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 TODO

- Add any missing citations
- Decide if to use FloatBarrier or not
- Our Work section
  - Begin with short, broad overview, then begin algorithm section
  - Pick Main Loop algorithm
  - FindRanges algorithm
  - AdaptRanges alg
  - For each algorithm, discuss it briefly in words
- Experimental results section
  - Discuss the programs we tested on
  - Maybe give a theoretical running time for a random searcher to find inputs for HardCFG
  - Give results (Tables and graphs)
- Discussion Section
  -
- Conclusion Section
  - Concluding remarks
  - Future work
- Read over the copy and pasted sections from proposal to make sure it all makes sense in this new context.
- Add a little bit more to the survey section, specifically more modern stuff

2 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.

3 Problem Definition

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 (CFG); 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.

4 Related Work

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].

TODO: add a few more recent papers

5 Our Work

Our approach focused on the idea of using test suites as an organism's chromosome rather than test cases. We also added some new evolutionary algorithm techniques that did not appear to be widely used in other literature for this problem. Additionally, we implemented several test programs of varying difficulty. Each of these items will now be discussed in turn.

As noted above, test suites compose the population and each of these consists of multiple test cases. The number of test cases for a given test suite was decided to be the sum of the number of edges and predicates in the target control flow graph; this would ensure every test suite would be able to cover all possible branches and predicates in the target program. Each test case contains an array of values that represent input parameters to the program being tested. For the purposes of this research, input parameters were restricted to the domain of 32-bit integers.

Although each organism in the GA is a test suite, the goal is not to converge the population towards a single test suite containing the maximal possible coverage. As the algorithm discovers test cases that offer new coverage, they are added to a final test suite maintained external to the population. This incrementally building of the final test suite, ensures that it exhibits all branch and multiple condition coverage at the end of the simulation. The organisms are then used as breeding grounds for test cases and are the vessels within which the main test case search is performed.

Many of the GAs from the literature for this problem are rather simplistic **TODO: add citations**. To improve upon them we implemented and tested many techniques, including several types of fitness scaling, several base fitness functions, several mutation operators for test suites, tournament selection, several types of local optimizations, several adaptive parameters et al Most of these were not fit enough to survive our culling. The subset that did survive includes local optimization for test cases, fitness sharing, and adaptive ranges. These are discussed in more detail in the algorithm sub-section.

TODO: Is the sentence about expanding predicates necessary? Also "Our implementation saves only..." sounds like were only storing edges not predicates, until you read the following sentence... Reword this to be clearer. Test programs are provided to the algorithm in the form of executable CFGs. A CFG represents linear blocks of instructions from the target program as nodes. The edges represent possible different directions the flow of control can follow when it reaches a predicate. When implementing multiple condition coverage, one could simply expand each predicate, turning it into one predicate for each possible outcome; this has its advantages but can make tracking branch versus MCC coverage harder. Our implementation saves edges only for possible directions of control in the original target program. A separate set for predicates is kept, and each node (aka block) is responsible for tracking its predicates. This means that in our implementation, covering all edges equates to achieving 100% branch coverage and covering all predicates equates to achieving 100% multiple condition coverage.

5.1 Algorithm

The following section has been broken down into the subsections for each major aspect of the proposed algorithm. These will be presented roughly in the order in which they are used by the main simulation loop. First the main simulation loop is introduced to give context throughout the rest of the sections. The notion of input parameter ranges and the simulation's range set is then introduced, followed by a discussion of the population initialization. Fitness calculation and association with organisms is explored. Then the main tools used for test case generation, including test case crossover, test case mutation, and local optimization are defined. Finally, the algorithm's perturbation operations, range set adaptation, test suite crossover, and test suite mutation are studied.

Main Simulation Loop The main simulation loop has several components that need to be address. First, it must keep track of how many generations it has been since no improvement was made. The simulation uses this variable as part of the ending predicate and in determining how often test suite crossover and range adaptation is preformed.

```
run(maxGens, numCutPoints, P_m)
   currentGen ← 0

   FindGoodRanges()
   BuildPopulation()

   do {
      updatePopulationsFitness()

      TestCaseCrossoverAndMutation(numCutPts, P_m)

      if currentGen is multiple of 10
         TryLocalOptimization()

      if no improvement for more than 20 generations or currentGen is multiple of 30 {
         AdaptRanges()
         TestSuiteCrossoverAndMutation(numCutPts, P_m)
      }

      currentGen ← currentGen + 1

   } while ending criteria has not been met
```

Fig. 1: Algorithm for main simulation loop

Initializing the Range Set The algorithm begins by executing a preprocessing procedure termed FindGoodRanges(). In this preprocessing step, the algorithm searches for promising ranges and adds them to the simulation's range set. A range includes a start value and an end value, as well as an array of input parameter buckets. The buckets are non-overlapping partitions of size 25 within the range, e.g. the range [0, 75] would contain the following array of buckets [[0, 24], [25, 49], [50, 74]]. 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 for the population. The sum of all of the buckets in a given range is considered the usefulness of that range. As we will see the algorithm actively maintains and adapts this range set based on information learned regarding the usefulness of its constituent ranges. The GA pulls ranges from the range set any time a test case needs to be created, and if a generated test case offers new coverage to the final test suite, the buckets within which the test case's parameters fall are incremented accordingly.

TODO: Merge these two descriptions to come up with one cohesive paragraph that introduces ranges and the range set. I considered just deleting the paragraph below but we'll discuss it first.

At the start of the algorithm the FindGoodRanges() 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. 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.

```
FindGoodRanges()
   rangeSet ← Empty Range Set

   for ( size ∈ {5000, 2500} ) {
      posStart ← −size/2, negStart ← size/2

      while posStart + size < max_int and negStart − size > min_int {
         posRange ← [posStart, posStart + size]
         negRange ← [negStart, negStart - size]

         posTC ← test case in posRange
         negTC ← test case in negRange

         if (posTC covers anything new) {
            add posRange to rangeSet
         }

         if (negTC covers anything new) {
            add negRange to rangeSet
         }

         posStart ← posStart + size
         negStart ← negStart − size
      }
   }

   return rangeSet
```

Fig. 2: Algorithm for finding good ranges

Initializing the Population During initialization each organism in the population uses only a single range from the simulation's range set to generate all its test cases. The main test case search occurs by crossing over and mutating test cases within organisms, as well as applying local optimization heuristics that manipulate or replace an organism's test cases. So the algorithm begins by thoroughly searched the space of possible test cases composed of parameters from singular ranges of the range set. After many generations of no improvements using test case crossover and local optimization, the algorithm performs two key perturbation procedures. First the range set adapts itself based on information learned regarding the fruitfulness of its constituent ranges. Then test suite crossover and mutation are applied to mix up the available input parameters inside of test suites. These perturbation procedures will be described in more detail below, but the idea is to attempts performs crossover and mutation on test suites. 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

Coverage Metadata 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 times its constituent test cases covered each edge and predicate. Finally the population maintains sums of the coverage counts across all of the test suites.

Fitness Sharing The algorithm assigns fitness for an organism based off both its coverage and the population's coverage; this is a form of fitness sharing. Each edge has a base score associated with it. These scores are then split amongst all the test suites that cover it. The motivation for using fitness sharing is the follow: if many organisms cover a particular edge, then it must be easy to reach and thus organisms should receive less fitness for covering it; however, if only one organism in the population covers an edge, then that organism should receive a higher fitness.

Test Case Generation and Search TODO: Brief introduction to these operations

Test Case Crossover and Mutation Every generation, a single organism is selected proportional to its fitness. The test cases within the selected organism are then subjected to the test case crossover and mutation procedure given below. Simple k-point crossover is used. This operator maintains the values of the input parameters and swaps them around to generate the two child test cases. Each of these test cases are then subjected to small point mutations. That is with probability P_m, a small delta value calculated using a gaussian distribution from the orginal value is added or subtracted to get the new parameter value. Because test case crossover is more important it is attempted several times before giving up.

```
TestCaseCrossover(cutPts, P_m)
   Proportionally select a test suite, ts, from the population

   for i = 0 to 100
      From ts  select two random test cases, tc1 and tc2

      child1, child2 ← crossover(tc1, tc2, cutPts)
      child1.mutate(P_m)
      child2.mutate(P_m)

      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
```

Fig. 3: Algorithm for test case crossover

Local Optimization Two simple local optimization techniques are used, but the structure of both is very similar. The LocalOptFromParameters function will take a test case and, beginning with a small neighbor size, will begin looking at other test cases in the neighborhood of the supplied one.

The LocalOptFromZero function tries to pull the test case's parameters closer to zero. This was added because values in the neighbor hood of zero, such as 1, -1, or 0, can drastically alter the flow of the program being tested, but because the search space was so large, the simulation was unable to find these values without this.

Both functions explore each neighborhood several times before expanding the neighborhood. If at any point, a test case is found that covers something not in the population then it is added to the organism by replacing a duplicate test case.

```
TryLocalOptimization()
      org ←P.Select()

      Randomaly select one of the two local opt functions and try it on a
      duplicate test case from org, store results in tc

      if tc  is not null replace a duplicate test case in org  with tc
```

Fig. 4: Algorithm for local optimization

```
LocalOptFromZero(testCase)
   neighborhoodSize ← 5
   for i = 1 to 1000
      temp ←testCase
      for j = 1 to 150
         for each parameter, p in temp
            95% of the time set p to a random value in the neighbor of zero
            5% of the time set p to p/2
         end-for
         if temp covers anything not covered in the population return temp
      end-for
      neighborhoodSize ←neighborhoodSize + 5
   end-for

   if no improvement was ever found then return null
```

Fig. 5: Algorithm for local optimization

```
LocalOptFromParameters(testCase)
   neighborhoodSize ← 5
   for i = 1 to 500
      temp ←testCase
      for j = 1 to 150
         for each parameter, p in temp
            set p to a random value in the neighborhood of p
         end-for
         if temp covers anything not covered in the population return temp
      end-for
      neighborhoodSize ←neighborhoodSize + 5
   end-for

   if no improvement was ever found then return null
```

Fig. 6: Algorithm for local optimization

Perturbation Operations

Adapting the Range Set The initial search for good ranges will probably fail to find all useful ranges; it may also think some ranges are useful which actually

are not. Range adaptation aims to give the simulation the ability to remove non-useful ranges, as well as focus in on the most fruitful ranges.

After several generations of no improvement, the range set is adapted based on the usefulness of its ranges. First, the range set deletes any ranges below one standard deviation usefulness. All ranges above one standard deviation usefulness are split into two equal sized ranges, with the input parameter buckets copied accordingly to the new ranges. The two ranges immediately above and below the original range are also added to the range set. These new adjacent ranges have the same size as the original range and their usefulness values are set to half that of the original range in order to prevent them being completely ignored. Finally, the range set adds a new random range into its pool to avoid missing ranges not found during initialization or a previous range adaptation.

```
AdaptRanges()
   for each range in rangeSet {
      if (range.usefulness < μ − σ) {
         remove range from the range set
      }

      if (range.usefulness > μ + σ) {
         addAdjacentRanges(range)
         splitRangeIntoTwo(range)
      }
   }

   add new random range to the RangeSet
```

Fig. 7: Algorithm for adapting ranges

Test Suite Crossover and Mutation Test suites are normal k-point crossover. After test suite crossover, the algorithm always replaces both parents.

6 Experimental Results

6.1 Test Programs

7 Discussion

8 Conclusion

8.1 Future Work

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.