to limit the direct dependence between functions, modules, objects and other constructs. This is related to testability and writing small functions although suggests limits on how much interaction there is between units of code in the algorithm. Less dependence means less side-effects of a given unit of code and ultimately complicated tests.
- *Test Independence*: Test should be independent from each other. Frameworks provide hooks to set-up and tear-down state prior to the execution of each test, there should be

no needed to have one test prepare data or state for other tests. Tests should be able to execute independently and in any order.

- *Test your own code*: Avoid writing tests that verify the behavior of framework or library code, such as the randomness of a random number generator or whether a math or string function behaves as expected. Focus on writing test for the manipulation of data performed by the code you have written.

- *Probabilistic testing*: Metaheuristics and Computational Intelligence algorithms generally make use of stochastic or probabilistic decisions. This means that some behaviors are not deterministic and are more difficult to test. As with the example, write probabilistic tests to verify that such processes behave as intended. Given that probabilistic tests are weaker than deterministic tests, consider writing deterministic tests first. A probabilistic behavior can be made deterministic by replacing the random number generator with a proxy that returns deterministic values, called a mock. This level of testing may require further impact to the original code to allow for dependent modules and objects to be mocked.

- *Consider test-first*: Writing the tests first in a Test-Driven methodology can help to crystallize expectations when implementing an algorithm from the literature, and help to solidify thoughts when developing or prototyping a new idea.

5 References

For more information on software testing, consult a good book on software engineering. Two good book's dedicated to testing are "Beautiful Testing: Leading Professionals Reveal How They Improve Software" that provides a compendium of best practices from professional programers and testers [5], and "Software testing" by Patton that provides a more traditional treatment [7].

Unit testing is covered in any good book on software testing. Two good books that focus on unit testing include "Test Driven Development: By Example" on the TDD methodology by Beck, a pioneer of Extreme Programming and Test Drive Development [1] and "Pragmatic unit testing in Java with JUnit" by Hunt and Thomas dedicated to the popular JUnit framework for Java [6].

6 Conclusions

This report provided an introduction to software testing, unit testing and provided heuristics for unit testing Metaheuristic and Computational Intelligence algorithms with examples.

References

[1] Kent Beck. *Test Driven Development: By Example*. Addison-Wesley Professional, 2002.

[2] Jason Brownlee. The clever algorithms project: Overview. Technical Report CA-TR-20100105-1, The Clever Algorithms Project http://www.CleverAlgorithms.com, January 2010.

[3] Jason Brownlee. Genetic algorithm. Technical Report CA-TR-20100303-1, The Clever Algorithms Project http://www.CleverAlgorithms.com, March 2010.

[4] Jason Brownlee. A template for standardized algorithm descriptions. Technical Report CA-TR-20100107-1, The Clever Algorithms Project http://www.CleverAlgorithms.com, January 2010.

[5] Adam Goucher and Tim Riley, editors. *Beautiful Testing: Leading Professionals Reveal How They Improve Software*. O'Reilly Media, 2009.

[6] Andrew Hunt and David Thomas. *Pragmatic unit testing in Java with JUnit*. Pragmatic Bookshelf, 2003.

[7] Ron Patton. *Software testing*. Sams, second edition edition, 2005.