Using Genetic Algorithms to Generate Test Suites

Austin Barket and Randall Hudson

Department of Computer Science
The Pennsylvania State University at Harrisburg
Middletown, PA 17057
`amb6470@psu.edu, rvh5220@psu.edu`

Abstract. This paper applies techniques of genetic algorithms to the problem of finding test suites that cover every edge or predicate in a programs control flow graph. It differs in other applications of genetic algorithms to software testing in that the organisms are test suites rather than test cases. Experimental results show....

1 Introduction

Software testing is a tedious and expensive process, costing as much as 50% of a projects budget [1]. One reason for this is the fact that the input domain for programs tends to be extremely large, often times infinite; this makes it impossible to test all possible inputs for a program [1]. Thus, testers spend a large portion of their time simply searching/designing good tests. One aspect of automatic testing research aims to alleviate this burden via the automatic generation of tests.

Automatic test generation can be characterized by the creation of some system that when fed information about a program will attempt to generate good tests for that program. In model based testing, one such method for automatic test generation, engineers must design a model of the system using the functional requirements for the system; the model then generates tests cases to test the functional requirements [2]. In rule based test generation, testers design a framework of rules, generally built around predicate statements; this framework then dictates the creations of test cases based on the predicate statements in the code [3]. These are two of the common methods used in automatic test generation. These methods generally require some upfront design costs before use.

Genetic algorithms represent a third, and perhaps more interesting, method of automatic test generation. Tests, or test suites, take the role of the organisms in this genetic algorithm while their coverage, e.g. branch, conditional, etc, determines their fitness [4]. This method represented an underexploited area of research and will be further examined in this paper.

1.1 Problem Statement

In high-level terms the problem succinctly presents itself as such: automatically generate good tests for a given program; but this leaves several open questions, such as "what information if any can be gotten from the program to help design the tests?" and "what constitutes a good test?" The field of structural testing will help answer both of these questions.

In any given program there are multiple paths in which flow of control can traverse; these can be represented by a control flow graph; predicates, i.e. conditional statements in the code, determine which path to take [1]. Structural testing encompasses the use of such information to design tests [1].

When evaluating a test's coverage, the predicates, and the flow of control they affect, play an important role. Consider the following simple metric: A test suite should cover every possible path in a program at least once. Using the control flow graph one can easily evaluate the quality of tests against this metric. A good test suite then becomes one in which most if not all of the possible paths are taken at least once. Structural testing uses this metric, and similar metrics, as its primary evaluation method for tests.

The problem to solve can now be stated with more detail: automatically create a test suite that covers all, or most, paths in a programs control flow graph. It is the authors' intention to solve this problem with a genetic algorithm.

To design a genetic algorithm that solves this problem it will need the following information: the input parameters for the program, the control flow graph, and the predicates affecting each path. The output would consist of a population of test cases or test suites, depending on design decisions, that should consist of good tests.

2 Summary of Research

2.1 Quick Survey

Using genetic algorithms to generate tests represents a relatively new area of research, with the first published results dating from only 1992 [5]. Initially, genetic algorithms evolved tests that would ensure all predicates for a single path of execution were true[6]. Later, Watkins created an adaptive fitness function that would create tests for one particular branch of the control flow graph and then switch to a different branch once some tests were found; this approach allowed a single run of the genetic algorithm to attempt to evolve the tests needed for full branch coverage[6]. Watson's results showed that a genetic algorithm, compared to random testing, required "an order of magnitude fewer tests to achieve path coverage"[6]

Up to this point, the fitness functions viewed a branch's predicate as just a boolean function, that is true or false; this did not take full advantage of white box testing in which the exact details of each predicate can be known. Sthammer wished to use information about these predicates to create boundary value tests. He accomplished this by using a distance function that reported the distance between a test's current values for a predicate and what values that predicate needed to be true. Sthammer's showed that his technique preformed better then a random search.

Pargas et al proposed the used of a control-dependence graph instead of a control flow graph to guide the search of test cases. They implemented this technique in a parallelizable algorithm called GenerateData[8]. When tested on programs with control flow graphs of cyclomatic complexity of seven significant improvements over random search were reported[8].

The newest genetic algorithms attempt to make use of past information in some way. In Beuno and Jino's research they used information about past runs of the genetic algorithm to influence the creation of the initial population used in a new run[6]. "The results with their six small test programs were promising."[6] Berndt et al try to take advantage of past information in a different way. Their algorithm includes a "fossil record that records past organisms, allowing any current fitness calculations to be influence by past generation."[6]

Those interested in a more detailed survey are encouraged to investigate the survey done by Aljahdali et al, on which this survey is partially based on [6].

2.2 Details of the Genetic Algorithm

The general structure of the genetic algorithms used in the above research changed very little since the early work done in this field and is generally quite simple: the inputs are encoded in binary, initial population is randomly created, single point cross over is used, uniform mutation is used, populations are generally between 25 and 100, and selection is done either at random or with fitness proportional selection [4][5][6][7][8].

3 Algorithm

TODO I wrote a bunch of stuff. I don't care for most of it and as you'll see I'm struggling to justify our algorithm. All this needs to be focused in, trimmed, reorganized and replaced with psuedocode where that makes more sense. Hopefully some of it can be used though or at least is helpful is trying to figure out what to say and what to leave out. It's crazy how much this changed in just the past few weeks. Comparing this to what we wrote in the proposal and even the progress report is interesting to say the least.

The domains of the input parameters of programs are typically massive, if not infinite in size. As a result the greatest challenge in solving the test suite generation problem is finding a way to narrow down the search space and devising a mechanism for the algorithm to learn which input parameters are likely to result in new coverage of the control flow graph. The problem is exacerbated further by the fact that programs are typically very sensitive to small changes in their input parameters and interactions and relationships between the parameters.

In order to tackle some of these challenges the algorithm proposed here attempts to emulate a very simplistic approach that human testers might take when trying to generate new test cases. Imagine a human tester analyzing a program for the first time. They might perform several, quick mental passes through the program tracing how input parameters in different ranges of those domains would cause shifts in the program's control flow. As a result of this quick analysis, they would narrow the potentially infinite domain of input parameters to a small set of values or ranges of values to use in test cases. As we will see, this is exactly the purpose served by the global range set in our algorithm.

Following this initial search, expert knowledge and reasoning would probably be used to figure out how to best combine parameters from these ranges into test cases and test suites. However one could imagine employing a more systematic approach to the problem of combining input parameters and evaluating the fruitfulness of these combinations based on the coverage obtained by the resulting test cases. As we will see this is the strategy employed by the genetic algorithm proposed in this paper.

The algorithm has three main stages. First an initial range search is performed to generate a global range set. The goal here is to find ranges of input parameters that seem promising and from which good test cases will likely be found. Then the main genetic algorithm commences, which attempts to evolve a population of test suites with diverse branch and multiple condition coverage. Finally a minimal sized, maximal coverage test suite is constructed from all the test cases across all organisms in the population. This final test suite is returned to the user, along with the percent of branch and multiple condition coverage achieved.

3.1 Global Range Set

A set of promising input parameter ranges is maintained throughout the search, from which the genetic algorithm can request new test cases during the initial population generation and test suite mutation steps. A range includes a start value and an end value, as well as an array of input parameter buckets. The buckets represent all the non-overlapping sub ranges of size 25 within the range. These buckets are mapped to counts of the number of times input parameters in those buckets were part of a test case that provided unique coverage population wide. The sum of all of these bucket counts forms a value referred to as the usefulness of that range.

This range set provides a problem independent mechanism for discovering and learning which input parameters are useful in generating good test cases.

Initialization: At the start of the algorithm a simple procedure is run to find promising ranges of input values and add them to the global range set. This procedure loops, starting with a range around 0 and extending in both directions towards max and min integer, generating a single test case from each range. These test cases are run through the control flow graph to determine which edges and predicates they cover. If a test case provides coverage that hasn't yet been seen during this initial procedure, its corresponding range is added to the global range set. This procedure can be repeated using various range sizes, which in some cases enabled to algorithm to find additional coverage and ranges not found in previous iterations. However this is an expensive procedure whose goal is to simply identify promising ranges that help focus the genetic algorithm's exploration. Thus for the purposes of the current research we found that performing two iterations, first with a range size of 5000, then with a range size of 2500 provided a good set of initial seed ranges for the genetic algorithm to utilize, while not taking too long to complete.

Range Set Adaptation: After several generations of no improvement in coverage at the population level, the range set is adapted based on the usefulness of its constituent ranges. Ranges that have a usefulness value greater than one standard deviation above the mean are split into two equal sized ranges. In addition the two ranges above and below the original range are added to the range set. These new adjacent ranges have the same size as the original range. Ranges with a usefulness value less one standard deviation from the mean are deleted from the range set. Finally a new random range is added to the range set to avoid missing out on ranges not found during the range set initialization. The idea behind this runtime adaptation is to give the algorithm the ability to learn which ranges are more useful than others, so that it can ultimately perform a more effective search for new test cases.

Test Case Generation: The range set provides two ways to generate new test cases. The first creates a test case by randomly selecting the values for all input parameters from a single range. This is used during the initial population generation described below.

The second uses repeated roulette wheel selection proportional to usefulness to select one range for each input parameter in the test case. In this way the algorithm generates test cases with input parameters stemming from the ranges that have been the most useful so far. The hope is that by combining parameters from several useful ranges into one test case, the new test case may provide coverage that would not be possible using parameters from just a single range. For example consider a multiple condition predicate that requires parameter 1 to be less than 500 but parameter 2 to be greater than 100000. In this situation ranges must be combined to form a test case that achieves the desired coverage.

3.2 Genetic Algorithm

Background and Encoding: The population of the genetic algorithm is made up of test suites, which are structures that hold a set of test cases. For the purposes of this research input parameters were restricted to the domain of 32 bit integers. Thus test cases are represented as an ordered list of integer input parameters, with the number of elements dictated by the control flow graph.

Control flow graphs have been implemented as a set of edges, set of predicates, and a set of blocks. Each edge in a CFG corresponds to a branch in the SUT, and connects two blocks. Covering all edges equates to achieving 100% branch coverage. Each predicate corresponds to a particular instance of either a simple or multiple condition in the SUT. Covering all predicates equates to achieving 100% multiple condition coverage. Each block represents a sequential block of code in the SUT, and is implemented as an executable function that simulates the behavior of the original software by implementing all control and data flows of the program.

The coverage of a test case is set by inputting the test's case's parameters into the CFG's run function. As the CFG executes it marks each edge and predicate covered by the test cases and returns the results. Three levels of coverage metadata are maintained by the algorithm for the purposes of fitness calculation. First each test case is assigned coverage by running it against the CFG. Each test suite then maintains counts of the number of time each edge and predicate was covered by its constituent test cases. Finally the population maintains sums of the coverage counts of all of the test suites.

Although each organism in the GA is a test suite, the goal is not necessarily to converge towards a single test suite containing the maximal possible coverage. Instead, the algorithm tries to evolve a population of test suites with diverse coverage of the control flow graph, such that the union of the coverage across all test cases in all test suites of the population is maximal.

Fitness: A fitness sharing scheme has been employed to assign fitness to organisms. Each edge and predicate has a base score associated with it, these scores are then split amongst all the test suites that cover each edge or predicate. This scheme is based off the observation that if a particular edge or predicate is covered by many organisms, then that edge or predicate is very easy to reach and thus no particular organism should be rewarded for covering it. However if an edge or predicate is covered by only one organism in the population. Then that organism should be rewarded and be selected more often.

Initialization: Each test suite is initialized in such a way that all input parameters of all test cases in a given test suite are randomly selected from a single range. In this way ranges found during the initial range search form the initial niches to be explored by the algorithm.

Each Generation: To this end test suites can better thought of as containers or niches for a set of input parameters. During each generation of the GA, some test cases in a test suite are broken down and recombined into test cases using k-point crossover. The recombined test cases are then subjected to small point mutations of each parameter. In this way the niche of input parameters in a selected test suite is further explored each generation.

After several generations of no improvement the algorithm performs k-point crossover between two selected test suites. This

Adaptive Parameters (Just mutation I think):

Local Optimization Heuristics: TODO

1. One algorithm when given an uncovered edge and a test case that covers an edge near the desired edge, will look in the neighborhood around that test case for parameters that cover the uncovered edge. The search is random inside the neighborhood and the neighborhood size expands several times before giving up.
2. The other two bias the search space to look around either the mean for the upper and lower bound on a given parameter or around zero. The idea is numbers around zero, such as 0, 1, and -1, can drastically effect the flow of control of a program; the stochastic nature of the genetic algorithm can cause it to have trouble finding these specific numbers in the search space, thus when the genetic algorithm gets stuck because it can not pick one of these numbers for a parameter, the local optimization will help it.

```
run()
```

```
TestCaseCrossover()
```

```
  $ts$ ← $P$.select()
  $tc1$ ← $ts$.getRandomTestCase()
  $tc2$ ← $ts$.getRandomTestCase()

  for $i = 0$ to 100
     $child1$, $child2$ ← crossover($tc1$, $tc2$)
     $child1$.mutate()
     $child2$.mutate()

     if $child1$ is covering anything not covered by ts
        $ts$.addTestCase($child1$)
        break
     else if $child2$ is covering anything not covered by ts
        $ts$.addTestCase($child2$)
        break
  end-for


TestSuiteCrossover()
  $ts1$ ← $P$.select()
  $ts2$ ← $P$.select()

 $child1$, $child2$ ← crossover($ts1$, $ts2$)
 $child1$.mutate()
 $child2$.mutate()



LocalOptimization()
```

4 Experiment

1. Control Flow Graphs: We have currently implemented three control flow graphs. These graphs are what the genetic algorithm tries to cover by finding test suites that when executed cover each edge and predicate.
   (a) A simple if else program
   (b) A program in which a user tries to guess a number
   (c) A program that, when given three numbers, will tell you what type of triangle has sides of those lengths.
2. Competing Algorithm: Implemented a random search algorithm that tries to find test cases to cover ever edges and predicates. This will be used to show our algorithm preforms better then random search.

5 Experimental Results

6 Conclusion

References

1. G. Meyers, *The Arts of Software Testing.* Hoboken, New Jersey: John Wiley & Sons Inc. 2nd edition, 2004.
2. M. Utting and B. Legeard, *Practical Model-Based Testing: A Tools Approach.* San Francisco: Morgan-Kaufmann. 2007.
3. K.-H. Chang, J. H. C. II, W. H. Carlisle and D. B. Brown *A framework for intelligent test data generation.* Journal of Intelligent and Robotic Systems vol. 5, no. 2, pp. 147-165, April 1992.
4. J. Wegener, H. Sthammer, B. Jones and D. Eyres, *Testing real-time systems using genetic algorithms.* Software Quality Journal. pp. 127-135, 1997.
5. X. S, E. C, S. C, L. G. A, K. S and K. K, *Applications of Genetic Algorithms to Software Testing.* in Internation Conference on Software Engineering and its Applications, Toulouse, 1992.**TODO: fix this citation**
6. S. Aljahdali, Taif, A. Ghiduk and M. El-Telbany, *The limitations of genetic algorithms in software testing.* in Computer Systems and Applications, Hammamet, 2010.
7. B. F. Jones, H. H. Sthammer and D. Eyres, *Automatic structural testing using genetic algorithms.* Software Engineering Journal, pp. 299-306, September 1996.
8. R. P. Pargas, M. J. Harrold and R. R. Peck, *Test-Data Generation Using Genetic Algorithms.* Journal of Software Testing, Verifcation, and Reliability, 1999.